Massimo Ruocchio

Welcome to Oracle 12c

Corso introduttivo al Database Oracle

Architettura
SQL
PL/SQL
C come Cloud

http://oracleitalia.wordpress.com
ISBN 978-1-291-82092-8

Copyright © 2011-2014 di Massimo Ruocchio. Tutti i diritti riservati.

Sommario

Sommario...3
Introduzione 9
1 Architettura di Oracle 11
1.1 Istanza e database...11
1.2 Istanza - Processi di base...11
1.2.1 PMON – Process Monitor 12
1.2.2 SMON – System Monitor 12
1.2.3 DBWn – Database Writer 12
1.2.4 LGWR – Log Writer..12
1.2.5 CKPT – Checkpoint 12
1.3 Istanza - Strutture di memoria 12
1.3.1 SGA – System Global Area 12
1.3.2 PGA – Program Global Area 13
1.4 Database - Strutture per la memorizzazione dei dati 13
1.4.1 Datafile e blocchi...13
1.4.2 Extent..14
1.4.3 Tablespace 14
1.4.4 Segment..15
1.5 Database - File di controllo e di log 15
1.5.1 Il control file...15
1.5.2 I Redo Log file...15
2 Installazione 17
3 Operazioni preliminari 19
3.1 Avviare ed arrestare Oracle...19
3.1.1 Startup 20
3.1.2 Shutdown 20
3.2 Connettersi al database...21
3.2.1 Utenti e schemi 21
3.2.2 Connettersi dal server dove è installato il db.........................22
3.2.3 Connettersi da un client remoto...22
3.2.4 Creare un nuovo utente 31
4 SQL*Plus e SQL Developer...32
4.1 SQL*Plus 32
4.1.1 Perché ancora SQL*Plus 33
4.1.2 Personalizzare SQL*Plus (Windows) 33

4.1.3 Parametri e comandi di SQL*Plus 36
4.1.4 Scrittura e modifica di un'istruzione SQL 38
4.1.5 esecuzione di uno script SQL 41
4.1.6 Cambio del prompt SQL 42
4.2 SQL Developer 43
4.2.1 Avvio di SQL Developer 43
4.2.2 Parametri e comandi di SQL Developer 45
5 Oggetti del DB 46
5.1 Il dizionario dati 46
5.2 Tabelle 49
5.2.1 Colonne di una tabella 50
5.2.2 Constraint 51
5.3 Indici 52
5.3.1 Indici B-Tree 53
5.3.2 Indici Bitmap 54
5.3.3 Più indici sullo stesso insieme di colonne 54
5.4 IOT 55
5.5 Cluster di tabelle 56
5.6 Viste 56
5.6.1 Check option constraint 57
5.7 Viste materializzate 57
5.8 Sequenze 58
5.9 Sinonimi 58
5.10 Database link 59
5.11 Oggetti PL/SQL 59
5.11.1 Procedure 59
5.11.2 Funzioni 59
5.11.3 Package 59
5.11.4 Trigger 60
5.12 Estensioni Object-Oriented 60
5.13 XML DB 60
6 CASE STUDY 61
6.1 Progettazione 61
6.2 Il database d'esempio 61
6.2.1 Tabelle 62
6.2.2 Dati 64
6.3 SCOTT/TIGER 65
7 SQL 67
7.1 Tipi di dato 68
7.1.1 Dati alfanumerici 69
7.1.2 Numeri 70
7.1.3 Date ed orari 71
7.1.4 Dati binari 71
7.2 Comandi DDL 72
7.2.1 CREATE 72
7.2.2 ALTER 91
7.2.3 RENAME 101
7.2.4 DROP 102
7.2.5 TRUNCATE 104

7.2.6 PURGE 107
7.2.7 FLASHBACK TABLE 108
7.3 Comandi DML 109
7.3.1 Valori fissi 109
7.3.2 INSERT 110
7.3.3 UPDATE 124
7.3.4 DELETE 126
7.3.5 MERGE 127
7.3.6 Gestione degli errori in SQL con LOG ERRORS 131
7.4 SELECT, il comando di ricerca 137
7.4.1 Proiezioni 137
7.4.2 La tabella DUAL 140
7.4.3 Pseudo colonne 141
7.4.4 Selezioni 143
7.4.5 Ordinamento 160
7.4.6 Raggruppamenti 164
7.4.7 Tornando su ROWNUM 173
7.4.8 Le clausole PIVOT ed UNPIVOT 175
7.4.9 Query gerarchiche 179
7.4.10 Paginazione dei record estratti in SQL 187
7.4.11 Pattern matching di righe 191
7.4.12 La clausola MODEL 195
7.5 Manipolazione dei dati 206
7.5.1 Operatore di concatenazione 206
7.5.2 Operatori aritmetici 207
7.5.3 Funzioni sulle stringhe 208
7.5.4 Funzioni sui numeri 218
7.5.5 Funzioni sulle date 222
7.5.6 Funzioni di conversione 229
7.5.7 Formati per la rappresentazione delle date 232
7.5.8 Formati per la rappresentazione dei numeri 235
7.5.9 Gestione dei valori nulli 236
7.5.10 Espressioni regolari 241
7.5.11 DECODE e CASE 245
7.5.12 GREATEST e LEAST 249
7.6 JOIN: ricerche da più tabelle 250
7.6.1 Sintassi per la JOIN 250
7.6.2 Prodotto cartesiano di tabelle 251
7.6.3 Inner join 253
7.6.4 Outer join 255
7.6.5 Le clausole APPLY e LATERAL 264
7.7 Gli operatori insiemistici 267
7.7.1 UNION 267
7.7.2 UNION ALL 270
7.7.3 INTERSECT 271
7.7.4 MINUS 271
7.8 Ricerche innestate 272
7.8.1 Operatori di confronto 273
7.8.2 Subquery correlate e scorrelate 277

7.8.3 La clausola WITH 279
7.9 Comandi TCL 280
7.9.1 Gestione delle transazioni 280
7.9.2 COMMIT 280
7.9.3 ROLLBACK 282
7.9.4 SAVEPOINT 283
7.9.5 SET TRANSACTION 285
7.9.6 Lock di dati ed oggetti 289
7.9.7 Una situazione particolare, il deadlock 290
7.10 Comandi DCL 291
7.10.1 GRANT 291
7.10.2 REVOKE 292
7.10.3 RUOLI 293
8 PL/SQL 294
8.1 Blocchi PL/SQL anonimi 294
8.2 Struttura di un blocco PL/SQL anonimo 295
8.3 Dichiarazione di costanti e variabili 297
8.3.1 Tipi di dato 297
8.3.2 Dichiarazione di variabili 298
8.3.3 Dichiarazione di costanti 299
8.4 Costrutti di base 299
8.4.1 Assegnazione di un valore. 299
8.4.2 Strutture Condizionali 299
8.4.3 Strutture Iterative 302
8.4.4 Strutture Sequenziali 305
8.5 Utilizzo dell'SQL 307
8.5.1 SQL Statico 307
8.5.2 SQL Dinamico 308
8.5.3 CURSORI 312
8.6 Gestione delle eccezioni 315
8.6.1 Blocco Exception 315
8.6.2 Eccezioni predefinite 316
8.6.3 SQLCODE e SQLERRM 320
8.6.4 Sollevare un'eccezione 321
8.6.5 Eccezioni utente 322
8.6.6 Eccezioni mute 323
8.6.7 Eccezioni in un ciclo 323
8.6.8 Propagazione delle eccezioni 325
8.7 Tipi di dato complessi 326
8.7.1 Record 326
8.7.2 Array associativi (PL/SQL Tables) 329
8.7.3 Varray 333
8.7.4 Nested Table 335
8.7.5 Bulk collect 336
8.7.6 Metodi delle collezioni 337
8.8 Oggetti PL/SQL nel DB 344
8.8.1 Procedure 344
8.8.2 Funzioni 352
8.8.3 Package 357

8.8.4 Database Trigger 365
8.8.5 Dipendenze tra oggetti del DB 378
8.8.1 La clausola ACCESSIBLE BY 381
8.8.2 Diritti d'esecuzione 382
8.8.3 La clausola BEQUEATH delle viste 385
8.8.4 Controllo del codice PL/SQL 387
8.8.5 Le direttive di compilazione PRAGMA 390
9 Cloud e Multitenant Architecture 398
9.1 C come Cloud 398
9.2 Introduzione alla Multitenant Architecture 399
9.3 Containers in un CDB 400
9.3.1 Il container CDB$ROOT 400
9.3.2 I container PDB 400
9.3.3 Container corrente 401
9.3.4 Data Dictionary in un CDB 401
9.4 Utenti e ruoli in un CDB 402
9.4.1 Utenti comuni o locali 402
9.4.2 Ruoli comuni o locali 403
9.5 Architettura fisica di un CDB 404
9.6 Operazioni amministrative in un CDB 404
9.6.1 Avviare ed arrestare Oracle 404
9.6.2 Connettersi al database 405
9.6.3 Unplug e trasferimento di un PDB 406
9.6.4 Duplicazione di un PDB 407
9.6.5 PDB Trigger 409
10 Prerequisiti 413
10.1 Database e database relazionali 413
10.1.1 database 413
10.1.2 database relazionali 414
10.1.3 DBA 415
10.2 Progettazione di un db relazionale 415
10.2.1 Modello entità/relazioni 415
10.2.2 Normalizzazione 416
10.2.3 Definizione dello schema fisico 419
10.3 Architettura di un computer 419
10.3.1 Architettura Hardware 420
10.3.2 Architettura Software 420
10.4 Memorizzazione dei dati 421
10.4.1 I sistemi binario ed esadecimale 421
10.4.2 Dal bit al TeraByte 422
10.4.3 La tabella ASCII 423
10.5 Rappresentazioni numeriche 423
10.5.1 Virgola fissa 423
10.5.2 Virgola mobile 424
10.6 Algoritmi di ricerca 424
10.6.1 Ricerca sequenziale 424
10.6.2 Ricerca dicotomica (o binaria) 424
10.7 Algoritmi e funzioni di HASH 425
10.8 Teoria degli insiemi 425

10.8.1 Insiemi e sottoinsiemi 425
10.8.2 Intersezione di insiemi 426
10.8.3 Unione di insiemi 426
10.8.4 Complemento di insiemi 426
10.8.5 Prodotto cartesiano di insiemi 426
10.9 Logica delle proposizioni 426
10.9.1 Operatore di congiunzione 427
10.9.2 Operatore di disgiunzione 427
10.9.3 Operatore di negazione 427
10.9.4 Regole di precedenza 427
10.9.5 Leggi di De Morgan 428
10.10 Logica delle proposizioni e teoria degli insiemi 428
Indice Analitico 430
Indice delle figure 437

Introduzione

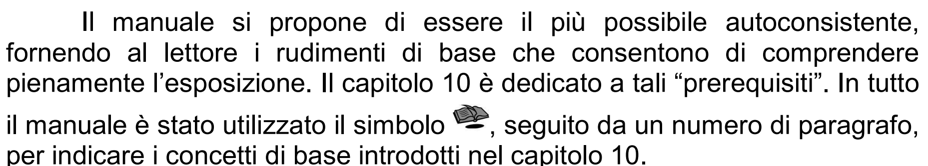
Welcome To Oracle nasce da una lunga esperienza di docenza in corsi su Oracle. Ho conosciuto la tecnologia Oracle nel 1996 ed ho cominciato ad insegnarla nel 1998, continuo tutt'ora sia ad utilizzarla per scopi professionali sia a tenere corsi.

Ho sempre incontrato una certa difficoltà a reperire il materiale didattico da fornire ai miei studenti. Un principiante ha bisogno di un manuale che parta proprio dalle basi, senza dare nulla per scontato, ma che allo stesso tempo si spinga abbastanza avanti da soddisfare le menti più curiose e veloci nell'apprendimento.

Dopo avere rinunciato a trovare un manuale adatto alle mie esigenze ho deciso di scriverlo, a giugno 2011 ho pubblicato la prima versione di "Welcome to Oracle". Questa nuova edizione incorpora le novità di Oracle 12c, disponibile dall'estate 2013.

Ho cercato di produrre un manuale che si possa leggere a più livelli. Un principiante deve essere, leggendo questo libro, in grado di comprendere gli elementi architetturali di base ed apprendere i primi rudimenti di SQL e PL/SQL. Un lettore di media esperienza deve finalmente comprendere pienamente l'architettura di Oracle, afferrandone alcuni dettagli essenziali, ed approfondire gli aspetti di SQL e PL/SQL che non ha mai avuto il tempo di guardare. Un lettore esperto deve poter, con questo manuale, rimettere ordine nelle sue conoscenze e trovare spunti per nuove indagini approfondite su temi "avanzati". In fondo c'è sempre qualcosa da imparare, qualunque sia il proprio livello di conoscenza ed esperienza.

Nel manuale sono stati utilizzati molti termini inglesi di uso consueto nell'ambiente tecnico italiano. Una traduzione del termine *tablespace*, ad esempio, sarebbe stata una forzatura non giustificata. Si sono invece tradotti in italiano i termini comuni come *tabella*, *vista* o *sequenza*.



Il manuale si propone di essere il più possibile autoconsistente, fornendo al lettore i rudimenti di base che consentono di comprendere pienamente l'esposizione. Il capitolo 10 è dedicato a tali "prerequisiti". In tutto il manuale è stato utilizzato il simbolo, seguito da un numero di paragrafo, per indicare i concetti di base introdotti nel capitolo 10.

Questo manuale nasce senza l'iter consueto di una casa editrice dove, prima di arrivare alla pubblicazione, l'opera è letta e riletta dall'autore e dai revisori di bozze. Dichiaro fin d'ora la mia gratitudine verso tutti coloro che vorranno segnalarmi errori di qualunque genere e possibili miglioramenti del manuale. Potete inviare tutte le segnalazioni all'indirizzo email

Welcome.to.oracle@gmail.com

Aprile 2014 Massimo Ruocchio

1 Architettura di Oracle

1.1 Istanza e database

Nel linguaggio comune si utilizza genericamente il termine database (10.1) per indicare sia i dati, strutturati in tabelle, che bisogna gestire sia le componenti software che ne consentono la gestione. Anche nei comuni database commerciali, Oracle incluso, questi due concetti sono entrambi presenti e spesso confusi. In questo capitolo si cercherà di fare chiarezza descrivendo l'architettura del database Oracle.

Un'*istanza* Oracle è l'insieme di alcuni processi di base (programmi della Oracle che girano sul server in cui il DB è installato) e della porzione di memoria RAM (10.3.1) che il server dedica a questi processi ed in generale al funzionamento del DB. L'istanza, dunque, non contiene tabelle né singoli dati. La definizione delle tabelle e tutti i dati in esse contenuti si trovano, invece, su alcuni file posti sul disco fisso del server (o su un disco esterno al server ma da esso raggiungibile). Il *database* Oracle è l'insieme di questi file e di alcuni altri file di sistema anch'essi residenti sul disco fisso.

Nell'architettura standard di Oracle ad ogni istanza corrisponde esattamente un database. Quando, invece, Oracle è installato in modalità "Real Application Clusters" un solo database è servito da diverse istanze che girano su server diversi collegati tra loro, in modo che l'eventuale danneggiamento di un server non pregiudichi la disponibilità del database.

In Oracle 12c è stata introdotta una grande novità: la Multitenant Architecture. Questa consente di gestire più database con una sola istanza, abilitando il cloud computing. Il capitolo 9 di questo libro è interamente dedicato a questa novità.

Nei prossimi paragrafi saranno descritte più dettagliatamente le componenti di un'istanza e di un database.

1.2 Istanza - Processi di base

Un'istanza Oracle include almeno i seguenti processi obbligatori:

1.2.1 PMON – Process Monitor

Si occupa di monitorare gli altri processi e riavviarli nel caso in cui terminino in maniera inattesa.

1.2.2 SMON – System Monitor

Si occupa del monitoraggio del sistema e di ottimizzare alcune aree di memoria fisica.

1.2.3 DBWn – Database Writer

Ha il compito di scrivere nei file archiviati sul disco i dati modificati dall'utente in memoria RAM.

In genere è presente uno solo di questi processi (DBW0) ma all'occorrenza se ne possono far partire altri (DBW1, DBW2, ecc..).

1.2.4 LGWR – Log Writer

Si occupa di tenere sempre aggiornati i Redo Log Files (vedi paragrafo 1.5), che contengono tutte le operazioni effettuate sul DB e consentono di ricostruire uno stato coerente del DB in caso di ripartenza dopo un errore grave.

1.2.5 CKPT – Checkpoint

Tiene aggiornato il control file (vedi paragrafo 1.5) e guida i DBWn indicandogli quando possono scrivere.

Oltre a questi processi obbligatori, sempre attivi, molti altri possono essere attivati o meno in funzione delle opzioni di Oracle che si è scelto di utilizzare.

Nella categoria dei processi di base rientrano anche i processi server. Questi consentono ai programmi esterni di aprire connessioni con il DB. Il meccanismo di connessione al database sarà ripreso ed approfondito nel paragrafo 3.2.

1.3 Istanza - Strutture di memoria

Quando l'istanza viene avviata Oracle fa partire i processi di base descritti nel paragrafo precedente ed alloca la memoria RAM necessaria al funzionamento del DB:

1.3.1 SGA – System Global Area

È un'area di memoria condivisa da tutti i processi di base. È composta principalmente dalle seguenti strutture di memoria:

- **Database Buffer Cache**: include tutti i blocchi contenenti dati prelevati dal database (quindi dal disco fisso) e messi in RAM per essere letti e/o modificati.
- **Redo Log Buffer**: contiene le informazioni che saranno scritte dal LGWR sui Redo Log Files.
- **Shared Pool**: contiene informazioni di sistema (parametri, istruzioni SQL o PL/SQL già parzialmente elaborate, contenuto del dizionario dati) che sono state già utilizzate e potrebbero essere di nuovo utili per successive elaborazioni.
- **Large Pool**: è un'area di memoria opzionale destinata a contenere ciò che è troppo grande per essere contenuto nella Shared Pool.
- **Java Pool**: è un'area dedicata a contenere i programmi java che vengono eseguiti nell'ambito del DB

È importante sottolineare che la SGA, essendo condivisa, è utilizzata da tutti i processi che accedono al DB. Per fare un esempio, ipotizziamo che, mediante un'applicazione web, un utente visualizzi i dati del cliente 111111. Oracle accede al disco fisso dove sono conservati i dati del cliente e li porta in memoria RAM, mettendoli nella Database Buffer Cache. Successivamente i dati vengono restituiti alla pagina web e visualizzati. Se un secondo utente si collega al DB, anche in un'altra modalità, ad esempio utilizzando il tool Oracle Sql*Plus (vedi capitolo 4), e cerca i dati del cliente 111111 Oracle potrà rispondere leggendo direttamente la SGA, senza dover di nuovo accedere al file contenente i dati che si trova sul disco fisso. Ciò consente un grande miglioramento dei tempi di risposta perché un accesso al disco fisso è enormemente più lento di un accesso alla memoria RAM.

1.3.2 PGA – Program Global Area

È un'area di memoria non condivisa il cui utilizzo è riservato ai processi di base. Ogni processo di base dispone della sua porzione di PGA.

1.4 Database - Strutture per la memorizzazione dei dati

1.4.1 Datafile e blocchi

Come già accennato in precedenza, i dati vengono archiviati nel database in file che si trovano sul disco fisso. Questi file prendono il nome di *datafile*. Per gestire con grande precisione i singoli dati Oracle suddivide ogni datafile in blocchi (*data block*) di una dimensione fissa, solitamente quattro oppure otto kilobyte(10.4.2). La dimensione del blocco viene decisa in fase di installazione del DB e non può più essere modificata successivamente.

All'interno di un blocco ci possono essere diverse righe della stessa tabella, non righe di tabelle diverse (a meno di una piccola eccezione, i cluster di tabelle, di cui diremo più avanti). Ogni blocco è composto da

- una testata, in cui Oracle conserva informazioni sul blocco e sui dati che esso contiene,
- uno spazio libero da utilizzare nel caso in cui i dati contenuti nel blocco vengano modificati in momenti successivi all'inserimento,
- i dati di una serie di righe appartenenti ad una stessa tabella,
- spazio vuoto da utilizzare per ulteriori righe della stessa tabella.

I blocchi non vengono riempiti mai per intero. Ogni volta che un utente inserisce una riga in una tabella, Oracle aggiunge una nuova riga nel blocco corrente facendo sempre attenzione a lasciare libero lo spazio riservato per gli aggiornamenti dei dati. Se Oracle determina che l'inserimento della nuova riga nel blocco farebbe diminuire eccessivamente lo spazio libero, passa al blocco successivo ed inserisce lì la nuova riga. La dimensione dello spazio libero minimo per ogni blocco può essere definita in fase di creazione del database o delle singole tabelle.

1.4.2 Extent

Vista l'esigua dimensione di un blocco, questo potrà contenere poche righe di una tabella. Quante righe possono essere contenute in un blocco dipende dal numero di dati presenti in tabella e da quanto spazio questi occupano. In ogni caso sarebbe inefficiente per Oracle dover riservare un blocco alla volta ogni volta che il blocco corrente è saturo. Per semplificare le cose Oracle riserva alla tabella un certo numero di blocchi consecutivi presenti nel datafile. Un insieme di blocchi consecutivi che vengono prenotati da Oracle tutti per la stessa tabella prendono il nome di *extent*. Dunque possiamo dire che la parte già utilizzata di un datafile è logicamente suddivisa in tanti extent. Ogni extent è un insieme di blocchi consecutivi tutti dedicati a contenere i dati della stessa tabella. Quando Oracle non ha più spazio per inserire nell'ultimo blocco disponibile nell'extent corrente prenota un nuovo extent. Quanti blocchi compongono l'extent? Dipende. Si può fare in modo che sia Oracle a gestire in autonomia questo aspetto oppure lasciare al DBA la gestione degli extent. La scelta di default, fortemente consigliata da Oracle, è la prima.

1.4.3 Tablespace

Oracle mette a disposizione dei contenitori logici per le tabelle che possono essere costituiti da uno o più datafile. Questi contenitori prendono il nome di *tablespace*. In pratica quando si crea una tabella bisogna che Oracle sappia in quale tablespace deve andare a mettere i dati, sceglierà poi autonomamente quale dei file del tablespace utilizzare. I dati di una tabella stanno sempre tutti sullo stesso tablespace, ma possono essere scritti in datafile diversi. Il tablespace che si intende utilizzare può essere esplicitamente indicato in fase di creazione della tabella, oppure Oracle utilizzerà un tablespace di default.

1.4.4 Segment

Fin qui abbiamo descritto le modalità di memorizzazione dei dati delle tabelle. In un database, però, ci possono essere anche altri tipi di oggetto. Il capitolo 5 è dedicato ad una carrellata sui diversi oggetti che si possono trovare in un database Oracle, alcuni di questi contengono dati e dunque occupano spazio come le tabelle, altri no. Gli oggetti del primo tipo vengono detti *segment*. Tutto ciò che abbiamo detto nel paragrafo 1.4 si adatta a tutti i segment, di cui le tabelle sono un caso particolare.

1.5 Database - File di controllo e di log

Nel paragrafo precedente abbiamo visto che un database è composto principalmente dai datafile, cioè i file che contengono i dati. Altri file però hanno un ruolo essenziale nella vita del DB.

1.5.1 Il control file

Il *control file* (file di controllo) è un file che contiene informazioni continuamente aggiornate da Oracle che sono essenziali per il funzionamento del DB. In particolare il control file contiene informazioni sui Redo Log file e sulle operazioni di log switching e checkpoint. Dei Redo Log file e del log switching parleremo nel prossimo paragrafo, il checkpoint è probabilmente l'operazione più importante eseguita dal DB. Si tratta della scrittura dei blocchi modificati sul disco. Quando un utente inserisce o modifica i dati di una tabella, infatti, questi non vengono subito modificati nel datafile, ma in blocchi che si trovano nella SGA, quindi in memoria RAM. Il processo di base CKPT periodicamente decide di eseguire l'operazione di checkpoint, dà ordine al processo DBW0 di eseguire la scrittura e registra quest'informazione nel control file. Senza il checkpoint tutte le modifiche fatte andrebbero perse allo spegnimento del server, con l'azzeramento della memoria RAM. Al riavvio del server Oracle dovrebbe recuperare tutte le operazioni fatte dagli utenti utilizzando i Redo Log file, di cui si parlerà più avanti.

Il control file è definito in fase di creazione del db ed è così importante che è sempre multiplexed, cioè esiste in più copie identiche in locazioni diverse, in modo che se una copia viene persa ce n'è sempre almeno un'altra.

1.5.2 I Redo Log file

I Redo Log file contengono tutte le operazioni eseguite sul DB. Sono essenziali per riportare il database ad uno stato consistente in caso di crash del sistema. Come detto, infatti, i blocchi modificati vengono conservati in RAM fino al prossimo checkpoint, ma cosa succede se il sistema si spegne improvvisamente prima del checkpoint? Ovviamente tutte le modifiche apportate ai dati dall'ultimo checkpoint in avanti sarebbero perse, perché non sono state ancora scritte sui datafile. In questo caso alla riaccensione del sistema Oracle fa partire automaticamente il processo di *recovery* che si occupa, leggendo i Redo Log file, di rieseguire tutte le operazioni che erano

state eseguite dopo l’ultimo checkpoint correttamente registrato nel control file ed erano andate perse.

I Redo Log file vengono definiti in fase di creazione del database. Il DBA può decidere quanti file creare, se e quante volte duplicarli su locazioni differenti (anche questi possono essere multiplexed come i control file). Siccome nei Redo Log file viene scritta una enorme mole di informazioni, essi in genere si esauriscono in tempi abbastanza ridotti e vengono riempiti in modalità ciclica. Ipotizziamo che siano tre. Finito il primo file Oracle passa a scrivere sul secondo, finito questo passa sul terzo e, alla fine del terzo, ricomincia a scrivere sul primo. Il cambio di log prende il nome di *log switch*. Ad ogni log switch viene eseguito sempre un checkpoint.

Se si desidera creare una copia dei Redo Log file prima che vangano sovrascritti dal processo ciclico si deve attivare la funzionalità Archivelog. Attivando questa modalità un apposito processo di base (ARCn, Archiver) si occupa di salvare una copia del Redo Log file prima che sia sovrascritta.

2 Installazione

Le attività di installazione e configurazione di Oracle sono normalmente a carico di un amministratore di database. Una precisa analisi di queste importantissime attività va, dunque, sicuramente al di là dello scopo di questo manuale. È però evidente che normalmente chi vuole cominciare a conoscere un prodotto software deve averne a disposizione una versione di prova, in modo da poter affiancare allo studio teorico le fondamentali esercitazioni. Per questo motivo, nelle prime edizioni di questo manuale, era stata fornita una guida passo-passo all'installazione di Oracle. Tale guida, però, era troppo sensibile ai cambiamenti frequenti nell'organizzazione del sito web Oracle e nel tool d'installazione del db. Assumerò, dunque, che il lettore abbia a disposizione una installazione di Oracle funzionante, rimandando al sito ufficiale della Oracle (http://www.oracle.com/) ed alle tante fonti disponibili sul web per i dettagli su come scaricare ed installare il software.

L'unica accortezza che bisogna avere durante l'installazione, in Oracle 12c, è relativa alla scelta di attivare o meno l'architettura multitenant, che sarà ampiamente discussa nel capitolo 9.

Tale architettura si abilita con un apposito check-box (riquadro rosso nell'immagine che segue) durante l'installazione guidata. Se il controllo viene attivato sarà creanto un database contenitore (CDB) ed, eventualmente, un pluggable database (PDB) al suo interno. Se non attiviamo questa opzione otterremo un db classico che, in Oracle 12c, è definito non-CDB.

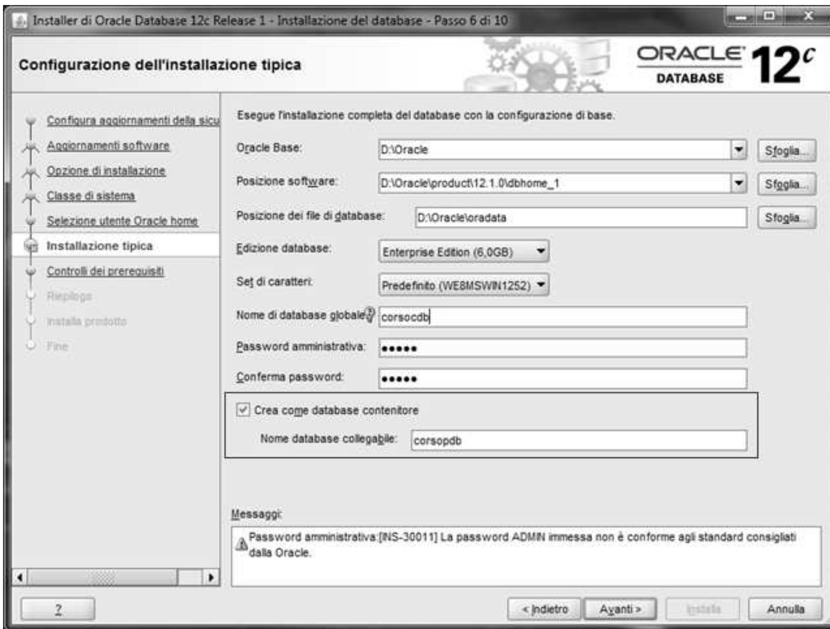


Figura 2-1 Multitenant Architecture: scelta tra CDB o non-CDB

Tutti gli esempi che seguono sono relativi ad un'architettura Oracle classica (versioni di Oracle fino alla 11g oppure un non-CDB in Oracle 12c), ad eccezione degli esempi mostrati nel capitolo 9, che assumono di avere a disposizione un CDB in Oracle 12c.

3 Operazioni preliminari

3.1 Avviare ed arrestare Oracle

Come descritto nel primo capitolo dedicato all'architettura, Oracle si divide sostanzialmente in due componenti distinte: l'istanza ed il database. In genere avviare Oracle vuol dire avviare entrambe le componenti, ma ci possono essere dei casi in cui, per esigenze del DBA, sia necessario avviare l'istanza e tenere offline il database. Per questo motivo i processi di avvio (startup) ed arresto (shutdown) di Oracle prevedono la possibilità di elaborazione graduale, che vedremo più dettagliatamente nei due paragrafi che seguono. L'avvio e l'arresto di Oracle si possono eseguire in vari modi. Si possono utilizzare alcuni tool di Oracle disponibili su qualunque sistema operativo come SQL*Plus, ampiamente trattato in questo manuale, oppure Enterprise Manager (la console di amministrazione del DB che, invece, non descriveremo). In ambiente Windows all'installazione di Oracle vengono creati alcuni servizi che possono essere gestiti mediante la console che si trova in ***Pannello di Controllo → Strumenti di Amministrazione → Servizi***.

Il servizio OracleService<SID>, dove <SID> è il nome di database scelto in fase di installazione, consente di avviare ed arrestare istanza e database.

Sempre in ambiente Windows Oracle fornisce oradim, un tool eseguibile da prompt dei comandi che consente, tra l'altro, anche di avviare ed arrestare istanza e database.

Nel seguito descriveremo la procedura di startup e shutdown da SQL*Plus, visto che questa è valida per tutti i sistemi operativi. Prima di eseguire un comando di Sql*Plus, però, bisogna lanciare il tool. È molto semplice: in Windows si può utilizzare l'apposita voce presente nel menù Avvio di Windows oppure, in qualunque sistema, da linea di comando basta digitare

```
sqlplus / as sysdba
```

Ovviamente perché questo comando abbia successo è necessario che l'utente di sistema operativo abbia i privilegi adatti. Utilizzando la stessa utenza che si è utilizzata per installare il DB di sicuro non si sbaglia.

3.1.1 Startup

La procedura di startup consiste nell'avviare un database spento e si concretizza nel susseguirsi dei seguenti quattro stati:

- SHUTDOWN, l'istanza è ferma ed il database non è disponibile
- NOMOUNT, l'istanza è attiva ed il database non è disponibile
- MOUNT, l'istanza è attiva ed il db è disponibile solo per operazioni di amministrazione
- OPEN, l'istanza è attiva ed il database è disponibile per tutte le operazioni.

Il comando che consente di passare dallo stato SHUTDOWN direttamente ad uno degli altri tre stati è STARTUP, specificando la parola chiave NOMOUNT o MOUNT se si desidera andare in uno di questi due stati intermedi. Il comando STARTUP da solo porta direttamente allo stato OPEN. Il comando che invece consente di passare dallo stato NOMOUNT a MOUNT, da MOUNT ad OPEN o direttamente da NOMOUNT ad OPEN è ALTER DATABASE seguito dallo stato desiderato. Ad esempio ALTER DATABASE OPEN porta il DB in stato OPEN sia partendo dallo stato NOMOUNT che partendo dallo stato MOUNT.

3.1.2 Shutdown

Il processo di shutdown esegue gli step inversi di quello di startup. Il comando SQL*Plus da utilizzare è SHUTDOWN che rende indisponibile il db ed arresta l'istanza a partire da qualunque stato.

Esistono quattro diverse modalità di SHUTDOWN:

- SHUTDOWN NORMAL, è la modalità di default ed equivale al semplice SHUTDOWN. Il database non viene spento immediatamente, non vengono accettate nuove connessioni ma il db resta in piedi fino a quando tutti gli utenti già collegati si siano volontariamente disconnessi. Lo shutdown effettivo può quindi avvenire anche molto tempo dopo l'esecuzione del comando.
- SHUTDOWN TRANSACTIONAL, Il database non viene spento immediatamente, non vengono accettate nuove connessioni ma il db resta in piedi fino a quando tutti gli utenti già collegati decidano volontariamente di salvare o annullare le modifiche apportate ai dati. Una volta che un utente salva o annulla le sue modifiche viene disconnesso automaticamente e non può più riconnettersi. Come prima, lo shutdown può avvenire anche molto tempo dopo l'esecuzione del comando.
- SHUTDOWN IMMEDIATE , Oracle esegue un checkpoint e butta fuori tutti gli utenti collegati. Le modifiche degli utenti non salvate andranno perse.

- SHUTDOWN ABORT , Equivale a staccare la spina del server su cui Oracle è installato, non c'è tempo per fare nulla, alla ripartenza di Oracle sarà sicuramente eseguito un recovery.

Un esempio di shutdown e startup del database da Sql*Plus è presentato in Figura 3-1.

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Versione 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Massimo>d:

D:\>cd oracle

D:\oracle>cd p*

D:\oracle\product>cd 1*

D:\oracle\product\11.2.0>cd d*

D:\oracle\product\11.2.0\dbhome_1>cd b*

D:\oracle\product\11.2.0\dbhome_1\BIN>sqlplus / as sysdba

SQL*Plus: Release 11.2.0.1.0 Production on Mer Ago 11 22:09:55 2010

Copyright (c) 1982, 2010, Oracle.  All rights reserved.


Connesso a:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> --ADESSO IL DB E L'ISTANZA SONO ENTRAMBI ATTIVI.
SQL> shutdown
Database chiuso.
NOMOUNT del database eseguito.
Istanza ORACLE chiusa.
SQL> --ADESSO E' TUTTO SPENTO, RIACCENDIAMO UN PO' PER VOLTA...
SQL> startup nomount
Istanza ORACLE avviata.

Total System Global Area  426852352 bytes
Fixed Size                  1375060 bytes
Variable Size             264242348 bytes
Database Buffers          155189248 bytes
Redo Buffers                6045696 bytes
SQL> --ORA C'E' SOLO L'ISTANZA
SQL> alter database mount;

Modificato database.

SQL> --ADESSO C'E' ANCHE IL DB MA E' CHIUSO
SQL> alter database open;

Modificato database.

SQL> --ADESSO E' TUTTO ACCESO.
SQL> exit
Disconnesso da Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Produ
ction
With the Partitioning, OLAP, Data Mining and Real Application Testing options

D:\oracle\product\11.2.0\dbhome_1\BIN>
```

Figura 3-1 Shutdown e startup da Sql*Plus

3.2 Connettersi al database

Ora che il database è installato ed avviato possiamo collegarci per cominciare a creare le nostre tabelle e finalmente giocare con i dati. Prima, però, dobbiamo capire come sono suddivisi i dati dei diversi utenti che si possono collegare al database e come avviene praticamente la connessione.

3.2.1 Utenti e schemi

Prima di tutto introduciamo due concetti, l'utente e lo schema. Lo schema è un insieme di oggetti, tabelle ed oggetti di altro tipo che incontreremo più avanti, che sono tutti di proprietà dello stesso utente. Lo schema e l'utente che lo possiede hanno lo stesso nome e, anche se

concettualmente sono due cose diverse, ai fini pratici sono sostanzialmente la stessa cosa.

Quando si installa Oracle vengono creati automaticamente vari utenti di sistema con i relativi schemi. Due di questi ci interessano particolarmente: SYS e SYSTEM. Le password di questi due utenti sono state scelte in fase di installazione.

L'utente SYS è il superutente, proprietario di tutti gli oggetti di sistema, dovrebbe essere utilizzato solo per particolarissime operazioni di amministrazione.

L'utente SYSTEM è comunque un utente speciale, ha ampia delega da parte di SYS ad eseguire la maggior parte delle attività di amministrazione e dovrebbe essere utilizzato solo per queste. Una delle operazione che tipicamente vengono fatte utilizzando SYSTEM è la creazione di altri utenti/schemi.

3.2.2 Connettersi dal server dove è installato il db

Se avete la possibilità di accedere direttamente al server dove è in stallato Oracle la connessione è davvero molto semplice. Basta fornire il nome dell'utente e la password. Ad esempio per lanciare Sql*Plus connettendosi con l'utente SYSTEM, ipotizzando che la password di SYSTEM sia "systempwd", basta fare da prompt dei comandi

```
sqlplus system/systempwd
```

Oppure, volendo tenere nascosta la password, semplicemente

```
sqlplus
```

e poi specificare nome utente e password quando vengono richiesti.

3.2.3 Connettersi da un client remoto

Un po' più complessa è la connessione ad Oracle quando vogliamo collegarci da un altro computer che nel seguito chiameremo *client*. Chiameremo invece *server* il computer su cui è installato Oracle. In questo caso bisogna fornire, oltre ad username e password, le seguenti informazioni:

- Nome o indirizzo IP del server.
- Nome del database scelto in fase di installazione.
- Protocollo di rete utilizzato per la connessione tra il client ed il server.
- Porta su cui è in ascolto il listener sul server.

Prima di procedere spiegando come si forniscono queste informazioni descriviamo velocemente come avviene tecnicamente la connessione.

Un programma che gira sul client e richiede la connessione ad un database Oracle è detto *servizio client*. Il servizio client deve comunicare col

server, quindi prima di tutto ci deve essere una rete tra client e server. Di norma client e server utilizzano come protocollo di comunicazione il TCP/IP e il client deve conoscere il nome del server oppure il suo indirizzo IP. Una volta che il client ha note queste due informazioni può aprire una connessione verso il server. A questo punto è necessario capire cosa succede sul server. Per poter accettare connessioni dall'esterno un server deve avere attivo un programma che sia in grado di ricevere le connessioni TCP/IP e smistarle verso Oracle. Questo programma si chiama *listener*. Siccome sul server ci potrebbero essere più programmi in attesa di connessioni TCP/IP, programmi che servono ad Oracle oppure ad altro, è necessario che il listener sia in ascolto su una porta particolare e che il client quando si connette conosca questa porta. Per default la porta su cui i listener Oracle si mettono in ascolto è la 1521, ma questo parametro può essere modificato a piacere dall'amministratore del database.

Se avete installato Oracle su Windows è stato creato un servizio che consente di avviare ed arrestare il listener.

In qualunque sistema operativo il listener si controlla con il tool **lsnrctl** che può essere utilizzato da prompt dei comandi.

Per avviare il listener useremo il comando

```
lsnrctl start
```

per stopparlo, invece, useremo il comando

```
lsnrctl stop
```

Torniamo alla descrizione del processo di connessione. Il client ha contattato il server usando il protocollo TCP/IP la porta del listener. Il listener ha risposto ed ha preso in carico la richiesta del servizio client, ora deve lavorarla. Ci sono due possibili modalità.

Nella prima modalità, detta *dedicated server*, per ogni processo client che si presenta il listener attiva un processo server dedicato, ovvero un processo di base che dovrà esaudire le richieste di quel processo client e solo di quello. Se il client dopo la connessione resta inattivo il processo server girerà a vuoto senza fare nulla.

Nella seconda modalità, detta *shared server*, il listener passa la connessione ad uno o più processi di base chiamati *dispatcher*. Il dispatcher che riceve la connessione ha a sua volta a disposizione un numero limitato di processi server, mette la richiesta in una lista ed il primo processo server libero se la prende e la lavora. Il giro è leggermente più complesso, ma consente di avere un numero molto più basso di processi server che sono sempre impegnati, anziché tanti processi server molti dei quali girano a vuoto. La scelta tra shared e dedicated server dipende da vari fattori ed è di competenza dell'amministratore di database. Non ha nessun impatto sul client.

Torniamo ai parametri di connessione. Alcuni programmi client di fornitori terzi che si collegano ad Oracle vi chiederanno esplicitamente di indicare i quattro parametri. Tutti i programmi client forniti da Oracle e molti anche di terze parti, invece, richiedono che sul client venga fatta una speciale configurazione. In pratica in un file di sistema che si trova sul client bisogna indicare, per ogni database Oracle a cui ci si intende collegare, un nome mnemonico per questa connessione associato ai quattro parametri di cui abbiamo parlato. Il nome mnemonico è detto *connect string*, quando ci si collega ad Oracle, dunque, sarà sufficiente fornire utente, password e connect string. Il programma utilizzato per collegarsi sarà in grado di ricavare dalla connect string i quattro parametri di base ed effettuare la connessione. Il file di configurazione in cui si definiscono le connect string si chiama *tnsnames.ora* e si trova nella cartella Admin sotto la directory Network nella Oracle Home in cui è stato installato il client Oracle.

Il tnsnames.ora non andrebbe mai modificato manualmente, anche se è prassi comune farlo. Per modificare il tnsnames.ora Oracle mette a disposizione un programma che si chiama Net Manager e si trova nel menù avvio.

Ovviamente il menù può cambiare per versioni diverse di Oracle, è infatti in generale possibile collegarsi ad un server Oracle di una certa versione con un client di versione differente. Vediamo passo per passo la configurazione di una nuova connect string da Net Manager.

Passo 1 – Aggiungere un nuovo servizio remoto posizionandosi su “Denominazione dei servizi” e facendo clic sul più verde a sinistra

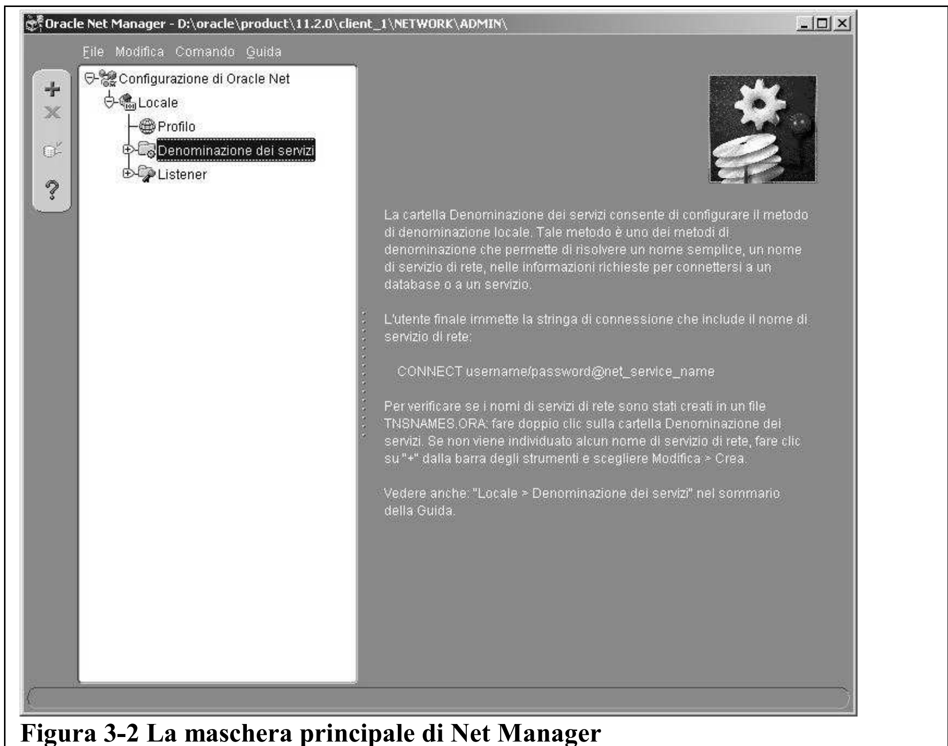


Figura 3-2 La maschera principale di Net Manager

Passo 2 – Inserire la nuova connect string che si intende creare e fare clic su Avanti

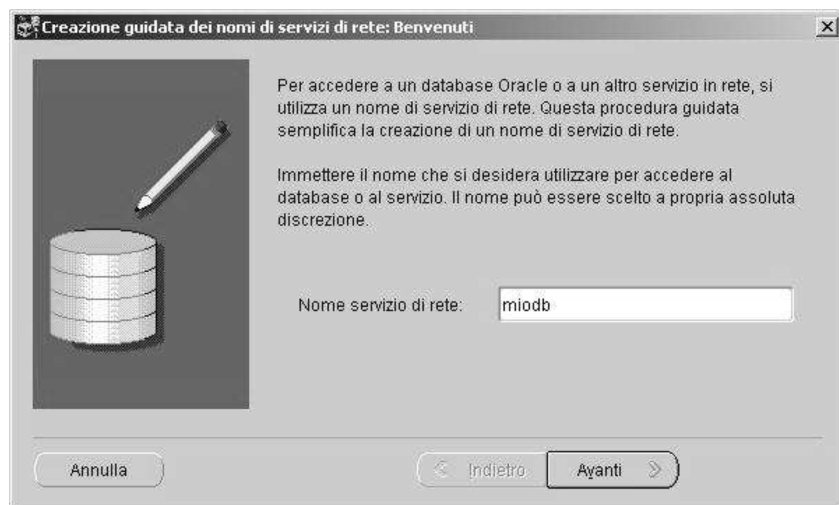


Figura 3-3 Definizione del nome della connect string

Passo 3 – Scegliere il protocollo di rete (in genere TCP/IP) e fare clic su Avanti

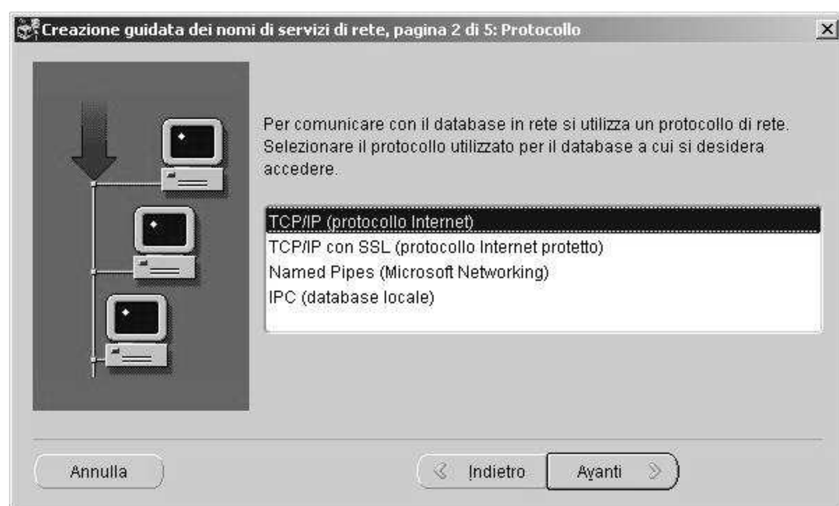


Figura 3-4 Scelta del protocollo di rete

Passo 4 – Indicare il nome (oppure l'indirizzo IP) del server e la porta su cui il listener è in ascolto e fare clic su Avanti.

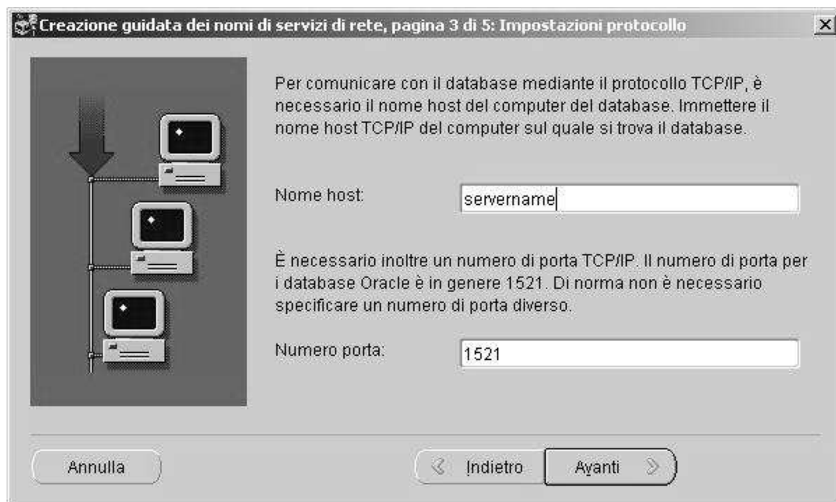


Figura 3-5 Scelta del server e della porta

Passo 5 – Indicare il nome del database remoto e fare clic su Avanti

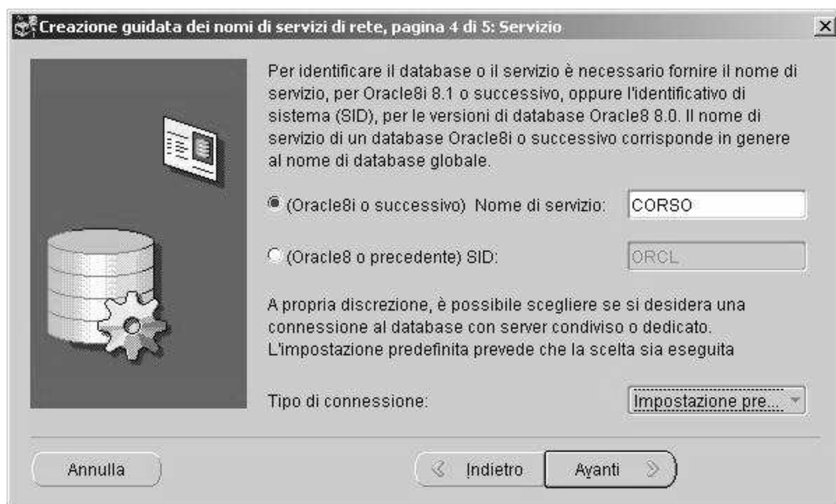


Figura 3-6 Scelta del nome del database remoto

Passo 6 – Cliccare su Test per effettuare il test della connessione

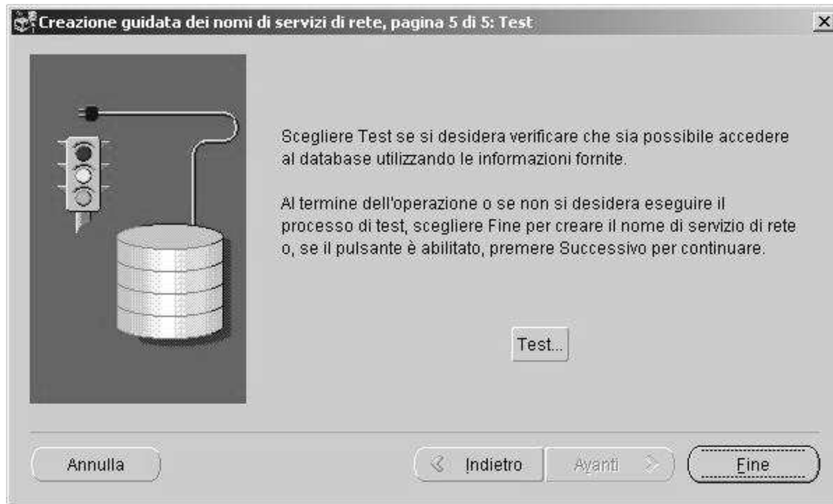


Figura 3-7 Esecuzione del test

Passo 7 – Il test va in errore perché cerca di utilizzare un utente di default (scott) che non esiste oppure ha una password differente da quella che è stata utilizzata, in tal caso cliccare su Cambia login

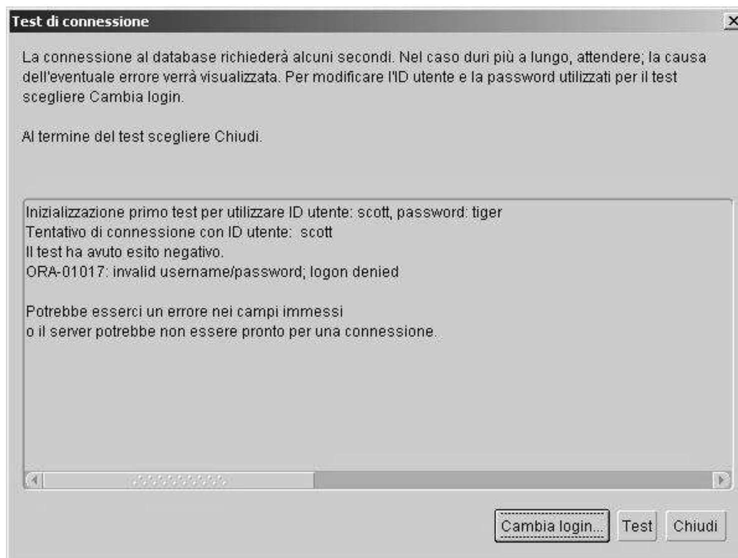


Figura 3-8 Errore nel test a causa di utente inesistente

Passo 8 – Indicare le credenziali corrette (in questo esempio esiste un utente che si chiama "corso", si può anche usare SYSTEM o un qualunque altro utente) e fare clic su OK e poi su Test

Figura 3-9 Indicazione delle credenziali corrette

Passo 9 – Il test è andato bene, la configurazione è OK

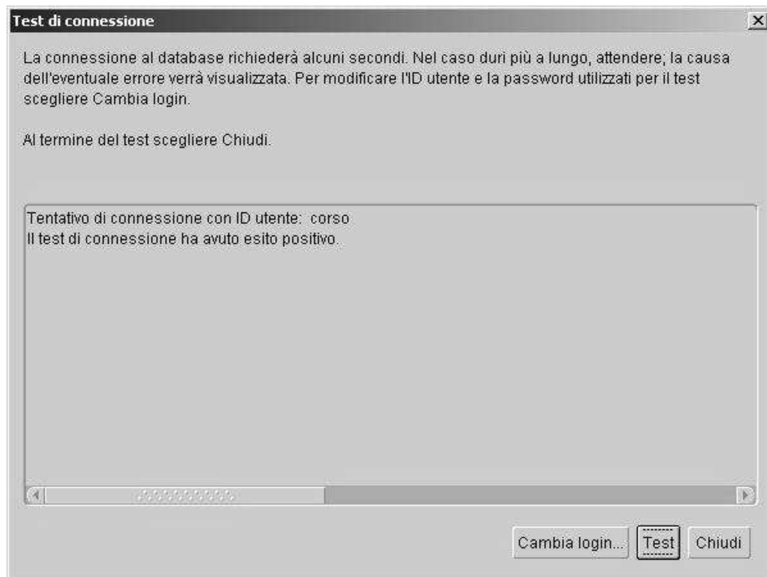


Figura 3-10 Test con esito positivo

Passo 10 – Chiudere Net Manager e salvare le modifiche

Figura 3-11 Maschera di salvataggio delle modifiche

Se ora apriamo il file tnsnames.ora all'interno troveremo, tra le altre configurazioni:

```
miodb =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST = servername)(PORT = 1521))
    )
    (CONNECT_DATA =
      (SERVICE_NAME = CORSO)
    )
  )
```

Da questo momento in poi è possibile connettersi da questo client a quel server Oracle specificando solo l'utente, la password e la stringa di connessione miodb. Per lanciare sqlplus collegandosi con l'utente CORSO, ad esempio, potremo utilizzare il seguente comando:

```
sqlplus corso/corsopwd@miodb
```

Oppure, volendo tenere nascosta la password, semplicemente

```
sqlplus
```

e poi specificare corso@miodb come nome utente e la password quando vengono richiesti. In Figura 3-12 un esempio di connessione con Sql*Plus ad un database remoto.

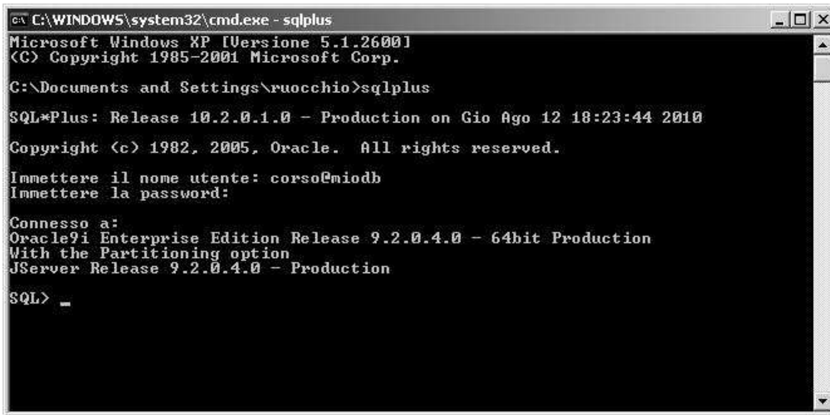


Figura 3-12 Connessione con Sql*Plus da client remoto.

3.2.4 Creare un nuovo utente

Per creare un nuovo utente basta connettersi a Sql*Plus con l'utente SYSTEM in una delle due modalità (locale o da client remoto) appena illustrate ed utilizzare le seguenti istruzioni

```
Create user corso identified by corsopwd;
Grant connect, resource to corso;
```

La prima istruzione crea l'utente ed il relativo schema e gli assegna la password "corsopwd". Attenzione: a partire da Oracle 11g la password è per default case sensitive. Questo significa che per collegarsi al db utilizzando l'utente corso bisognerà specificare la password in caratteri minuscoli. Tutto il resto delle due istruzioni, invece, non è case sensitive, avreste potuto scriverlo in maiuscolo o minuscolo indifferentemente, non sarebbe cambiato nulla.

La seconda istruzione assegna all'utente corso i privilegi minimi che gli servono per connettersi al db e creare oggetti nel proprio schema.

4 SQL*Plus e SQL Developer

Oracle fornisce alcuni strumenti software che consentono di collegarsi al database, locale o remoto che sia, e realizzare tutte le attività di amministrazione e gestione dei dati.

In questo capitolo saranno introdotti SQL*Plus e SQL Developer, due strumenti profondamente diversi tra loro nell'interfaccia utente che consentono di interagire con il database, definire, modificare ed eliminare oggetti in esso contenuti, gestirne i dati ed eseguire, più genericamente, una qualunque istruzione SQL.

L'introduzione di questi strumenti in una fase del corso in cui ancora non si è cominciato a parlare di SQL e PL/SQL comporta necessariamente una trattazione non approfondita. L'obbiettivo di queste pagine è dunque comprendere la differenza tra i due strumenti, essere in grado di avviarli e procedere con le configurazioni di base, riuscire ad eseguire una semplicissima istruzione SQL di esempio.

Una familiarità sempre maggiore con gli strumenti e con le funzionalità che offrono verrà naturalmente nel corso dei capitoli successivi.

*4.1 SQL*Plus*

SQL*Plus è un tool a linea di comando (non grafico) presente in Oracle fin dalla versione 5. È stato per anni l'unico strumento fornito con il DB sia per l'amministrazione che per la gestione dei dati. Dalla versione 7 alla versione 10g Release 2 del database, e solo in ambiente Windows, Oracle ha fornito anche una versione graficamente più evoluta di SQL*Plus che è stata eliminata nella versione 11g. Da questa scelta traspare l'orientamento di Oracle a continuare il supporto di SQL*Plus solo al fine di consentire agli utenti la realizzazione di alcune attività molto specifiche, in particolare quelle che richiedono una forte automatizzazione, e spingere il grosso degli utenti all'utilizzo del nuovo strumento grafico, SQL Developer, fornito a partire dalla Release 2 di Oracle 10g.

4.1.1 Perché ancora SQL*Plus

Nonostante questa tendenza alla riduzione dell'ambito di SQL*Plus sia ormai evidente, è mia opinione che un corso base sul database Oracle non possa prescindere dall'utilizzo di questo strumento. Se è vero, infatti, che SQL Developer ha copertura funzionale maggiore di SQL*Plus, è anche vero che esistono in giro molte installazioni di Oracle ferme alla versione 9i, se non addirittura alla 8 o 8i. Su tali installazioni non c'è nessuna valida alternativa all'utilizzo di SQL*Plus (a meno che, ovviamente, non si intenda di utilizzare prodotti di terze parti).

Dal punto di vista strettamente didattico, poi, l'utilizzo dello strumento a linea di comando obbliga lo studente ad affrontare le difficoltà dovute alla sintassi dei comandi, laddove lo strumento grafico molto spesso agevola l'utilizzatore con utility e wizard che eliminano tali difficoltà. In fase di apprendimento la necessità di dover scrivere, e spesso riscrivere più volte, un'istruzione nella sintassi corretta aiuta a comprendere a fondo il senso di ciò che si sta facendo; la disponibilità di utility troppo semplici, invece, tende a far trascurare i dettagli. Lo studente che impara a creare una tabella usando il comando CREATE TABLE, di sicuro non avrà difficoltà in futuro a portare a termine la stessa attività utilizzando un qualunque strumento grafico. Il viceversa ovviamente non vale.

4.1.2 Personalizzare SQL*Plus (Windows)

Bisogna ammettere che dal punto di vista estetico e dell'usabilità SQL*Plus lascia decisamente a desiderare rispetto agli strumenti grafici con cui siamo normalmente abituati a lavorare.

Per ridurre le difficoltà possiamo configurare lo strumento come descritto di seguito. Innanzi tutto conviene sfruttare le proprietà del prompt dei comandi di Windows. Una volta lanciato SQL*Plus basta fare clic col tasto destro del mouse sul titolo della finestra e scegliere la voce "Proprietà" dal menù contestuale, come in Figura 4-1.

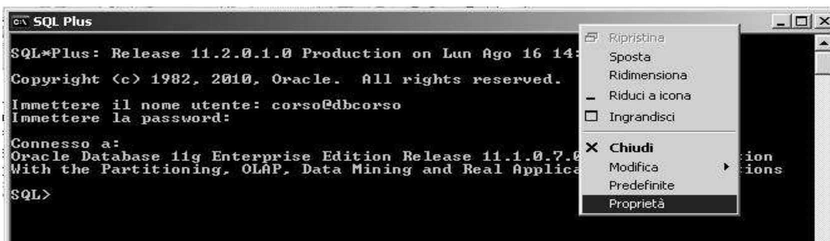


Figura 4-1 Menù contestuale.

Comparirà la finestra mostrata in Figura 4-2. La prima opzione da selezionare è "Modalità Modifica Rapida" che consente di agevolare di molto il "Copia-Incolla".

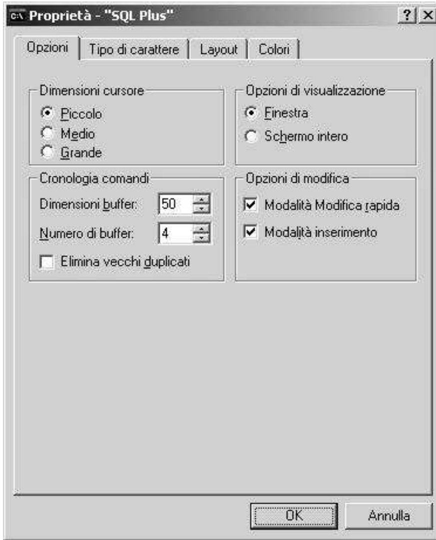


Figura 4-2 Proprietà di SQL*Plus.

Si può poi passare ad impostare a piacimento il tipo di carattere ed i colori. Quello che assolutamente conviene fare è aumentare la dimensione delle righe e delle colonne visualizzate, cliccando su “Layout” basta impostare i campi come in Figura 4-3.

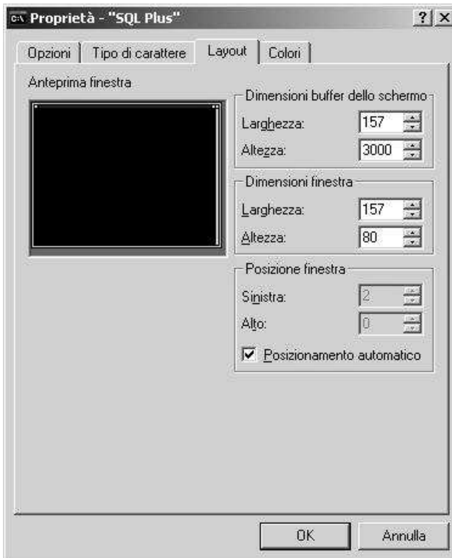


Figura 4-3 Proprietà del Layout.

I parametri indicati sono adatti ad uno schermo con risoluzione 1280x1024, con risoluzioni diverse si può fare qualche tentativo per trovare il giusto numero di righe e colonne della finestra.

Quando si chiude la finestra delle proprietà bisogna scegliere la voce “Modifica il collegamento che ha aperto questa finestra”, come mostrato in

Figura 4-4, in modo che le impostazioni scelte saranno utilizzate ogni qual volta si lancia SQL*Plus dal menù.

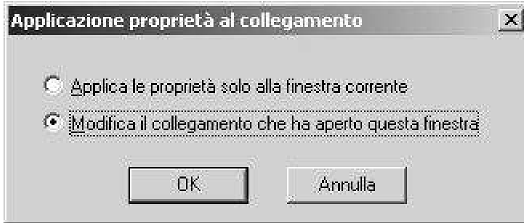


Figura 4-4 Modifica del collegamento.

Se ci si collega frequentemente allo stesso database con le medesime credenziali, si può creare un collegamento sul desktop che lanci SQL*Plus ed effettui automaticamente la connessione al DB. Facendo "Tasto destro del mouse – Copia" sull'icona di SQL*Plus del menù e poi "Tasto destro del mouse – Incolla" sul desktop otteniamo un collegamento a SQL*Plus che eredita tutte le configurazioni appena impostate. Possiamo modificare il collegamento in modo che si colleghi automaticamente. Facendo "Tasto destro del mouse – Proprietà" sul collegamento appena creato si apre la finestra presentata in Figura 4-5.

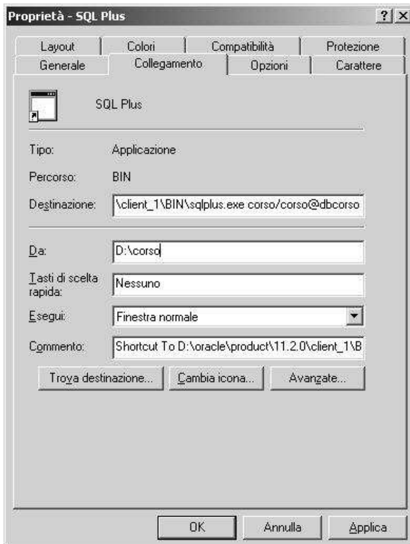


Figura 4-5 Modifica del collegamento.

Nel campo "Destinazione" dopo il nome del programma aggiungiamo le informazioni relative a nome utente, password e stringa di connessione. Ovviamente se si intende tenere riservata la password questa può essere omessa e sarà richiesta da SQL*Plus alla partenza. Conviene anche modificare il campo "Da". Tutti i file letti e scritti da SQL*Plus saranno per default cercati ed inseriti nella cartella indicata, nell'esempio il campo è stato valorizzato con la cartella D:\Corso. Ovviamente sarà sempre possibile da

SQL*Plus leggere o scrivere file che si trovano in cartelle differenti, sarà solo necessario indicare esplicitamente il path oltre al nome dei file.

4.1.3 Parametri e comandi di SQL*Plus

Le proprietà che abbiamo impostato nel paragrafo precedente non sono specifiche di SQL*Plus ma appartengono al prompt dei comandi di Windows. SQL*Plus ha un insieme di parametri propri che ne consentono la personalizzazione, ne vedremo alcuni in questo paragrafo.

Per prima cosa, come mostrato in Figura 4-6, eseguiamo il comando

```
Show all
```

che mostra tutti i parametri disponibili ed i valori attualmente impostati.

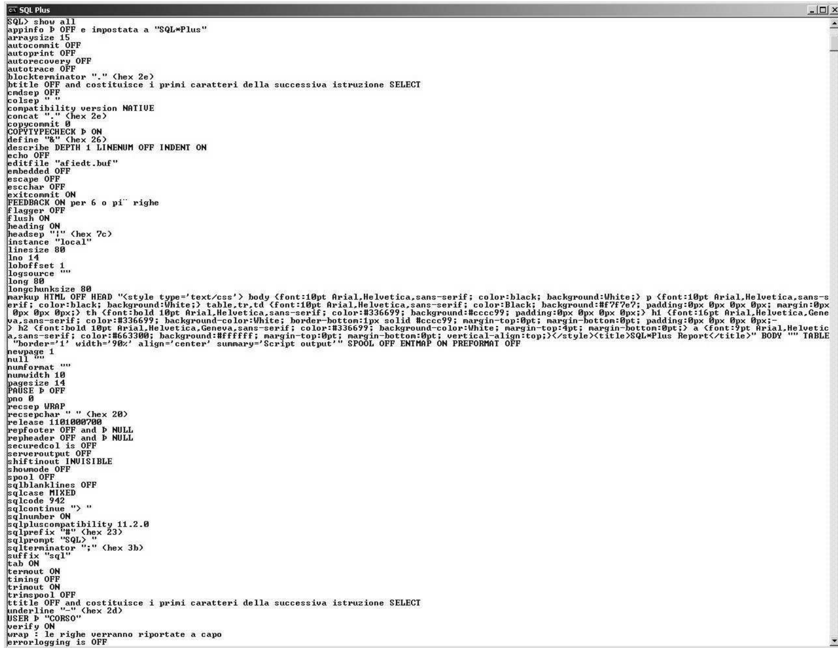


Figura 4-6 Tutti i parametri di SQL*Plus.

Se invece avessimo voluto visualizzare il valore di uno specifico parametro avremmo dovuto lanciare il comando.

```
Show <nomeparametro>
```

Ad esempio

```
Show linesize
```

mostra il valore del parametro linesize, per default 80, che rappresenta il numero di caratteri che formano ogni linea orizzontale in SQL*Plus. Siccome abbiamo allargato la finestra per contenere 157 caratteri in orizzontale (Figura 4-3) non ha senso tenere la linesize ad 80 caratteri, i restanti 77 resterebbero sempre inutilizzati. Impostiamo dunque la larghezza della linea di SQL*Plus a 156 caratteri (per una visualizzazione ottimale

conviene tenere la linea di SQL*Plus un carattere più piccola della linea visualizzabile nella finestra):

```
set linesize 156
```

e poi ripetiamo il comando show per verificare se tutto è ok, Figura 4-7.

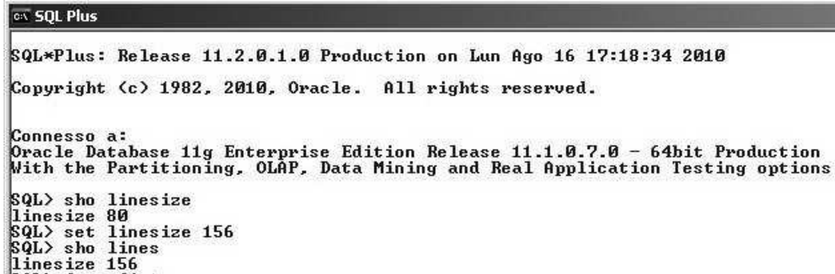


Figura 4-7 Impostazione della linesize.

Si noti che il comando SHOW ha il diminutivo SHO che funziona esattamente allo stesso modo. Anche i parametri hanno spesso dei diminutivi, ad esempio il parametro linesize ha il diminutivo lines. Si noti anche che i comandi di SQL*Plus possono essere scritti indifferentemente in minuscolo o maiuscolo senza alterarne il risultato.

Detto dei comandi SET e SHOW, usati per vedere e modificare i valori dei parametri, un altro comando di SQL*Plus molto importante è DESCRIBE (diminutivo DESC). Questo comando consente di visualizzare la struttura di un oggetto, ad esempio di una tabella. Non abbiamo ancora creato alcuna tabella, ma ce ne sono alcune di sistema accessibili da qualunque utente. Proviamo dunque ad eseguire il comando

```
Desc DICT
```

che mostra la struttura della tabella DICT. Questa tabella (in verità non si tratta proprio di una tabella ma di una vista, al momento non fa alcuna differenza), contiene l'indice del "dizionario dati". Il dizionario dati è un insieme di tabelle che contengono informazioni su tutti gli oggetti presenti nel database. L'output del comando DESC, come evidenziato in Figura 4-8, mostra un riepilogo delle colonne della tabella DICT. Per ogni colonna viene visualizzato il nome, se è sempre valorizzata oppure no ed il tipo di dati contenuti nella colonna. Non ci soffermiamo adesso su queste informazioni perché saranno ampiamente trattate nel capitolo dedicato al linguaggio SQL.

```
SQL Plus

SQL*Plus: Release 11.2.0.1.0 Production on Lun Ago 16 17:18:34 2010

Copyright (c) 1982, 2010, Oracle.  All rights reserved.


Connesso a:
Oracle Database 11g Enterprise Edition Release 11.1.0.7.0 - 64bit Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> sho linesize
linesize 80
SQL> set linesize 156
SQL> sho lines
linesize 156
SQL> desc dict
 Nome                                                                                 Nullo?   Tipo
 ------------------------------------------------------------------------------------ -------- ----------------
 TABLE_NAME                                                                                    VARCHAR2(30)
 COMMENTS                                                                                      VARCHAR2(4000)

SQL>
```

Figura 4-8 Il comando DESC.

Non esamineremo in questo capitolo uno per uno i parametri ed i comandi di SQL*Plus perché da un lato non trarremmo alcun beneficio immediato da una noiosa carrellata di comandi ed opzioni e, dall’altro, non avendo ancora introdotto l’SQL, sarebbe difficoltoso fare esempi significativi. I parametri ed i comandi che servono saranno illustrati nei prossimi capitoli, laddove ci sarà bisogno di utilizzarli.

4.1.4 Scrittura e modifica di un’istruzione SQL

È invece utile vedere subito come si scrive e si modifica un’istruzione SQL in SQL*Plus. Non bisogna fare confusione tra i comandi di SQL*Plus, che hanno senso solo all’interno di questo strumento ed hanno la funzione di modificarne il comportamento, ed i comandi SQL che invece servono per amministrare il database; gestirne gli oggetti; inserire, modificare, cancellare e visualizzare i dati in esso contenuti.

Una importante differenza tra i due insiemi di comandi è che i comandi SQL devono essere terminati con un carattere speciale che dica a SQL*Plus di eseguirli. Se si preme invio nel corso della digitazione di un comando SQL, SQL*Plus si disporrà sulla riga successiva in attesa del completamento dell’istruzione. In Figura 4-9 ad esempio è stato premuto il tasto Invio dopo avere scritto “select *”. SQL*Plus si è messo sulla seconda riga (indicando il numero di riga sulla sinistra) ed è in attesa che il comando venga completato. Per terminare il comando SQL e mandarlo in esecuzione bisogna utilizzare un punto e virgola “;” oppure uno slash “/”. Se si decide di utilizzare lo slash è necessario posizionarlo da solo sull’ultima riga dell’istruzione.

```
SQL Plus

SQL*Plus: Release 11.2.0.1.0 Production on Mar Ago 17 09:51:03 2010

Copyright (c) 1982, 2010, Oracle.  All rights reserved.


Connesso a:
Oracle Database 11g Enterprise Edition Release 11.1.0.7.0 - 64bit Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> select *
  2
```

Figura 4-9 Ritorno a capo durante la scrittura di un comando SQL.

È importante ribadire che sia il punto e virgola che lo slash non sono parte del comando SQL, ma sono semplicemente un modo per comunicare a SQL*Plus il fatto che il comando SQL è completo e deve essere eseguito. Un

esempio di utilizzo di questi caratteri speciali è presentato in Figura 4-10 dove è stata eseguita un'istruzione che estrae la data di sistema.

```
SQL Plus

SQL*Plus: Release 11.2.0.1.0 Production on Mar Ago 17 10:01:30 2010

Copyright (c) 1982, 2010, Oracle.  All rights reserved.


Connesso a:
Oracle Database 11g Enterprise Edition Release 11.1.0.7.0 - 64bit Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> select sysdate
  2  from dual;

SYSDATE
---------
17-AGO-10

SQL> select sysdate
  2  from dual
  3  /

SYSDATE
---------
17-AGO-10

SQL>
```

Figura 4-10 Esecuzione di un comando SQL.

I comandi di SQL*Plus come SET, SHOW e DESC, invece, non richiedono un carattere terminatore e vengono eseguiti alla pressione del tasto Invio.

Per modificare un'istruzione SQL appena eseguita è sufficiente digitare il comando "ed". Questo comando aprirà un file di testo in notepad (sotto linux sarà utilizzato il *vi*) contenente l'istruzione appena eseguita e denominato *afiedit.buf*. Fatte tutte le modifiche all'istruzione sarà sufficiente chiudere il file, salvando le modifiche, e digitare il comando "r" per eseguirla. Un esempio di modifica dell'istruzione è fornito in Figura 4-11 e Figura 4-12 . Nella prima, dopo la digitazione del comando "ed", si è aperto in notepad il file *afiedit.buf*. Una volta modificato, chiuso e salvato il file si ottiene la visualizzazione della nuova istruzione SQL, si veda l'immagine successiva, che viene poi eseguita mediante il comando "r".

```
SQL Plus

SQL*Plus: Release 11.2.0.1.0 Production on Mar Ago 17 10:01:30 2010

Copyright (c) 1982, 2010, Oracle.  All rights reserved.


Connesso a:
Oracle Database 11g Enterprise Edition Release 11.1.0.7.0 - 64bit Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> select sysdate
  2  from dual;

SYSDATE
---------
17-AGO-10

SQL> select sysdate
  2  from dual
  3  /

SYSDATE
---------
17-AGO-10

SQL> ed
Registrato file afiedt.buf
```

```
afiedt.buf - Blocco note
File  Modifica  Formato  Visualizza  ?
select sysdate
from dual
/
```

Figura 4-11 Apertura di *afiedit.buf*

```
SQL Plus

SQL*Plus: Release 11.2.0.1.0 Production on Mar Ago 17 10:01:30 2010

Copyright (c) 1982, 2010, Oracle.  All rights reserved.


Connesso a:
Oracle Database 11g Enterprise Edition Release 11.1.0.7.0 - 64bit Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> select sysdate
  2  from dual;

SYSDATE
---------
17-AGO-10

SQL> select sysdate
  2  from dual
  3  /

SYSDATE
---------
17-AGO-10

SQL> ed
Registrato file afiedt.buf

  1  select sysdate+1
  2* from dual
SQL> r
  1  select sysdate+1
  2* from dual

SYSDATE+1
---------
18-AGO-10

SQL> _
```

Figura 4-12 Esecuzione del comando SQL modificato.

Questo modo di procedere può sembrare all'inizio un po' ostico, ma ci si fa velocemente l'abitudine.

Alcuni comandi speciali di SQL*plus ("i", "del", "c") consentono di modificare l'ultima istruzione SQL eseguita che è conservata in memoria. Mediante questi comandi è possibile inserire o cancellare una linea dell'istruzione oppure sostituire una stringa con un'altra. Un esempio è fornito

in Figura 4-13. Tutte le righe precedute da un doppio trattino "--" sono commenti che vengono ignorati da SQL*Plus e sono stati aggiunti per chiarire ciò che si stava facendo.

```
SQL Plus

SQL*Plus: Release 11.2.0.1.0 Production on Mar Ago 17 11:19:24 2010

Copyright (c) 1982, 2010, Oracle.  All rights reserved.

Connesso a:
Oracle Database 11g Enterprise Edition Release 11.1.0.7.0 - 64bit Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> select sysdate
  2  from dual;

SYSDATE
---------
17-AGO-10

SQL> --aggiungo una riga tra le due dell'istruzione precedente
SQL> --prima però mi posiziono sulla prima riga
SQL> 1
  1* select sysdate
SQL> i
  2i +1
  3i .
SQL> l
  1  select sysdate
  2  +1
  3* from dual
SQL> --digitando 1 mi sono posizionato sulla prima riga
SQL> -- digitando i ho richiesto l'inserimento di una riga sotto la prima
SQL> -- ho poi digitato +1 che è il contenuto della nuova linea
SQL> -- Il punto chiude l'attività di modifica
SQL> -- il comando "l" lista la nuova istruzione modificata
SQL> r
  1  select sysdate
  2  +1
  3* from dual

SYSDATE+1
---------
18-AGO-10

SQL> --con il comando "r" eseguo l'istruzione SQL visualizzata prima
SQL> del 2
SQL> --Il comando "del" seguito dal numero di linea cancella una linea
SQL> --quindi adesso l'istruzione SQL in memoria è
SQL> l
  1  select sysdate
  2* from dual
SQL> --se la eseguo ottengo
SQL> r
  1  select sysdate
  2* from dual

SYSDATE
---------
17-AGO-10

SQL> --se poi voglio scrivere "sysdate+1" al posto di "sysdate" posso fare
SQL> 1
  1* select sysdate
SQL> c.sysdate.sysdate+1
  1* select sysdate+1
SQL> l
  1  select sysdate+1
  2* from dual
SQL> r
  1  select sysdate+1
  2* from dual

SYSDATE+1
---------
18-AGO-10
```

Figura 4-13 I comandi "i", "del" e "c".

In alternativa all'uso dei comandi di modifica appena illustrati è possibile riscrivere l'istruzione SQL direttamente sulla linea di comando, magari utilizzando il copia-incolla.

4.1.5 esecuzione di uno script SQL

Le istruzioni SQL fin qui viste sono state scritte direttamente a linea di comando SQL*Plus. Per maggior comodità è possibile scrivere un file con tutti i comandi SQL che si vogliono eseguire, salvare lo script nel file system e poi utilizzare il comando SQL*Plus *start* oppure il suo sinonimo '@' per eseguirlo. Ipotizziamo, ad esempio, di avere scritto il comando

```
Select sysdate from dual;
```

nel file c:\test.sql. Per eseguire lo script sarà sufficiente scrivere da SQL*Plus

```
start c:\test.sql
```

oppure

```
@c:\test.sql
```

come mostrato nella figura seguente.

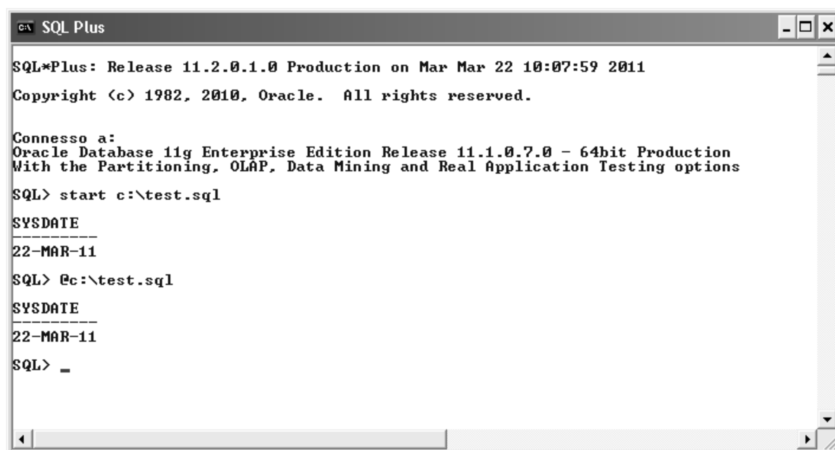


Figura 4-14 I comandi "start" e "@".

4.1.6 Cambio del prompt SQL

Nel prosieguo di questo manuale si vedrà spesso un prompt di SQL*Plus diverso dallo standard "SQL> ".

Per modificare il prompt di SQL*Plus si utilizza il comando

```
SET SQLPROMPT "<nuovo prompt>"
```

Come mostrato nella figura seguente

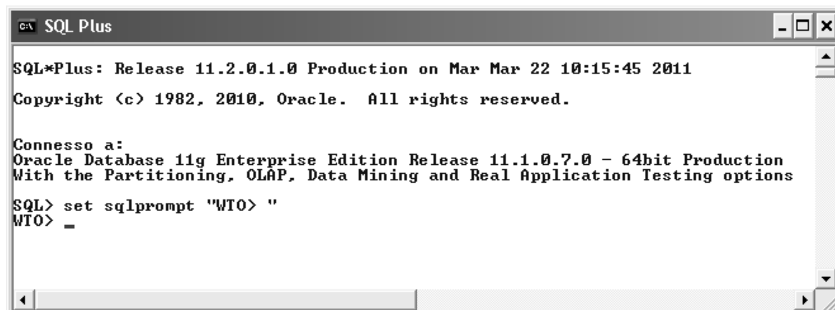


Figura 4-15 Il comando "set sqlprompt".

4.2 SQL Developer

SQL Developer è un tool grafico scritto in Java che, nelle intenzioni di Oracle, dovrebbe raccogliere l'eredità di SQL*Plus fornendo un'interfaccia utente più accattivante ed intuitiva.

4.2.1 Avvio di SQL Developer

SQL Developer può essere lanciato dal menù avvio. Alla prima partenza il prodotto chiederà all'utente quali tipi di file vuole associare a questa applicazione e poi visualizzerà la pagina principale come in Figura 4-16.

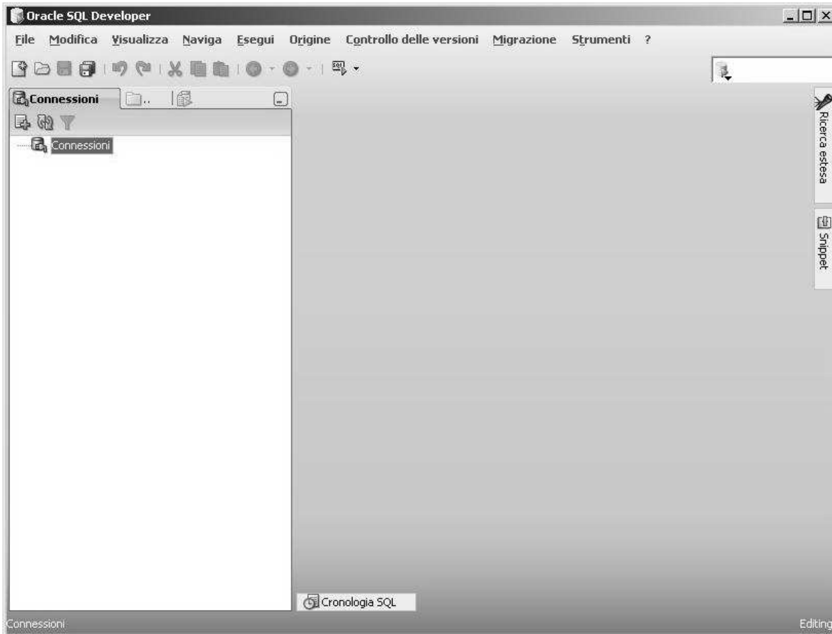


Figura 4-16 Pagina principale di SQL Developer.

Come in SQL*Plus, la prima attività da realizzare è la configurazione delle connessioni. Posizionandosi sulla scritta "Connessioni" nell'elenco di sinistra e facendo clic sul tasto "più" di colore verde immediatamente sopra, si aprirà la finestra di configurazione rappresentata in Figura 4-17.

In questa finestra bisogna indicare il nome che si intende assegnare alla connessione, il nome dell'utente e la relativa password. È possibile salvare la password con l'apposita opzione.

Se la connessione che si sta definendo è già presente sul client nel file *tnsnames.ora* sarà sufficiente scegliere TNS come tipo di connessione e selezionare da una lista denominata "Alias Rete" la connect string desiderata, come nell'esempio illustrato.

Se invece non è stata fatta una configurazione del tnsnames.ora si può lasciare "Tipo di connessione" al valore Basic ed indicare i soliti

parametri: nome o indirizzo del server, porta del listener, nome del servizio di database.

SQL Developer consente anche semplicemente di accedere ad un database Microsoft Access. Selezionando Access invece di Oracle sarà possibile indicare il nome del file mdb contenente il database.

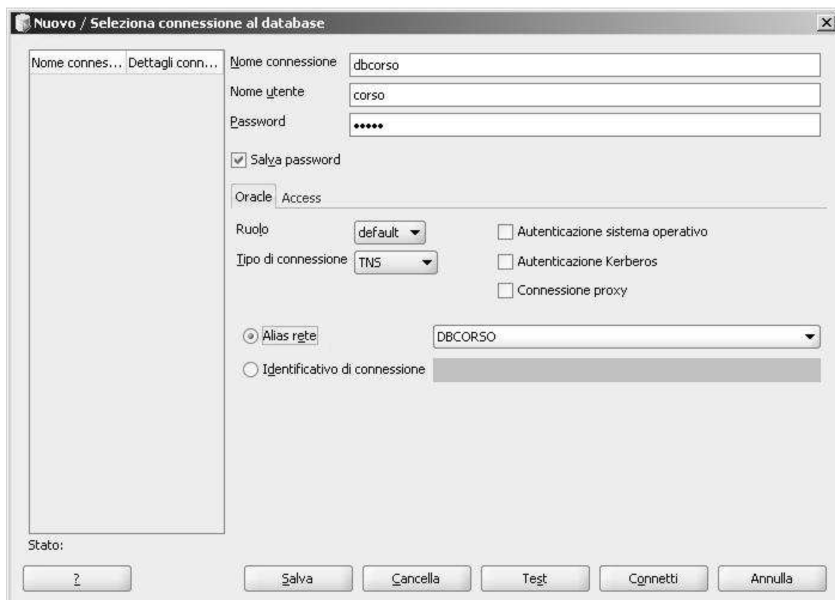


Figura 4-17 Configurazione delle connessioni.

Completata la configurazione della connessione si torna alla pagina principale facendo clic su “Salva” e poi su “Annulla”.

Nella lista di sinistra compare la nuova connessione e, cliccandoci sopra, SQL Developer si collega al db e si presenta come in Figura 4-18. A sinistra, sotto il nome della connessione, compare l’elenco di tutti i tipi di oggetto che possono essere contenuti nel database. Ispezionando i vari nodi si possono visualizzare tutti gli oggetti effettivamente contenuti. Al centro della pagina si apre una finestra utilizzabile per eseguire un’istruzione SQL. Basta scrivere l’istruzione, ovviamente senza punto e virgola, e fare clic sulla freccia verde che si trova subito sopra la finestra. Nell’esempio è stata eseguita la medesima lettura della data di sistema che era stata utilizzata anche in SQL*Plus.

Il risultato dell’istruzione è visualizzato in basso.

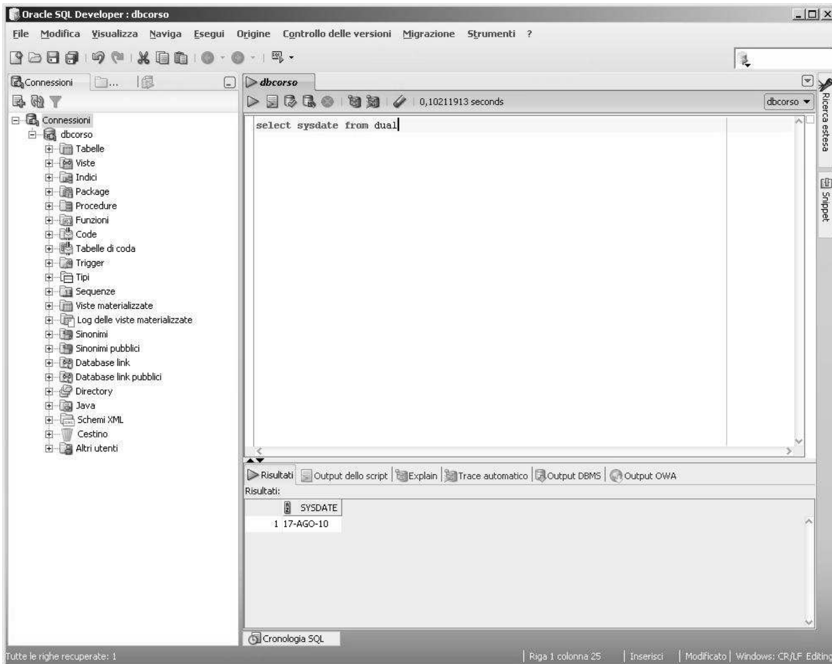


Figura 4-18 Esecuzione di un'istruzione SQL.

4.2.2 Parametri e comandi di SQL Developer

L'interfaccia utente di SQL Developer è molto intuitiva e non richiede particolari spiegazioni. Tutti i comandi sono eseguibili mediante il menu posto in alto ed i più importanti hanno un tastino dedicato. In questa fase del corso, non avendo ancora visto l'SQL, non è possibile entrare nel dettaglio delle singole funzionalità ma l'utilità della maggior parte dei comandi sarà chiarita nel corso dei prossimi capitoli.

5 Oggetti del DB

Fino a questo momento si è parlato genericamente di “oggetti contenuti nel database” menzionando, però, esplicitamente soltanto le tabelle. Un database è molto di più di un insieme di tabelle, per questo motivo in questo capitolo saranno introdotti i principali tipi di oggetto che possono essere creati in uno schema. Ciò consentirà al lettore di comprendere meglio le potenzialità del database Oracle e costituirà un indispensabile base di partenza per il prossimo capitolo, dove verrà introdotto il linguaggio SQL.

5.1 Il dizionario dati

Il dizionario dati è una collezione di tabelle che contengono informazioni sul database e tutti gli oggetti in esso contenuti. Il dizionario dati è la risorsa più importante di cui dispone l’utilizzatore del database, qualunque sia il ruolo che egli ricopre.

Ogni tipo di oggetto del database ha associate una o più tabelle del dizionario che ne descrivono tutte le caratteristiche. Nei prossimi paragrafi ogni volta che sarà introdotto un tipo di oggetto, saranno parallelamente indicate le tabelle del dizionario che ne descrivono gli elementi. È possibile comunque indicare dei criteri generali di consultazione del dizionario.

La tabella principale si chiama DICTIONARY e contiene il nome e la descrizione di tutte le tabelle presenti nel dizionario. Per accedere alla tabella DICTIONARY è possibile utilizzare anche il nome breve DICT.

Una delle tabelle di dizionario più utilizzate è senz’altro la USER_CATALOG (abbreviata CAT) che visualizza tutti i principali oggetti di proprietà dell’utente connesso. Come piccola deroga all’ordine di esposizione degli argomenti bisogna anticipare il comando SQL utile per visualizzare tutti i dati presenti in una tabella:

```
SELECT * FROM <NOME_TABELLA>
```

Utilizzando questo comando ed il comando di SQL*Plus DESC su DICT e CAT verifichiamo come sono fatte le due tabelle di dizionario appena citate e cosa contengono:

```
WTO> desc dictionary
 Nome                                      Nullo?   Tipo
 ----------------------------------------- -------- -------------
 TABLE_NAME                                         VARCHAR2(30)
 COMMENTS                                           VARCHAR2(4000)

WTO> desc dict
 Nome                                      Nullo?   Tipo
 ----------------------------------------- -------- -------------
 TABLE_NAME                                         VARCHAR2(30)
 COMMENTS                                           VARCHAR2(4000)

WTO> desc user_catalog
 Nome                                      Nullo?   Tipo
 ----------------------------------------- -------- -------------
 TABLE_NAME                                NOT NULL VARCHAR2(30)
 TABLE_TYPE                                         VARCHAR2(11)

WTO> desc cat
 Nome                                      Nullo?   Tipo
 ----------------------------------------- -------- -------------
 TABLE_NAME                                NOT NULL VARCHAR2(30)
 TABLE_TYPE                                         VARCHAR2(11)

WTO> select * from cat;

Nessuna riga selezionata
```

Provando ad eseguire l'istruzione

```
WTO> select * from dict;
```

ci si renderà conto della notevole quantità di tabelle presenti nel dizionario…

Per una migliore visualizzazione del risultato in SQL*Plus si possono utilizzare i comandi

```
WTO> set line 156
WTO> col comments for a120
WTO> set pages 1000
```

Il primo comando imposta a 156 caratteri la larghezza della linea di output di SQL*Plus, il secondo stabilisce che la colonna "comments" deve avere una larghezza di 120 caratteri, il terzo che l'interruzione di pagina deve essere ripetuta ogni 1000 righe estratte.

Il risultato è presentato in Figura 5-1.

```
Seleziona SQL Plus
SQL> set line 156
SQL> col comments for a120
SQL> set pages 1000
SQL> select * from dict;

TABLE_NAME                     COMMENTS
------------------------------ ------------------------------------------------------------
USER_CONS_COLUMNS              Information about accessible columns in constraint definitions
ALL_CONS_COLUMNS               Information about accessible columns in constraint definitions
USER_LOG_GROUP_COLUMNS         Information about columns in log group definitions
ALL_LOG_GROUP_COLUMNS          Information about columns in log group definitions
USER_LOBS                      Description of the user's own LOBs contained in the user's own tables
ALL_LOBS                       Description of LOBs contained in tables accessible to the user
USER_CATALOG                   Tables, Views, Synonyms and Sequences owned by the user
ALL_CATALOG                    All tables, views, synonyms, sequences accessible to the user
USER_CLUSTERS                  Descriptions of user's own clusters
ALL_CLUSTERS                   Description of clusters accessible to the user
USER_CLU_COLUMNS               Mapping of table columns to cluster columns
USER_COL_COMMENTS              Comments on columns of user's tables and views
ALL_COL_COMMENTS               Comments on columns of accessible tables and views
USER_COL_PRIVS                 Grants on columns for which the user is the owner, grantor or grantee
ALL_COL_PRIVS                  Grants on columns for which the user is the grantor, grantee, owner,
                                or an enabled role or PUBLIC is the grantee

USER_COL_PRIVS_MADE            All grants on columns of objects owned by the user
ALL_COL_PRIVS_MADE             Grants on columns for which the user is owner or grantor
USER_COL_PRIVS_RECD            Grants on columns for which the user is the grantee
ALL_COL_PRIVS_RECD             Grants on columns for which the user, PUBLIC or enabled role is the grantee
ALL_ENCRYPTED_COLUMNS          Encryption information on all accessible columns
USER_ENCRYPTED_COLUMNS         Encryption information on columns of tables owned by the user
USER_INDEXES                   Description of the user's own indexes
ALL_INDEXES                    Descriptions of indexes on tables accessible to the user
USER_IND_COLUMNS               COLUMNs comprising user's INDEXes and INDEXes on user's TABLES
ALL_IND_COLUMNS                COLUMNs comprising INDEXes on accessible TABLES
USER_IND_EXPRESSIONS           Functional index expressions in user's indexes and indexes on user's tables
ALL_IND_EXPRESSIONS            FUNCTIONAL INDEX EXPRESSIONs on accessible TABLES
USER_JOIN_IND_COLUMNS          Join Index columns comprising the join conditions
ALL_JOIN_IND_COLUMNS           Join Index columns comprising the join conditions
USER_OBJECTS                   Objects owned by the user
ALL_OBJECTS                    Objects accessible to the user
USER_OBJECTS_AE                Objects owned by the user
ALL_OBJECTS_AE                 Objects accessible to the user
USER_ROLE_PRIVS                Roles granted to current user
USER_SEQUENCES                 Description of the user's own SEQUENCEs
ALL_SEQUENCES                  Description of SEQUENCEs accessible to the user
USER_SYNONYMS                  The user's private synonyms
ALL_SYNONYMS                   All synonyms for base objects accessible to the user and session
USER_TABLES                    Description of the user's own relational tables
USER_OBJECT_TABLES             Description of the user's own object tables
USER_ALL_TABLES                Description of all object and relational tables owned by the user's
ALL_TABLES                     Description of relational tables accessible to the user
ALL_OBJECT_TABLES              Description of all object tables accessible to the user
ALL_ALL_TABLES                 Description of all object and relational tables accessible to the user
USER_TAB_COLS                  Columns of user's tables, views and clusters
ALL_TAB_COLS                   Columns of user's tables, views and clusters
USER_TAB_COLUMNS               Columns of user's tables, views and clusters
ALL_TAB_COLUMNS                Columns of user's tables, views and clusters
USER_TAB_COMMENTS              Comments on the tables and views owned by the user
ALL_TAB_COMMENTS               Comments on tables and views accessible to the user
USER_TAB_PRIVS                 Grants on objects for which the user is the owner, grantor or grantee
ALL_TAB_PRIVS                  Grants on objects for which the user is the grantor, grantee, owner,
                                or an enabled role or PUBLIC is the grantee
```

Figura 5-1 Lettura del dizionario da SQL*Plus.

In alternativa si può utilizzare SQL Developer (Figura 5-2).

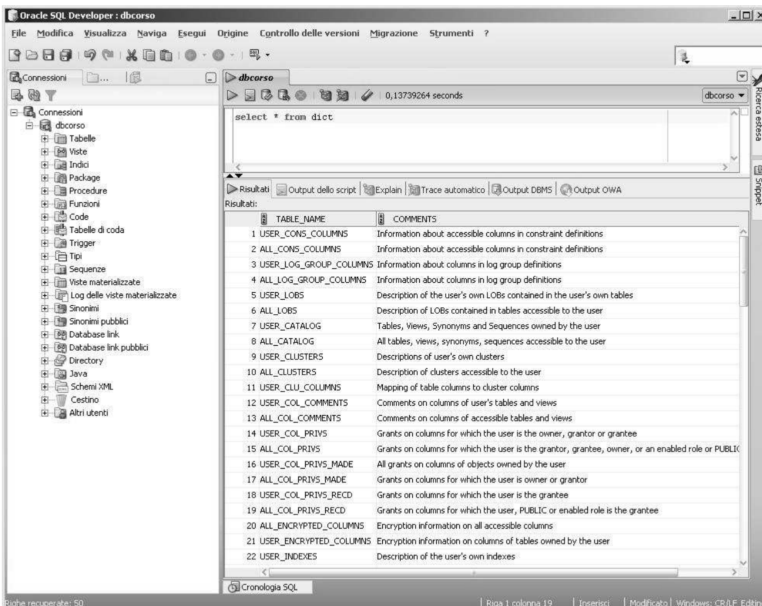


Figura 5-2 Lettura del dizionario da SQL Developer.

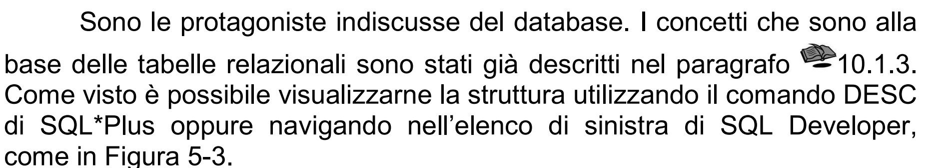
Le tabelle del dizionario possono essere classificate in quattro gruppi:

- **Tabelle il cui nome inizia per “USER_”** – contengono informazioni su tutti gli oggetti di proprietà dell’utente che esegue la query.
- **Tabelle il cui nome inizia per “ALL_”** – contengono informazioni su tutti gli oggetti a cui l’utente che esegue la query ha diritto di accedere. Un utente ha sicuramente diritto di accedere agli oggetti di sua proprietà, ma anche agli oggetti di proprietà di altri utenti per cui è stato autorizzato mediante il comando SQL GRANT.
- **Tabelle il cui nome inizia per “DBA_”** – contengono informazioni su tutti gli oggetti contenuti nel database, queste tabelle sono accessibili solo agli utenti “speciali” che hanno privilegi tali da essere considerati amministratori di database.
- **Tabelle il cui nome inizia per “V$” e “GV$”** – Si tratta delle cosiddette “viste dinamiche”. Il contenuto di queste tabelle cambia continuamente in funzione dello stato del database e non soltanto degli oggetti che vengono creati e distrutti. Da queste tabelle si può, per esempio, sapere in un determinato istante quanti utenti sono collegati al database e cosa stanno facendo oppure quanta memoria è in uso. Anche queste tabelle sono utilizzabili solo dagli utenti che hanno privilegi di amministratore.
- **Tabelle il cui nome inizia per “CDB_”** – presenti a partire da Oracle 12c, contengono informazioni su tutti gli oggetti contenuti nel database contenitore ed in tutti i pluggrable Db in esso contenuti.

Le tabelle degli ultimi tre gruppi contengono informazioni essenziali per l’amministrazione del database, quindi vanno oltre gli obiettivi introduttivi di questo corso. Le tabelle ALL sono praticamente identiche alle USER a meno del fatto che nelle ALL c’è in più la colonna OWNER che indica chi è il proprietario dell’oggetto (nella USER non ce n’è bisogno perché il proprietario è per definizione chi sta eseguendo la query). Per questi motivi nel seguito saranno menzionate sempre le tabelle USER.

5.2 Tabelle

Sono le protagoniste indiscusse del database. I concetti che sono alla base delle tabelle relazionali sono stati già descritti nel paragrafo 10.1.3. Come visto è possibile visualizzarne la struttura utilizzando il comando DESC di SQL*Plus oppure navigando nell’elenco di sinistra di SQL Developer, come in Figura 5-3.



```
WTO> DESC PROVA
 Nome                          Nullo?   Tipo
 ----------------------------- -------- ----------------
 A                                      NUMBER
 B                             NOT NULL VARCHAR2(20)
 C                                      DATE
```

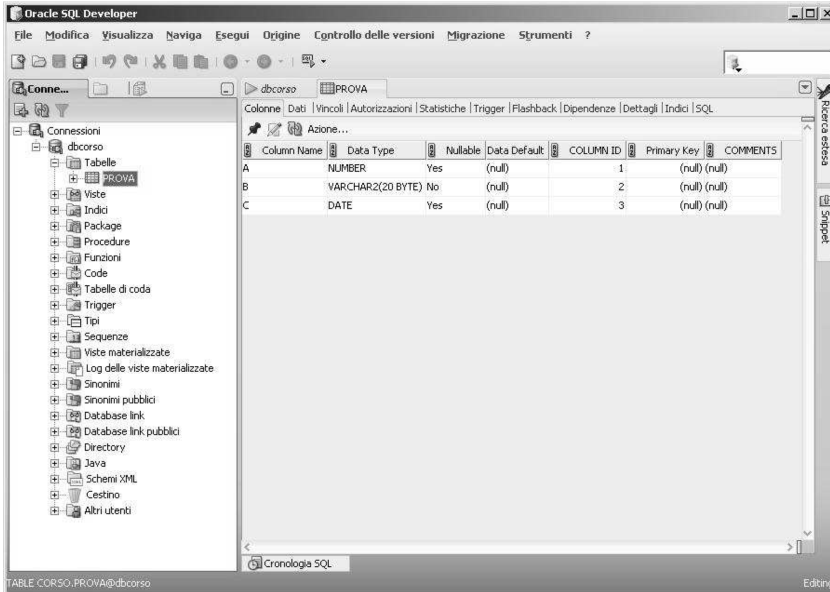


Figura 5-3 Caratteristiche principali di una tabella.

Le tecniche utilizzate da Oracle per la memorizzazione dei dati di una tabella sono stati già discusse, si può aggiungere che in casi del tutto particolari i dati di una tabella possono risiedere al di fuori del database (External Table) oppure essere suddivisi in sottotabelle (Partizioni) per velocizzare l’accesso ai dati. Questi tipi particolari di tabelle si comportano, dal punto di vista dell’utente, in maniera del tutto analogo alle tabelle standard.

5.2.1 Colonne di una tabella

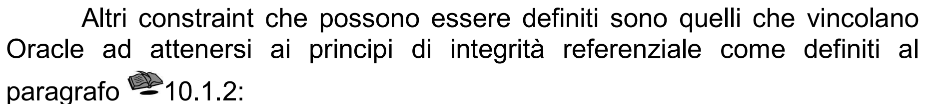
La caratteristica più importante di una tabella sono senz’altro le colonne che la compongono. Ogni colonna è descritta mediante un nome, un tipo di dato e la sua obbligatorietà. I tipi di dato saranno illustrati all’inizio del prossimo capitolo, si può qui anticipare che si tratta di una caratterizzazione della colonna che consente un miglior controllo da parte di Oracle dei dati inseriti. La scritta “NOT NULL” in SQL*Plus e l’attributo “Nullable=No” di SQL Developer indicano l’obbligatorietà della colonna, ogni volta che si inserirà una riga in tabella le colonne NOT NULL dovranno essere per forza valorizzate.

L’esempio semplicissimo di tabella visualizzato in Figura 5-3 possiede tre colonne. La prima si chiama “A”, è un numero e non è obbligatoria; la seconda si chiama “B” è una stringa alfanumerica a lunghezza variabile di massimo 20 caratteri ed è obbligatoria; la terza si chiama “C”, è una data e non è obbligatoria.

5.2.2 Constraint

Un constraint è una regola che devono verificare i dati per poter essere contenuti in tabella. Il primo tipo di constraint definito implicitamente che è stato già incontrato è la conformità al tipo di dato. Quando, in fase di creazione o di modifica della tabella, si indica che una colonna è di tipo numerico si sta implicitamente richiedendo ad Oracle di controllare i dati che saranno inseriti e rigettare tutto ciò che non è numerico.

Anche l'obbligatorietà di una colonna è un constraint. Definire una colonna "NOT NULL" equivale a definire un vincolo che Oracle dovrà verificare ogni qual volta i dati saranno inseriti o modificati.

Altri constraint che possono essere definiti sono quelli che vincolano Oracle ad attenersi ai principi di integrità referenziale come definiti al paragrafo 10.1.2:

- ***Primary Key* – Chiave primaria.** È un constraint che consente all'utente di definire che una colonna, oppure un insieme di colonne, funge da chiave primaria della tabella. Conseguenza dell'esistenza di questo constraint sarà che
    1. La colonna o le colonne inserite in chiave primaria diventano obbligatorie.
    2. Non sarà mai possibile inserire due righe in tabella aventi lo stesso valore, oppure la stessa combinazione di valori, nella chiave primaria.
- ***Foreign Key* – Chiave esterna.** È un constraint che consente all'utente di definire che una colonna, oppure un insieme di colonne, è un riferimento alla chiave primaria di un'altra tabella. La tabella su cui è definita la chiave esterna si dice tabella "figlia" mentre la tabella su cui si trova la chiave primaria referenziata si dice tabella "madre". Conseguenza dell'esistenza di questo constraint sarà che:
    1. La colonna o le colonne inserite in chiave esterna nella tabella figlia possono assumere solo valori già contenuti nella chiave primaria della tabella madre.
    2. Non sarà mai possibile cancellare una riga dalla tabella madre se ci sono righe nella tabella figlia che contengono quei valori.

Per maggiore chiarezza si faccia riferimento agli esempi del paragrafo 10.1.2.

In alcuni casi oltre alla primary key della tabella può succedere che altre colonne, o combinazioni di colonne, non debbano poter assumere gli stessi valori in righe differenti. Si pensi come esempio al codice fiscale nelle tabelle di anagrafe dei clienti. Il codice fiscale non può quasi mai essere utilizzato come chiave primaria perché spesso non viene fornito o viene fornito in un secondo momento dai clienti e dunque raramente si può considerare un campo obbligatorio. Quando viene fornito, però, è

normalmente un dato univoco. Non ci possono essere due clienti in anagrafe che hanno lo stesso codice fiscale, altrimenti c'è un errore oppure si tratta della stessa persona. Per queste situazioni esiste il constraint ***unique key***, chiave univoca.

Un altro tipo di vincolo molto utilizzato è il ***check constraint*** che consente di fissare una regola che i dati di una riga devono verificare in fase di inserimento o aggiornamento. Ad esempio si può verificare che un numero o una data siano compresi in un dato intervallo oppure che una colonna assuma solo i valori "S" oppure "N". Il controllo può coinvolgere anche più colonne della stessa riga, ad esempio se si sta definendo la tabella dei prestiti di una biblioteca si può imporre che la data restituzione sia maggiore o uguale della data del prestito.

Tutti i constraint possono essere temporaneamente disabilitati. La loro definizione, in questo caso, resta associata alla tabella ma essi non vengono più controllati. Alla riabilitazione del constraint Oracle ricontrolla i dati per verificare se questi violano il constraint da abilitare, in tal caso viene sollevato un errore.

5.3 Indici

Uno dei maggiori problemi nell'utilizzo dei database è l'ottimizzazione dei tempi di risposta delle ricerche. Frequentemente le tabelle contengono milioni di righe e spesso vengono lette insieme a molte altre tabelle in istruzioni SQL molto complesse.

L'ottimizzazione di un database Oracle è un'attività di elevata complessità che richiede competenze di tipo amministrativo e sicuramente gli si potrebbe dedicare un manuale a se stante. In questo corso il tema sarà approcciato solo da un particolare punto di vista, quello dell'utente di database che scrive una query. Volendosi concentrare solo sull'aspetto "utente" dell'ottimizzazione il tema fondamentale da trattare è sicuramente quello degli *indici:* strutture di dati che consentono l'individuazione veloce di un valore specifico in una colonna di database.

Si ipotizzi dunque di avere la solita tabella CLIENTI contenente il codice del cliente, il cognome, il nome e tutti gli altri dati anagrafici. Si ipotizzi poi che la tabella contenga un milione di righe.

I record della tabella non sono registrati nel database in nessun ordine specifico, se eseguiamo quindi la ricerca dei clienti il cui cognome è "ROSSI" Oracle sarà costretto a scorrere un milione di righe, scartare tutti i clienti che non si chiamano Rossi e prendere gli altri. Sarebbe tutto molto più semplice se i record fossero ordinari per cognome, in quel caso infatti mediante una ricerca dicotomica (10.6) sarebbe possibile trovare i Rossi in, al più, 21 letture. Oracle però non può certo archiviare i dati sul disco in uno specifico ordine, quindi è necessario utilizzare delle strutture dati aggiuntive che consentano di velocizzare l'accesso.

5.3.1 Indici B-Tree

Definire un indice sulla colonna Cognome della tabella CLIENTI significa chiedere ad Oracle di creare, e tenere sempre aggiornata, una nuova struttura dati in cui siano presenti in una struttura ad albero tutti i cognomi della tabella CLIENTI, ad ogni cognome sarà associato un codice (*rowid*) che indica la collocazione fisica (file, blocco, riga) del record che contiene quel valore. L'albero è composto da un certo numero di livelli, al primo livello c'è un solo gruppo che fa da indice tra i gruppi di secondo livello, così procedendo ad ogni livello ogni gruppo gestisce un gruppo sempre più piccolo di cognomi e fa da indice per il gruppo di livello inferiore. All'ultimo livello, nelle foglie dell'albero, ci sono i singoli cognomi ognuno associato al suo rowid. Per un esempio si faccia riferimento alla Figura 5-4. Dovendo cercare un cognome particolare Oracle partirà dal primo nodo in altro e percorrerà l'albero un livello alla volta fino ad arrivare alla foglia che contiene il cognome cercato. A quel punto utilizzerà il rowid per recuperare il record legato a quel cognome.

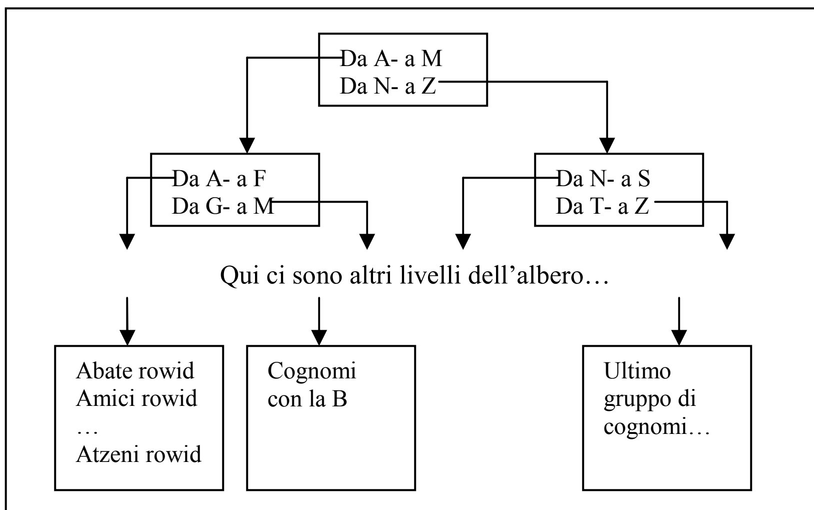


Figura 5-4 Struttura di un indice B-tree.

Questo tipo di indice si chiama B-Tree (balanced tree, albero bilanciato). Se in questo tipo di indice ogni gruppo intermedio puntasse ad esattamente due gruppi di livello inferiore, come nell'esempio, sarebbe praticamente equivalente all'utilizzo di una ricerca dicotomica. Oracle però può decidere quanti gruppi intermedi creare in funzione dei dati da gestire.

Un indice è ovviamente un segment e può essere definito anche su insiemi di colonne della stessa tabella. Ovviamente in questo caso la posizione delle colonne nell'indice fa la differenza visto che la distribuzione nell'albero avverrà in base al valore della prima colonna e poi, solo a parità di valore nella prima colonna, su quello della seconda e su tutte le altre.

Il B-Tree è il tipo di indice predefinito in Oracle ma genera dei notevoli problemi quando una colonna indicizzata assume pochi valori distinti. Ipotizzando di indicizzare il campo SESSO, che assume solo due possibili valori, la ricerca di tutti i clienti cha hanno SESSO=M sarà più lenta di una ricerca sequenziale. Ipotizzando infatti che la metà dei clienti sia maschio l'accesso all'indice restituirà cinquecentomila rowid e per ognuno di esso sarà fatto un accesso al datafile per recuperarne i dati. In questo scenario la presenza dell'indice B-Tree non solo non agevola, ma addirittura peggiora drasticamente i tempi di esecuzione della ricerca.

5.3.2 Indici Bitmap

Gli indici di tipo *Bitmap* nascono per velocizzare l'accesso a quelle colonne che assumono solo pochi valori distinti, come la colonna SESSO che può valere solo M o F. Un indice di tipo B-Tree contiene tante foglie per quanti sono i record presenti nella tabella indicizzata, un indice di tipo bitmap, invece, contiene una foglia per ogni valore della colonna, nel caso della colonna SESSO, l'indice avrebbe solo due foglie, una per il valore M, l'altra per il valore F. Ad ognuna delle due foglie è associata una mappa di bit costituita da un bit per ogni record della tabella. Ogni bit della mappa indica se un particolare rowid contiene oppure no il valore di SESSO a cui la foglia si riferisce.

5.3.3 Più indici sullo stesso insieme di colonne

A differenza delle precedenti versioni, in Oracle 12c è possibile creare più indici sullo stesso insieme di colonne (con lo stesso ordine) purché gli indici differiscano per una o più delle seguenti caratteristiche:

- Tipo di indice (B-tree/bitmap)
- Partizionamento degli indici
- Indice Univoco/Non univoco

Uno solo degli indici può essere visibile in un determinato istante. Creiamo una tabella con una sola colonna:

```
O12c>create table test_ind (
  2  id number);

Tabella creata.
```

Cominciamo con un indice B-tree univoco:

```
O12c>create unique index test1 on test_ind(id);

Indice creato.
```

Poi cerchiamo di crearne uno bitmap:

```
O12c>create bitmap index test2 on test_ind(id);
create bitmap index test2 on test_ind(id)
                                      *
ERRORE alla riga 1:
ORA-01408: esiste giÓ un indice per questa lista di colonne
```

Non è possibile perché il primo è visibile, allora procediamo rendendo prima invisibile il primo indice:

```
O12c>alter index test1 invisible;

Modificato indice.

O12c>create bitmap index test2 on test_ind(id);

Indice creato.
```

Procedendo con la stessa logica, rendiamo invisibile il secondo indice e poi creiamone un terzo, è sempre B-tree ma non univoco:

```
O12c>alter index test2 invisible;

Modificato indice.

O12c>create index test3 on test_ind(id);

Indice creato.
```

Per vedere la situazione complessiva degli indici eseguiamo una semplice query:

```
O12c>select index_name, INDEX_TYPE, uniqueness, visibility
  2  from dba_indexes
  3  where table_name='TEST_IND';

INDEX_NAME   INDEX_TYPE   UNIQUENES VISIBILITY
------------ ------------ --------- ----------
TEST3        NORMAL       NONUNIQUE VISIBLE
TEST2        BITMAP       NONUNIQUE INVISIBLE
TEST1        NORMAL       UNIQUE    INVISIBLE
```

E adesso cerchiamo di rendere visibile l’indice TEST2, ovviamente ottenendo un errore:

```
O12c>alter index test2 visible;
alter index test2 visible
*
ERRORE alla riga 1:
ORA-14147: Sullo stesso set di colonne Þ giÓ stato definito un indice
VISIBLE.
```

5.4 IOT

Una tabella *index-organized* ha tutte le caratteristiche di una tabella classica a parte il fatto che, internamente, i dati sono archiviati in una struttura B-Tree basata sulla primary key.

Ovviamente, mentre una tabella classica può essere priva di primary key, una IOT ne deve essere necessariamente fornita.

Dal punto di vista dell’utente non c’è alcuna differenza tra una tabella di questo tipo ed una classica.

Conviene utilizzate una IOT quando la maggior parte delle ricerche sulla tabella includono tra i criteri di ricerca una condizione sulla primary key.

Una index-organized table è, ovviamente, un segment.

5.5 Cluster di tabelle

Come detto nel paragrafo 1.4.1 un blocco di memoria contiene solitamente diverse righe di un'unica tabella. Unica eccezione a questa regola è quella dei *cluster di tabelle*.

Nel caso in cui le righe di due tabelle siano logicamente legate e quindi vengano lette quasi sempre contemporaneamente è possibile fare in modo che Oracle le tratti, dal punto di viste della memorizzazione fisica, come se fossero un unico segment. Quest'unico segment prende il nome di cluster.

In fase di definizione del cluster è necessario individuare la chiave di clusterizzazione, si tratta di un'informazione presente in entrambe le tabelle che viene utilizzata per stabilire quali righe delle due tabelle devono essere archiviate nello stesso blocco. Normalmente viene utilizzata una foreign key, sebbene sia possibile definire il cluster anche in assenza di chiavi esterne.

Per ogni valore della chiave Oracle conserva nello stesso blocco, o insieme di blocchi di database, tutte le righe di entrambe le tabelle che hanno quel valore della chiave. Per default Oracle riserva un blocco per ogni valore della chiave, nel comando di creazione del cluster si può indicare di riservare uno spazio maggiore per ogni valore.

Esistono due tipi di cluster, in funzione della tecnica che Oracle utilizza per trovare i dati nella chiave di clusterizzazione e quindi scegliere quali blocchi leggere quando l'utente esegue una query.

Negli INDEXED CLUSTER Oracle utilizza un indice per associare ad un valore della chiave di clusterizzazione il blocco in cui i dati relativi sono conservati. L'indice, detto cluster index, deve essere creato sul cluster prima di poter inserire dei record.

Negli HASH CLUSTER Oracle utilizza un algoritmo di hash (10.7) al posto dell'indice. Chi crea il cluster non deve quindi creare l'indice ma deve indicare quanti valori distinti della chiave di clusterizzazione saranno approssimativamente presenti.

5.6 Viste

Una tabella è, come visto, un contenitore di dati generalmente relativi ad una specifica entità del sistema. Ad esempio si potrebbe definire la tabella dei clienti, quella degli ordini oppure quella delle fatture. Normalmente accade di dover accedere ai dati di più tabelle contemporaneamente, voler quindi vedere contemporaneamente i dati di clienti, ordini e fatture. Ciò può essere effettuato mediante una query su più tabelle, si tratta di una istruzione SQL complessa del tipo descritto nel paragrafo 7.5.11 e successivi. Una vista non è altro che un nome assegnato ad una tale query SQL complessa. Una

vista dunque non contiene dati, ma solo la definizione di una istruzione SQL. L'utente però interagisce con la vista come se fosse una qualunque tabella, può vederne la struttura utilizzando il comando DESC e leggerne tutti i dati con il comando

```
SELECT * FROM <NOME_VISTA>
```

Una vista può essere utilizzata in istruzioni SQL complesse insieme a tabelle oppure ad altre viste. Chiaramente la vista non contiene dati, non è un segment. Ogni volta che un utente esegue una istruzione SQL contenente una vista Oracle trasforma l'istruzione sostituendo il nome della vista con la query SQL ad essa associata.

Oltre che per leggere i dati da più tabelle una vista può anche essere utilizzata per leggere solo una parte dei dati di una tabella. Si pensi al caso di una anagrafica in cui una colonna SESSO indichi il sesso della persona. Sarebbe possibile derivare da questa tabella due viste, ad esempio denominate MASCHI e FEMMINE, ognuna delle quali estragga solo le persone appartenenti ad uno dei due sessi.

Si tratta in generale di un tipo di vista più semplice perché legge i dati da una sola tabella e si limita a filtrarne le righe ed eventualmente le colonne. Queste viste più semplici possono essere utilizzate non solo per leggere i dati, ma anche in fase di inserimento ed aggiornamento. Ovviamente un inserimento nella vista sarà automaticamente trasformato in un inserimento sulla tabella sottostante.

5.6.1 Check option constraint

Nel caso, appena descritto, di vista "inseribile" si può verificare una situazione paradossale. Ipotizziamo che la tabella PERSONE contenga la colonna SESSO valorizzata ad F o M e che le viste FEMMINE e MASCHI leggano, rispettivamente, solo le righe aventi SESSO=F e quelle aventi SESSO=M. È possibile inserire una riga in FEMMINE impostando il valore di SESSO a M. La riga viene correttamente inserita in PERSONE ma non potrà mai essere letta mediante una query dalla vista FEMMINE. Apparirà invece nelle query dalla vista MASCHI. È quindi possibile inserire in una vista una riga di dati che poi non potrà mai essere letta attraverso quella stessa vista.

Per evitare che si verifichi questa situazione è possibile, in fase di creazione della vista, attivare il constraint *check option*.

Quando questo constraint è attivo Oracle impedisce di inserire un record in una vista se questo non può essere poi letto usando la stessa vista. Nel nostro esempio tentando di inserire una riga in FEMMINE con SESSO=M si otterrebbe un errore.

5.7 Viste materializzate

Le viste sono strumenti molto comodi per semplificare la scrittura delle query SQL più complesse, talvolta però, in caso molto complessi dove le tabelle coinvolte sono molte e molto ricche di righe, la lettura di una vista può

essere piuttosto lenta. Oltre ad applicare tutti i metodi di ottimizzazione del database o delle singole istruzioni SQL, come la creazione di indici, in queste occasioni si può ricorrere alle *viste materializzate*, in inglese materialized views.

Tali oggetti vengono definiti in maniera simile alle viste ma, all'atto della creazione, vengono effettivamente riempite con i dati estratti dalla query SQL. Ovviamente questi dati "duplicati" diventano obsoleti non appena cambiano i dati contenuti in una delle tabelle su cui la vista insiste. Per tale motivo è necessario aggiornare la vista materializzata manualmente prima di utilizzarla oppure impostare l'aggiornamento automatico periodico.

L'aggiornamento può essere completo o rapido (*fast* in inglese). L'aggiornamento completo cancella tutti i record dalla vista e reinserisce il risultato della query completa. Per eseguire l'aggiornamento rapido, Oracle tiene traccia in una tabella di log delle modifiche intervenute sulle tabelle originali dall'ultimo aggiornamento e, al momento del nuovo aggiornamento, riporta solo i cambiamenti effettivi.

Se in fase di definizione della vista non si è definita una politica di aggiornamento automatico periodico, si può eseguire un aggiornamento manuale utilizzando la procedura REFRESH del package di sistema DBMS_MVIEW.

Le viste materializzate occupano un proprio spazio, quindi sono dei segment.

5.8 Sequenze

Le sequenze sono generatori di numeri progressivi univoci. Sono spesso utilizzate per valorizzare la chiave primaria di una tabella.

Una sequenza può partire dal numero uno oppure da un altro numero a scelta, incrementarsi o decrementarsi di una o più unità ad ogni utilizzo, avere un valore massimo predefinito oppure no, riprendere dal numero iniziale quando si esaurisce oppure no.

La sequenza non è un segment.

5.9 Sinonimi

Un sinonimo è un nome alternativo che viene assegnato ad un oggetto del database. Abbiamo già incontrato alcuni sinonimi, DICT è sinonimo di SYS.DICTIONARY, CAT è sinonimo di SYS.USER_CATALOG.

Non c'è un limite al numero di sinonimi che possono essere assegnati ad un oggetto. Il sinonimo può essere privato, cioè utilizzabile solo dall'utente che lo definisce, oppure pubblico cioè utilizzabile da tutti gli utenti del database. DICT e CAT sono sinonimi pubblici.

Un sinonimo può essere pubblico o privato. Un sinonimo privato può essere ovviamente utilizzato dall'utente che lo ha creato, gli altri utenti possono utilizzarlo solo se sono stati autorizzati e devono anteporre al nome

del sinonimo il nome dello schema in cui è stato definito. Un sinonimo pubblico può essere utilizzato da tutti gli utenti del database senza anteporre mai il nome di uno scema. Di fatto il sinonimo pubblico appartiene ad uno schema speciale, pubblico appunto, i cui oggetti sono accessibili a chiunque utilizzandone solo il nome.

I sinonimi non sono segment.

5.10 Database link

Un *database link* consente di mettere in collegamento due database, in modo che uno dei due possa leggere le tabelle dell'altro. Il collegamento non è bidirezionale, quindi se entrambi i db hanno bisogno di accedere all'altro ambiente bisogna creare due database link, uno su ogni db.

Per la connessione viene utilizzata la tecnica della connect string spiegata nel paragrafo 3.2.3, quindi nel db che fa da client ci sarà un tnsnames.ora in cui sono archiviati i parametri necessari per la connessione.

5.11 Oggetti PL/SQL

Il database Oracle fornisce il PL/SQL, un utilissimo linguaggio di programmazione che incorpora l'SQL in un linguaggio procedurale standard. Programmi di diverso tipo scritti in PL/SQL possono essere salvati nel DB al fine di essere richiamati quando serve. Accenniamo qui brevemente a questi oggetti rimandando al capitolo dedicato al PL/SQL per gli approfondimenti.

5.11.1 Procedure

Una procedura PL/SQL è un programma registrato nel DB che accetta un determinato numero di parametri di input/output ed esegue un'attività senza tornare uno specifico valore al chiamante. Una procedura non può essere utilizzata in un comando SQL, ma solo in un altro PL/SQL.

5.11.2 Funzioni

Una funzione PL/SQL è un programma registrato nel DB che accetta un determinato numero di parametri di input/output ed esegue un'attività tornando uno specifico valore al chiamante. Le funzioni possono essere utilizzate anche all'interno dei comandi SQL a patto di non modificare nulla nel database.

5.11.3 Package

Un Package PL/SQL è un insieme di procedure, funzioni e variabili PL/SQL che vengono riunite in un unico oggetto. L'organizzazione degli oggetti PL/SQL in package garantisce vari vantaggi che saranno approfonditi nel capitolo dedicato al PL/SQL.

5.11.4 Trigger

Un Trigger è un programma PL/SQL registrato nel DB ed associato ad un evento. Il trigger non può essere richiamato per l'esecuzione esplicitamente ma scatta automaticamente quando si verifica l'evento a cui è associato. Ad esempio è possibile creare un trigger BEFORE INSERT sulla tabella CLIENTI che scatta automaticamente subito prima dell'inserimento di qualunque record in tabella.

5.12 Estensioni Object-Oriented

Con la versione 8 del database, Oracle ha introdotto una serie di estensioni Object-Oriented che consentono di mappare in modo più naturale sul DB istanze di classi definite con linguaggi di programmazione ad oggetti. In fase di lancio di Oracle 8 c'era stata una forte attenzione su queste nuove funzionalità del db. Poi, però, l'esplosione di internet ha spostato l'attenzione su altre caratteristiche e la stessa Oracle ha cambiato direzione denominando la seconda release della stessa versione Oracle 8i (dove i, appunto, sta per internet). Le estensioni OO hanno dunque perso molta visibilità ed hanno finito per essere complessivamente poco utilizzate. Sono state comunque conservate per compatibilità con il passato in tutte le versioni successive di Oracle. In questo corso non saranno approfondite.

5.13 XML DB

Con la seconda release della versione 9 del DB Oracle ha introdotto un insieme di nuove funzionalità che consentono di conservare nel db, e manipolare molto più semplicemente del passato, contenuti in formato XML. Tali funzionalità sono accorpate sotto il nome di Oracle XML DB. Queste caratteristiche sono state nelle versioni successive ulteriormente implementate raggiungendo un buono stato di maturità e completezza. In questo corso è stato riservato all'argomento il capitolo **Errore. L'origine riferimento non è stata trovata.**. In quel capitolo non ci si propone di trattare il tema in modo organico, tale trattazione richiederebbe infatti troppo spazio ed è sicuramente al di là degli obiettivi di questo corso. L'obiettivo del capitolo **Errore. L'origine riferimento non è stata trovata.**, invece, è di fornire alcuni spunti di riflessione relativi ad alcune specifiche tematiche che accendano l'interesse del lettore sull'argomento.

Il database XML DB a partire dalla versione 12c è un componente obbligatorio, non può essere disinstallato e non c'è alcuna opzione che consenta di non includerlo quando si crea un nuovo database.

6 CASE STUDY

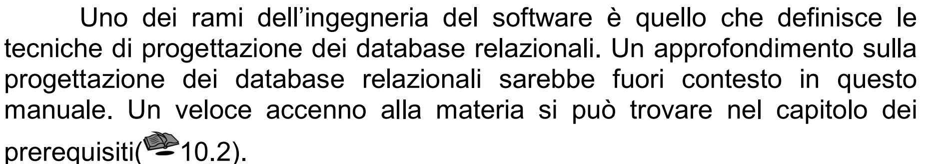
I prossimi argomenti, SQL e PL/SQL, richiedono una massiccia dose di esempi ed esercitazioni.

Per seguire una trattazione coerente ed omogenea è comodo definire un piccolo progetto che sarà sviluppato, a partire dalla creazione delle tabelle, nei prossimi capitoli.

6.1 Progettazione

Un'applicazione software è a tutti gli effetti un prodotto ingegneristico. Nessuna persona sana di mente penserebbe di poter costruire un ponte o un edificio senza prima averne fatto realizzare ad un professionista competente un progetto completo. Nel mondo del software, invece, capita spesso di cominciare a produrre, cioè a "scrivere il codice", senza avere prima condotto una fase di progettazione completa e rigorosa oppure avendo affidato la fase di progettazione a personale non sufficientemente qualificato. Questa è indubbiamente la causa della maggior parte dei fallimenti di progetti informatici.

La progettazione di un'applicazione software, dunque, è un'attività molto delicata che rientra nel campo di studio di una scienza rigorosa detta "ingegneria del software".

Uno dei rami dell'ingegneria del software è quello che definisce le tecniche di progettazione dei database relazionali. Un approfondimento sulla progettazione dei database relazionali sarebbe fuori contesto in questo manuale. Un veloce accenno alla materia si può trovare nel capitolo dei prerequisiti(10.2).

Lo schema tabellare descritto nel seguito è in terza forma normale.

6.2 Il database d'esempio

Il piccolo database utilizzato per gli esempi di questo manuale è pensato per contenere informazioni relative alla gestione di ordini e fatture. Sono previste tabelle per contenere i dati anagrafici dei clienti, gli ordini che questi dispongono e le fatture che vengono emesse. Le tabelle del database

sono descritte nel prossimo paragrafo, è possibile scaricare gli script completi per la creazione ed il popolamento delle tabelle e degli altri oggetti di database all'indirizzo web http://oracleitalia.wordpress.com/WelcomeToOracle. Alla stessa pagina sono anche disponibili le informazioni sulle modalità di esecuzione degli script.

L'utente WTO_ESEMPIO che viene creato dallo script, nel cui schema sono definiti tutti gli oggetti che seguono, è stato munito dei ruoli e privilegi minimi al fine di poter eseguire tutti gli esempi presenti nel corso. In particolare gli sono stati attribuiti i ruoli CONNECT e RESOURCE ed i privilegi di sistema CREATE VIEW, CREATE MATERIALIZED VIEW, CREATE SYNONYM, CREATE DATABASE LINK, CREATE PUBLIC SYNONYM.

6.2.1 Tabelle

Per ogni tabella prevista è fornito l'elenco delle colonne. Per ogni colonna viene indicato il nome, il tipo di dato, se la colonna è obbligatoria, se fa parte della chiave primaria, se fa parte di una chiave univoca, se fa parte di una chiave esterna ed eventuali note. Il tipo di dato è espresso utilizzando i tipi Oracle descritti all'inizio del prossimo capitolo.

CLIENTI

Nome Colonna	**Tipo**	**Obbl.**	**PK**	**UK**	**FK**	**Note**
COD_CLIENTE	NUMBER(8)	SI	SI			I valori di questa colonna sono generati mediante la sequenza SEQ_CLIENTI
NOME	VARCHAR2(30)	SI				
COGNOME	VARCHAR2(30)	SI				
COD_FISC	CHAR(16)			SI		
INDIRIZZO	VARCHAR2(30)	SI				
CAP	CHAR(5)	SI				
COMUNE	CHAR(4)	SI			SI	Referenzia la tabella COMUNI

COMUNI

Nome Colonna	**Tipo**	**Obbl.**	**PK**	**UK**	**FK**	**Note**
COD_COMUNE	CHAR(4)	SI	SI			
DES_COMUNE	VARCHAR2(50)	SI				
PROVINCIA	CHAR(2)	SI				

PRODOTTI

Nome Colonna	Tipo	Obbl.	PK	UK	FK	Note
COD_PRODOTTO	VARCHAR2(12)	SI	SI			
DESCRIZIONE	VARCHAR2(50)	SI				
PREZZO_LISTINO	NUMBER(7,2)	SI				

ORDINI

Nome Colonna	Tipo	Obbl.	PK	UK	FK	Note
NUM_ORDINE	NUMBER(8)	SI	SI			I valori di questa colonna sono generati mediante la sequenza SEQ_ORDINI
DATA_ORDINE	DATE	SI				
COD_CLIENTE	NUMBER(8)	SI			SI	Referenzia la tabella CLIENTI
IMPORTO	NUMBER(10,2)	SI				
NOTE	VARCHAR2(100)					

PRODOTTI_ORDINATI

Nome Colonna	Tipo	Obbl.	PK	UK	FK	Note
NUM_ORDINE	NUMBER(8)	SI	SI		SI	Referenzia la tabella ORDINI
COD_PRODOTTO	VARCHAR2(12)	SI	SI		SI	Referenzia la tabella PRODOTTI
QUANTITA	NUMBER	SI				
PREZZO_UNITARIO	NUMBER(7,2)	SI				

FATTURE

Nome Colonna	Tipo	Obbl.	PK	UK	FK	Note
NUM_FATTURA	NUMBER(8)	SI	SI			I valori di questa colonna sono generati mediante la sequenza SEQ_FATTURE
NUM_ORDINE	NUMBER(8)	SI			SI	Referenzia la tabella ORDINI
DATA_FATTURA	DATE	SI				
IMPORTO	NUMBER(7,2)	SI				
PAGATA	CHAR(1)	SI				Valori ammessi ‘S’, ‘N’

6.2.2 Dati

Una volta lanciati i due script sarà creato uno schema denominato WTO_ESEMPIO contenente tutte le tabelle sopra descritte ed i seguenti dati:

```
WTO >select * from clienti;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964

WTO >select * from comuni;

COD_ DES_COMUNE                                         PR
---- -------------------------------------------------- --
H501 ROMA                                               RM
G811 POMEZIA                                            RM
M297 FIUMICINO                                          RM
F205 MILANO                                             MI
G686 PIOLTELLO                                          MI
E415 LAINATE                                            MI
F839 NAPOLI                                             NA
G964 POZZUOLI                                           NA
G902 PORTICI                                            NA

WTO >select * from prodotti;

COD_PRODOTTO DESCRIZIONE                          PREZZO_LISTINO
------------ ------------------------------------ --------------
TVLCD22      Televisore LCD 22 pollici                       300
TVLCD32      Televisore LCD 32 pollici                       500
TVLCD40      Televisore LCD 40 pollici                       700
NOTEBOOK1    PC Notebook tipo 1                              500
NOTEBOOK2    PC Notebook tipo 2                              800
NOTEBOOK3    PC Notebook tipo 3                             1200
EBOOK        E-book reader                                   200
ROUTER       Router N300                                     100

WTO >select * from ordini

NUM_ORD DATA_ORDI COD_CLI IMPORTO NOTE
------- --------- ------- ------- ------------------------------
      1 01-OTT-10       1     800 Sconto 60 euro per promozione.
```

```
       2 20-OTT-10        2     700
       3 01-NOV-10        4    1000 Fattura differita 90gg
       4 01-NOV-10        5    1200
       5 01-DIC-10        7    1700

WTO >select * from prodotti_ordinati;

NUM_ORDINE COD_PRODOTTO   QAUNTITA PREZZO_UNITARIO
---------- ------------ ---------- ---------------
         1 TVLCD32               1             500
         1 EBOOK                 2             180
         2 TVLCD40               1             700
         3 ROUTER               10             100
         4 NOTEBOOK3             1            1200
         5 TVLCD40               1             700
         5 NOTEBOOK3             1            1000

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S
```

6.3 SCOTT/TIGER

Il database Oracle fornisce per default alcuni schemi d'esempio muniti di alcune tabelle già popolate. Questi schemi sono utili per i corsi di formazione. Il più famoso di questi schemi è senz'altro SCOTT (password *tiger*), presente fin dalle primissime versioni di Oracle.

L'utente SCOTT inizialmente ha l'account bloccato. Tentando di connettersi a questo schema da SQL*Plus si otterrà un errore.

```
WTO> conn scott/tiger
ERROR:
ORA-28000: the account is locked
```

Prima di utilizzarlo, dunque, è necessario sbloccarlo. Connessi con un utente DBA, SYSTEM ad esempio, si procede come nell'esempio seguente:

```
WTO> alter user scott account unlock;

Utente modificato.
```

A questo punto, visto che la password di SCOTT è inizialmente scaduta, alla connessione bisognerà impostare una nuova password.

```
WTO> conn scott/tiger
ERROR:
ORA-28001: the password has expired

Cambio password per scott
Nuova password: *****
Ridigitare la nuova password: *****
Password cambiata
Connesso.
```

Lo schema SCOTT è munito delle seguenti tabelle:

```
WTO> select * from cat;

TABLE_NAME                     TABLE_TYPE
------------------------------ -----------
BONUS                          TABLE
DEPT                           TABLE
EMP                            TABLE
SALGRADE                       TABLE

WTO> select * from emp;

EMPNO ENAME    JOB         MGR HIREDATE        SAL      COMM DEPTNO
----- -------- --------- ----- --------- --------- --------- ------
 7369 SMITH    CLERK      7902 17-DEC-80       800               20
 7499 ALLEN    SALESMAN   7698 20-FEB-81      1600       300     30
 7521 WARD     SALESMAN   7698 22-FEB-81      1250       500     30
 7566 JONES    MANAGER    7839 02-APR-81      2975               20
 7654 MARTIN   SALESMAN   7698 28-SEP-81      1250      1400     30
 7698 BLAKE    MANAGER    7839 01-MAY-81      2850               30
 7782 CLARK    MANAGER    7839 09-JUN-81      2450               10
 7788 SCOTT    ANALYST    7566 19-APR-87      3000               20
 7839 KING     PRESIDENT       17-NOV-81      5000               10
 7844 TURNER   SALESMAN   7698 08-SEP-81      1500         0     30
 7876 ADAMS    CLERK      7788 23-MAY-87      1100               20
 7900 JAMES    CLERK      7698 03-DEC-81       950               30
 7902 FORD     ANALYST    7566 03-DEC-81      3000               20
 7934 MILLER   CLERK      7782 23-JAN-82      1300               10

14 rows selected.

WTO> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

WTO> select * from bonus;
no rows selected

WTO> select * from salgrade;

     GRADE      LOSAL      HISAL
---------- ---------- ----------
         1        700       1200
         2       1201       1400
         3       1401       2000
         4       2001       3000
         5       3001       9999
```

Le tabelle di SCOTT saranno di tanto in tanto utilizzate negli esempi di questo manuale.

7 SQL

La principale attività da compiere con un database è la manipolazione dei dati e delle strutture che li contengono. Per realizzare queste attività ogni database mette a disposizione uno specifico linguaggio. Il linguaggio utilizzato per la gestione dei database relazionali si chiama SQL, Structured Query Language. Creato originariamente da IBM negli anni ’70 il linguaggio è diventato uno standard ISO nel 1986, quindi in teoria dovrebbe essere uguale per tutti i database relazionali. In realtà, però, tutti i produttori di database, pur adeguandosi abbastanza allo standard, hanno introdotto nei propri linguaggi SQL delle particolarità oppure funzionalità aggiuntive che ne fanno, in pratica, dei dialetti dello standard. Oracle non fa eccezione a questa tendenza.

Questo corso è dedicato al database Oracle, di conseguenza da questo momento in poi parleremo di SQL nella sola accezione di “versione di SQL fornita da Oracle”.

L’insieme delle istruzioni SQL può essere macroscopicamente suddiviso in quattro sottoinsiemi.

- Le istruzioni DDL, Data Definition Language, che consentono di creare, modificare ed eliminare gli oggetti contenuti nel database.
- Le istruzioni DML, Data Manipulation Language, che consentono di inserire, modificare, cancellare i dati presenti negli oggetti contenuti nel database.
- Le istruzione DCL, Data Control Language, che consentono di concedere e revocare agli utenti di database il permesso di accedere agli oggetti di un altro utente. Talvolta queste istruzioni sono considerate parte delle istruzioni DDL.
- Le istruzioni TCL, Transaction Control Language, che consentono di gestire le transazioni, cioè le sessioni di lavoro degli utenti. Mediante questi comandi è possibile confermare le modifiche apportate ai dati o annullarle completamente o parzialmente. Talvolta queste istruzioni sono considerate parte delle istruzioni DML.

L'istruzione SELECT, che consente di leggere i dati dal database ma non di modificarli, viene normalmente fatta rientrare nel linguaggio DML sebbene forse sarebbe più corretto considerarla in un insieme a se stante.

Nello spirito "introduttivo" di questo corso non si analizzeranno i comandi SQL nel dettaglio di tutte le opzioni. Ci si limiterà agli aspetti principali utilizzati nella maggior parte dei casi. Per un'analisi dettagliata dei comandi con riferimento a tutte le opzioni si rimanda alle 1508 pagine del manuale "SQL Language Reference" scaricabile liberamente, come tutta la documentazione del db, dal sito http://www.oracle.com/.

Negli esempi capiterà talvolta di utilizzare comandi che ancora non sono stati descritti. Si è ricorso a questa soluzione quando si voleva subito spiegare un concetto con un esempio senza rinviare ad un momento successivo. In tutti questi casi i comandi utilizzati "in anticipo" sono molto intuitivi e comunque commentati.

Tutti i comandi SQL sono case-insensitive, dunque possono essere scritti indifferentemente in maiuscolo, in minuscolo o con una qualunque mescolanza di caratteri minuscoli e maiuscoli. Le uniche parti delle istruzioni che debbono essere considerate case-sensitive sono quelle che appaiono racchiuse da apici singoli o doppi, ad esempio le due seguenti

```
'Contenuto Case-Sensitive'

"Contenuto Case-Sensitive"
```

7.1 Tipi di dato

Dal punto di vista strutturale i dati non sono tutti uguali. Una data di nascita ha, ovviamente, una serie di caratteristiche ben diverse dall'importo di una fattura. Ovviamente in teoria tutti i dati potrebbero essere trattati come stringhe di caratteri alfanumerici, ma ciò impedirebbe al database di effettuare controlli formali all'inserimento dei dati e quindi sarebbe possibile inserire, ad esempio, "pippo" nella data di nascita di un soggetto.

Per consentire il controllo formale di tutti i dati Oracle mette a disposizione una serie di tipi di dato tra cui l'utente del database può scegliere per ogni singola informazione da gestire. Scegliere per ogni informazione il tipo di dato più corretto consente un elevato livello di controllo.

I dati si dividono essenzialmente in quattro macro categorie:

- **stringhe alfanumeriche**, ad esempio nomi e cognomi, descrizioni, note;
- **numeri**, ad esempio importi, quantità;
- **date ed orari**;
- **dati binari**, ad esempio foto, video, audio.

Anche i tipi di dato forniti da Oracle ricalcano più o meno questa suddivisione. Li analizzeremo nei prossimi paragrafi.

7.1.1 Dati alfanumerici

I dati alfanumerici possono essere classificati in due sottoinsiemi: dati a lunghezza fissa e dati a lunghezza variabile. I dati a lunghezza fissa sono sicuramente meno frequenti, alcuni esempi possono essere

- Il codice fiscale italiano delle persone fisiche, lungo 16 caratteri.
- Il codice di avviamento postale italiano, lungo 5 caratteri.
- Il codice IBAN dei conti correnti italiani, lungo 27 caratteri.
- Il codice ISBN dei libri, lungo 13 caratteri.

Per la gestione di questo genere di informazioni Oracle mette a disposizioni il tipo di dato **CHAR(N)**, dove N è un numero che rappresenta la lunghezza fissa dell'informazione e può variare da 1 a 2000. Di conseguenza un codice fiscale sarà dichiarato CHAR(16), un IBAN sarà CHAR(27) e così via. In un campo di tipo CHAR(N) saranno contenuti sempre esattamente N caratteri (lettere, cifre, segni di punteggiatura). Se un utente inserisce in un campo CHAR(N) un dato più corto questo viene automaticamente riempito di spazi a destra fino a raggiungere la lunghezza dichiarata. Se si cerca di inserire un dato più lungo di N il database solleva un errore.

La maggior parte delle informazioni alfanumeriche hanno lunghezza variabile, in fase di definizione possiamo ipotizzare solo una lunghezza massima. Ciò vale ad esempio per nomi e cognomi, descrizioni e note libere. Oracle mette a disposizione per questo tipo di informazioni il tipo **VARCHAR2(N)** dove N è un numero che rappresenta la lunghezza massima dell'informazione e può variare da 1 a 4000. In Oracle 12c, seguendo una particolare procedura, è possibile abilitare gli utenti ad utilizzare campi VARCHAR2 lunghi fino a 32767 byte. Un nome potrebbe essere dunque definito VARCHAR2(30) ed una descrizione VARCHAR2(200). Se un utente inserisce in un campo VARCHAR2(N) una stringa alfanumerica più corta di N caratteri Oracle la inserisce così com'è senza apportare alcuna modifica. Se invece si tentasse di inserire più di N caratteri Oracle solleverebbe un errore.

In Oracle esiste anche il tipo di dati VARCHAR(N). Attualmente questo tipo si comporta esattamente nello stesso modo del VARCHAR2(N) ma Oracle ne sconsiglia l'utilizzo perché in future versioni del DB potrebbe applicare a questo tipo di dato una diversa logica di confronto tra stringhe rispetto a quella applicata per il tipo VARCHAR2. Se si definisce un campo di una tabella utilizzando il tipo VARCHAR Oracle lo converte automaticamente in VARCHAR2.

```
WTO> create table A (X VARCHAR(3));
Tabella creata.
WTO> DESC A
 Nome                                      Nullo?   Tipo
 ----------------------------------------- -------- ------------
 X                                                  VARCHAR2(3)
```

Nell'esempio precedente viene creata una tabella A contenente una sola colonna X di tipo VARCHAR(3). Dopo la creazione il comando DESC

mostra che la colonna è stata automaticamente definita di tipo VARCHAR2(3).

Come visto la lunghezza massima di un campo VARCHAR2 è 32767 caratteri. Laddove fosse necessario memorizzare stringhe alfanumeriche più lunghe è possibile definire più campi lunghi ognuno 4000 caratteri oppure utilizzare il tipo **CLOB** che significa Character Long Object e può contenere, in un database standard con data block di 8K, fino a 32TB di dati. I campi CLOB sono soggetti però ad alcune limitazioni:

- Un campo CLOB non può essere chiave primaria nella tabella.
- Non può essere incluso in un cluster di tabelle.
- Non può essere utilizzato nelle clausole ORDER BY e GROUP BY di una SELECT.
- Non può essere utilizzato in una SELECT DISTINCT.
- Non può essere utilizzato in un indice standard.

Queste limitazioni sono ovviamente poco comprensibili in questa fase del corso, ma saranno tutte chiare alla fine.

Prima dell'introduzione del tipo CLOB, avvenuta con Oracle 8, per definire campi alfanumerici più lunghi di 2000 caratteri bisognava utilizzare il tipo **LONG**. Questo tipo di dato può contenere fino a 2GB di caratteri ma è soggetto a forti limitazioni (ad esempio ce ne può essere al massimo uno per tabella). Il tipo LONG è ancora presente per compatibilità con le vecchie versioni del database ma se ne sconsiglia fortemente l'utilizzo.

7.1.2 Numeri

Il tipo di dato principalmente utilizzato in Oracle per la conservazione dei dati numerici è NUMBER(P,S). P, la precisione, è un numero che varia da 1 a 38 e rappresenta il numero di cifre significative complessive, includendo sia la parte intera che la parte decimale del numero. S, la scala, è un numero che varia da -84 a 127 ed indica la massima cifra frazionaria da gestire. Se S assume un valore negativo si intende che non si vogliono gestire cifre frazionarie né le S cifre intere meno significative. Ad esempio un campo definito NUMBER(5,-2) potrà contenere al massimo cinque cifre significative e non conterrà né cifre frazionarie né unità né decine. Sarà quindi un numero sempre arrotondato alle centinaia: 1234500, 400, 98700…

Precisione e scala non devono essere obbligatoriamente indicate. In particolare scrivere NUMBER(P) equivale a scrivere NUMBER(P,0) e rappresenta un numero intero di massimo P cifre. Se non si specificano né P né S si ottiene un numero a virgola mobile (10.5) (tutti i NUMBER(P,S) visti finora sono a virgola fissa) che rappresenta il numero avente la massima precisione e scala gestito in Oracle.

Nella tabella seguente sono elencati alcuni esempi di NUMBER con i relativi valori positivi minimi e massimi.

Tipo	Minimo numero positivo rappresentabile	Massimo numero positivo rappresentabile
NUMBER	10^{-130}	$<10^{126}$
NUMBER(38)	1	$<10^{38}$
NUMBER(5,2)	0,01	999,99
NUMBER(5,-2)	100	9999900
NUMBER(3,8)	0,00000001	0,00000999

7.1.3 Date ed orari

Il tipo di dato **DATE** consente di registrare nel database una data comprensiva di anno, mese, giorno, ore, minuti e secondi. Non è possibile indicare in un capo DATE le frazioni di secondo. L'intervallo di date rappresentabile con un campo di tipo DATE va dal 31/12/4712 Avanti Cristo al 31/12/9999 Dopo Cristo. L'anno 0 non può essere inserito.

A partire dalla versione 9i del database Oracle ha aggiunto il nuovo tipo di dato **TIMESTAMP(N)** dove N è un numero da 1 a 9 che rappresenta la precisione delle frazioni di secondo da gestire. Un campo TIMESTAMP(2) contiene ad esempio anno, mese, giorno, ora, minuti, secondi, decimi e centesimi di secondo.

Nel caso in cui si inserisca una data, o timestamp, priva della componente orario, questa verrà comunque valorizzata utilizzando come default la mezzanotte esatta. Sia in un campo DATE che TIMESTAMP è anche possibile omettere la parte relativa al giorno. Inserendo solo l'orario, ad esempio "15:10:20", Oracle inserirà la data di sistema (oggi) all'orario specificato.

7.1.4 Dati binari

Una stringa binaria è una successione lunga a piacere di bit. Tipicamente sono trattati come stringhe binarie le password cifrate e tutti i contenuti multimediali: foto, video, audio.

Fin dalle prime versioni Oracle mette a disposizione il tipo **RAW(N)**, dove N è un numero che va da 1 a 32767 (in Oracle 12c, prima il massimo era 2000), per memorizzare una stringa di N bit. Questo campo è ovviamente utilizzabile, per un problema di dimensione, per memorizzare una password cifrata ma certo non un contenuto multimediale. Per contenuti binari di maggior dimensione è disponibile il tipo **BLOB**, Binary Long Object, che è identico al CLOB, anche nelle limitazioni, con l'unica differenza di contenere dati binari anziché caratteri.

Come nel caso del LONG, nelle vecchie versioni di Oracle era presente il tipo **LONG RAW** che poteva contenere fino a 2GB di dati binari.

Anche questo tipo è ancora presente per compatibilità con le vecchie versioni ma se ne sconsiglia l'utilizzo.

7.2 Comandi DDL

I comandi DDL servono a creare, modificare ed eliminare gli oggetti presenti nel database.

7.2.1 CREATE

Il comando CREATE si utilizza per definire nuovi oggetti di database. Il comando ovviamente assume una forma diversa per ogni tipo di oggetto da creare. Per ogni tipo di oggetto preso in considerazione nel capitolo 5 sarà fornito un esempio e saranno discusse le possibili varianti.

Quando si crea un oggetto è sempre possibile definire lo schema in cui si desidera che l'oggetto sia inserito anteponendo il nome dello schema al nome dell'oggetto e separando i due nomi con un punto.

Ad esempio in fase di creazione di una tabella l'istruzione comincerà con

```
Create table <nome schema>.<nome tabella> (
```

Il nome dello schema è facoltativo, nel caso in cui venga omesso la tabella sarà creata nello schema dell'utente attualmente connesso.

Regole di nomenclatura

I nomi degli oggetti di database hanno una lunghezza massima di 30 caratteri. Possono essere virgolettati ("quoted", tra doppi apici) oppure non virgolettati. Nel primo caso i nomi sono case-sensitive e possono essere formati da qualunque carattere disponibile. Ovviamente se il nome è virgolettato dovrà sempre essere utilizzato con le virgolette in tutti i comandi SQL:

```
WTO >create table "123TesTVirgo@à++" (
  2  a number);

Tabella creata.

--IL COMANDO SEGUENTE LEGGE L'INTERO CONTENUTO DELLA TABELLA.
WTO >select * from 123TesTVirgo@à++;
select * from 123TesTVirgo@à++
              *
ERRORE alla riga 1:
ORA-00903: invalid table name


WTO >select * from "123TesTVirgo@à++";

Nessuna riga selezionata
```

Eccezione a questa regola è solo il caso in cui il nome virgolettato sia definito in maiuscolo e sia un nome valido anche senza virgolette. In tal caso nei comandi SQL il nome può essere utilizzato anche senza virgolette. Nell'esempio seguente viene creata la tabella "test" e si vede che è

impossibile leggerla utilizzando il nome test senza virgolette. Successivamente viene definita la tabella “TEST”, in questo caso è possibile leggerla utilizzando il nome TEST senza virgolette.

```
WTO >create table "test"  (a number);

Tabella creata.

WTO >select * from test;
select * from test
              *
ERRORE alla riga 1:
ORA-00942: table or view does not exist

WTO >select * from "test";

Nessuna riga selezionata

WTO >create table "TEST"  (a number);

Tabella creata.

WTO >select * from test;

Nessuna riga selezionata
```

I nomi non virgolettati, invece, devono cominciare con una lettera dell’alfabeto, possono contenere solo lettere, cifre ed i caratteri “#”, “_” e “$” e, soprattutto, sono case-insensitive.

Il consiglio, per non dover sempre ricordare se un oggetto è stato definito con lettere maiuscole, minuscole o misto, è utilizzare nomi non virgolettati. Le restrizioni sui nomi sono, normalmente, più che accettabili.

Creazione di una tabella

Il comando di creazione di una tabella assume generalmente la forma semplice

```
create table <nome schema>.<nome tabella> (
  <nome colonna> <tipo colonna> <null/not null>,
  <nome colonna> <tipo colonna> <null/not null>,
  ...
  <nome colonna> <tipo colonna> <null/not null>
);
```

Ad esempio la tabella CLIENTI può essere definita come segue:

```
Create table wto_esempio.CLIENTI (
COD_CLIENTE    NUMBER(8)      NOT NULL,
NOME           VARCHAR2(30)   NOT NULL,
COGNOME        VARCHAR2(30)   NOT NULL,
COD_FISC       CHAR(16)       ,
INDIRIZZO      VARCHAR2(30)   NOT NULL,
CAP            CHAR(5)        NOT NULL,
COMUNE         CHAR(4)        NOT NULL
);
```

Tra parentesi appare l’elenco delle colonne della tabella. Per ogni colonna bisogna obbligatoriamente indicare solo il nome ed il tipo. Facoltativamente può essere indicata l’obbligatorietà della colonna (mediante

le clausole NULL o NOT NULL). L’assenza della clausola di obbligatorietà equivale alla presenza della clausola NULL, quindi il campo non è obbligatorio.

A fronte di ogni colonna è possibile indicare un valore di default, ciò si ottiene semplicemente aggiungendo dopo alla definizione della colonna la clausola DEFAULT seguita da un valore. Nel caso in cui, durante l’inserimento, non venga fornito alcun valore per una colonna munita di valore di default, questa viene automaticamente inizializzata con il valore indicato in fase di creazione. L’argomento sarà approfondito nell’ambito del comando INSERT.

In molti database è possibile indicare che una colonna deve essere automaticamente popolata con una sequenza numerica. In Oracle ciò non è possibile. Esistono, come già detto, appositi oggetti di database che nascono per generare sequenze numeriche, ma queste sequenze non possono essere associate esplicitamente in fase di creazione della tabella ad una colonna. Tale associazione deve essere fatta via programma, in PL/SQL o in qualunque altro programma che possa effettuare operazioni SQL. Ci torneremo quando tratteremo le sequenze ed i trigger PL/SQL.

Nel comando di creazione della tabella è possibile specificare i constraint che si intendono definire sui dati. Nell’esempio che segue viene creata una tabella TEST con le colonne A, B, C, D. La colonna A è definita come primary key, la colonna B come unique key, la colonna C come foreign key verso la colonna C di un’altra tabella TEST2, sulla colonna D viene definito un check constraint: la colonna ammetterà solo valori maggiori di 3.

```
WTO >create table test (
  2  a number primary key,
  3  b number unique,
  4  c number references test2(c),
  5  d number check (d>3)
  6  );

Tabella creata.
```

Siccome le colonne definite come PRIMARY KEY devono essere necessariamente NOT NULL, anche in assenza della clausola di obbligatorietà la colonna A viene definita obbligatoria.

```
WTO >desc test
 Nome                    Nullo?   Tipo
 ----------------------- -------- ------------
 A                       NOT NULL NUMBER
 B                                NUMBER
 C                                NUMBER
 D                                NUMBER
```

Per semplificare le istruzioni SQL da eseguire, la creazione dei constraint è in genere rinviata ad un momento immediatamente successivo alla creazione della tabella. Questa modalità sarà trattata con il comando di ALTER TABLE.

Il comando CREATE TABLE … AS SELECT … consente di creare una tabella specificando, mediante una SELECT, sia la struttura che deve

assumere che i dati che deve contenere. Nell'esempio che segue viene dapprima create una tabella TEST, poi mediante l'istruzione CREATE TABLE … AS SELECT … si crea un duplicato identico della tabella denominato TEST_COPY. Il comando DESC che chiude l'esempio mostra la struttura della tabella TEST_COPY.

```
WTO >create table test (
  2  A number not null,
  3  B varchar2(30),
  4  C date
  5  );

Tabella creata.

WTO >create table test_copy as
  2  select * from test;

Tabella creata.

WTO >desc test_copy
 Nome              Nullo?   Tipo
 ----------------- -------- ------------
 A                 NOT NULL NUMBER
 B                          VARCHAR2(30)
 C                          DATE
```

É importante notare che, oltre a creare la tabella, l'istruzione CREATE TABLE … AS SELECT … esegue anche l'inserimento dei record estratti dalla query.

In fase di creazione della tabella è possibile specificare anche alcuni parametri che ne influenzano l'occupazione fisica dello spazio sul disco. Non si approfondirà il tema perché è sicuramente al di là di un corso introduttivo, in ogni caso è importante fare un esempio per vedere come scegliere il tablespace in cui Oracle deve collocare la tabella.

Basta semplicemente aggiungere all'istruzione la clausola TABLESPACE seguita dal nome del tablespace che si intende utilizzare.

```
WTO >create table test (
  2  A number,
  3  B number)
  4  tablespace USERS;

Tabella creata.
```

In assenza della clausola TABLESPACE viene utilizzato il tablespace definito di default per l'utente.

Creazione di un indice

La forma più semplice dell'istruzione di creazione di un indice è la seguente:

```
create index <nome schema>.<nome indice> ON <nome tabella>(
  <nome colonna>,
  <nome colonna>,
  ...
  <nome colonna>
);
```

Se, ad esempio, si intendono indicizzare le colonne A e B della tabella TEST si utilizzerà il comando:

```
WTO >create index test_idx on test(a,b);

Indice creato.
```

È possibile specificare che l'indice deve essere univoco, cioè le colonne che lo compongono non devono ammettere duplicazione di valori, aggiungendo la clausola UNIQUE dopo la parola chiave CREATE

```
WTO >create unique index test_idx on test(a,b);

Indice creato.
```

Se si vuole creare un indice di tipo bitmap anziché un classico indice B-Tree si utilizzerà la clausola BITMAP:

```
WTO >create bitmap index test_idx on test(a,b);

Indice creato.
```

Un indice di tipo BITMAP non può essere UNIQUE.

Normalmente i valori delle colonne che compongono l'indice vengono archiviati nell'albero in ardine crescente. In alcuni casi può essere utile forzare l'archiviazione dei valori di una o più colonne in ordine decrescente. In tal caso si può aggiungere la clausola DESC subito dopo il nome delle colonne interessate:

```
WTO >create index test_idx on test(a desc, b);

Indice creato.
```

Come detto quando si sono introdotti i cluster di tabelle, la chiave di clusterizzazione deve essere indicizzata. Un indice di questo tipo prende il nome di *cluster index* e si crea in maniera analoga a quanto visto finora ma esplicitando la clausola CLUSTER prima del nome del cluster e tralasciando la specifica delle colonne.

```
WTO >create cluster my_clust (a number);
Creato cluster.

WTO >create index my_clust_idx on cluster my_clust;
Indice creato.
```

Creazione di una IOT

Una IOT, come già detto, è una ordinaria tabella relazionale che viene archiviata in memoria sotto forma di indice, cioè in una struttura di tipo B-Tree. Per creare una IOT si utilizza l'istruzione CREATE TABLE seguita dalla clausola ORGANIZATION INDEX.

Una IOT non può esistere se sulla tabella non è definita la primary key. Nell'esempio che segue si cerca dapprima di creare una IOT senza primary key e poi si modifica l'istruzione aggiungendo la definizione del constraint.

```
WTO >create table test_iot (
  2  A number not null,
  3  B varchar2(30),
  4  C date
  5  )
  6  organization index;
organization index
            *
ERRORE alla riga 6:
ORA-25175: no PRIMARY KEY constraint
found


WTO >create table test_iot (
  2  A number not null primary key,
  3  B varchar2(30),
  4  C date
  5  )
  6  organization index;

Tabella creata.
```

Creazione di un cluster di tabelle

Come già detto un cluster è un insieme di tabelle in cui l'archiviazione dei record nei datafile avviene in maniera differente dal solito: per ogni valore della chiave di clusterizzazione Oracle riserva uno o più blocchi per archiviare insieme i record delle diverse tabelle che hanno quello specifico valore sulla chiave di clusterizzazione.

Primo passo per la creazione del cluster è la definizione del nome del cluster, della chiave di clusterizzazione e della dimensione (in byte) dello spazio da riservare ad ogni valore della chiave di clusterizzazione.

L'istruzione in generale assume la seguente forma

```
Create cluster <nome schema>.<nome cluster> (
<nome colonna> <tipo>,
<nome colonna> <tipo>,
<nome colonna> <tipo>
) SIZE <numero byte>;
```

Nell'esempio seguente viene definito il cluster COMUNI_PROVINCE avente come chiave di clusterizzazione il codice della provincia e come spazio riservato ad ogni valore della chiave (clausola SIZE) due blocchi di database (16384 byte nel caso specifico):

```
WTO >create cluster COMUNI_PROVINCE (
  2  cod_provincia char(2)
  3  ) SIZE 16384;

Creato cluster.
```

Successivamente viene creato il cluster index:

```
WTO >create index COMUNI_PROVINCE_IDX
  2  on cluster COMUNI_PROVINCE;

Indice creato.
```

Poi viene creata la tabella COMUNI ed inserita nel cluster. Rispetto alle CREATE TABLE viste in precedenza risalta la clausola CLUSTER che chiude l'istruzione.

```
WTO >create table COMUNI (
  2  cod_comune char(4) not null primary key,
  3  descrizione varchar2(100),
  4  cod_provincia char(2)
  5  ) cluster COMUNI_PROVINCE (cod_provincia);

Tabella creata.
```

Infine viene creata la tabella PROVINCE nello stesso modo. In questo caso la colonna che funge da chiave di clusterizzazione ha un nome diverso (ID_PRO) da quello usato in precedenza sia in fase di definizione del cluster sia nella tabella COMUNI (COD_PROVINCIA).

```
WTO >create table PROVINCE (
  2  id_pro char(2) not null primary key,
  3  descrizione varchar2(100)
  4  ) cluster COMUNI_PROVINCE (id_pro);

Tabella creata.
```

Questa appena mostrata è la sequenza da utilizzare per creare un indexed cluster.

In caso di hash cluster non serve la definizione dell'indice e si aggiunge la clausola HASHKEYS per indicare quanti valori differenti approssimativamente assumerà la chiave di clusterizzazione:

```
WTO >create cluster COMUNI_PROVINCE (
  2  cod_provincia char(2)
  3  ) SIZE 16384 HASHKEYS 103;

Creato cluster.

WTO >create table COMUNI (
  2  cod_comune char(4) not null primary key,
  3  descrizione varchar2(100),
  4  cod_provincia char(2)
  5  ) cluster COMUNI_PROVINCE (cod_provincia);

Tabella creata.
```

```
WTO >create table PROVINCE (
  2  id_pro char(2) not null primary key,
  3  descrizione varchar2(100)
  4  ) cluster COMUNI_PROVINCE (id_pro);

Tabella creata.
```

Creazione di una vista

Una vista è semplicemente una query SQL salvata nel database. Della query viene salvata la sola definizione, non i dati che essa estrae. Un utente del database percepisce la vista come se fosse effettivamente una tabella, può eseguire il comando DESC per vederne la struttura e utilizzarla in query più complesse insieme ad altre viste e tabelle. Al momento dell'esecuzione di una query che contiene la vista, all'interno dell'istruzione il nome della vista viene sostituito con la query SQL ad essa associata.

L'istruzione è molto semplice:

```
CREATE VIEW <nome schema>.<nome vista> AS
SELECT ...
```

Ad esempio, volendo creare una vista che includa solo il nome e cognome dei clienti residenti a Roma (codice H501) utilizzeremo la seguente istruzione:

```
WTO >desc clienti
 Nome                     Nullo?   Tipo
 ------------------------ -------- ----------------
 COD_CLIENTE              NOT NULL NUMBER(8)
 NOME                     NOT NULL VARCHAR2(30)
 COGNOME                  NOT NULL VARCHAR2(30)
 COD_FISC                          CHAR(16)
 INDIRIZZO                NOT NULL VARCHAR2(30)
 CAP                      NOT NULL CHAR(5)
 COMUNE                   NOT NULL CHAR(4)

WTO >create view clienti_roma as
  2  select nome, cognome
  3  from clienti
  4  where comune='H501';
Vista creata.

WTO >desc clienti_roma
 Nome                     Nullo?   Tipo
 ------------------------ -------- ----------------
 NOME                     NOT NULL VARCHAR2(30)
 COGNOME                  NOT NULL VARCHAR2(30)
```

A questo punto se un utente esegue il comando

```
WTO >select *
  2  from clienti_roma;

NOME       COGNOME
---------- ----------
MARCO      ROSSI
GIOVANNI   BIANCHI
```

Questo viene interpretato da Oracle esattamente come il seguente:

```
WTO >select *
```

```
  2  from (select nome, cognome from clienti where comune='H501');

NOME       COGNOME
---------- ----------
MARCO      ROSSI
GIOVANNI   BIANCHI
```

Ritornando alla creazione della vista, c'è la possibilità di imporre il constraint di CHECK OPTION. Basterà aggiungere a fine istruzione la clausola WITH CHECK OPTION:

```
WTO >create view clienti_roma as
  2  select nome, cognome
  3  from clienti
  4  where comune='H501'
  5  with check option;

Vista creata.
```

In questa vista, a differenza della precedente, non sarà possibile inserire record relativi a clienti residenti in un comune diverso da Roma.

Le viste possono essere create con la clausola OR REPLACE subito dopo la parola chiave CREATE. In questa modalità, se nello stesso schema esiste già un vista avente lo stesso nome di quella che si sta creando, la prima viene eliminata prima di procedere alla creazione della seconda.

Nell'esempio seguente si cerca di creare di nuovo la vista CLIENTI_ROMA senza usare la clausola OR REPLACE e si ottiene un errore di vista già esistente. Successivamente si utilizza la clausola OR REPLACE e l'errore non si ripresenta.

```
WTO >desc clienti_roma
 Nome                  Nullo?   Tipo
 ----------------- -------- ------------
 NOME                  NOT NULL VARCHAR2(30)
 COGNOME               NOT NULL VARCHAR2(30)
```

```
WTO >create view clienti_roma as
  2  select nome, cognome
  3  from clienti
  4  where comune='H501';
create view clienti_roma as
            *
ERRORE alla riga 1:
ORA-00955: name is already used by an
existing object


WTO >create or replace
  2  view clienti_roma as
  3  select nome, cognome
  4  from clienti
  5  where comune='H501';

Vista creata.
```

Creazione di una vista materializzata

Una vista materializzata appare all'utente esattamente come una vista classica. La differenza risiede nel fatto che la vista materializzata contiene effettivamente dei dati e dunque potrebbe essere disallineata rispetto al risultato attuale della query che la definisce. La sintassi di creazione è identica a quella delle viste a meno della parola chiave MATERIALIZED:

```
CREATE MATERIALIZED VIEW <nome schema>.<nome vista> AS
SELECT ...
```

Per fare un esempio utilizziamo la stessa query della vista CLIENTI_ROMA:

```
WTO >create materialized view clienti_roma_mat as
  2  select nome, cognome
  3  from clienti
  4  where comune='H501';

Creata vista materializzata.

WTO >select * from clienti_roma_mat;

NOME       COGNOME
---------- ----------
MARCO      ROSSI
GIOVANNI   BIANCHI
```

La vista materializzata CLIENTI_ROMA_MAT è definita con lo stesso SQL della vista CLIENTI_ROMA, le due viste hanno lo stesso contenuto. Ma in realtà solo la CLIENTI_ROMA_MAT contiene effettivamente i dati, mentre la CLIENTI_ROMA fa solo riferimento ai dati presenti nella tabella CLIENTI. Adesso effettuiamo una modifica ai dati nella tabella CLIENTI. Cambiamo il nome del cliente ROSSI in MASSIMO anziché MARCO e rieseguiamo la query sia dalla vista che dalla vista materializzata:

```
WTO >update clienti
  2  set nome='MASSIMO'
  3  where cognome='ROSSI';

Aggiornata 1 riga.
```

```
WTO >select * from clienti_roma;

NOME       COGNOME
---------- ----------
MASSIMO    ROSSI
GIOVANNI   BIANCHI

WTO >select * from clienti_roma_mat;

NOME       COGNOME
---------- ----------
MARCO      ROSSI
GIOVANNI   BIANCHI
```

I risultati sono differenti perché solo la vista classica, facendo riferimento direttamente alla tabella CLIENTI, riflette automaticamente le modifiche apportate a quella tabella. La vista materializzata necessita di un aggiornamento per recepire le modifiche.

Visto che in fase di creazione della vista materializzata non abbiamo specificato nulla in merito all'aggiornamento automatico, l'unico modo per aggiornare la vista è utilizzare la procedura REFRESH del package di sistema DBMS_MVIEW passandogli come primo parametro il nome della vista e come secondo la lettera C che significa refresh completo:

```
WTO >exec dbms_mview.refresh('CLIENTI_ROMA_MAT','C')

Procedura PL/SQL completata correttamente.

WTO >select * from clienti_roma_mat;

NOME       COGNOME
---------- ----------
MASSIMO    ROSSI
GIOVANNI   BIANCHI
```

Per creare una vista materializzata con aggiornamento automatico si utilizzano le clausole REFRESH, START WITH e NEXT.

REFRESH introduce la parte di istruzione in cui si parla dell'aggiornamento, START WITH indica l'istante di tempo in cui deve avvenire il primo aggiornamento e NEXT il tempo del secondo aggiornamento. Dopo il secondo aggiornamento ci sarà un successivo aggiornamento automatico continuo ogni volta che passa un intervallo di tempo pari a NEXT – START WITH.

Se ad esempio START WITH è pari ad oggi alle 18.30 e NEXT è pari ad oggi alle 19.00, allora l'aggiornamento avverrà alle 18.30, poi alle 19.00 e via così ogni mezz'ora.

Se si omette la clausola START WITH il primo aggiornamento coincide con l'istante di creazione della vista.

Se si omette la clausola NEXT ci sarà un solo aggiornamento.

Se si omette del tutto la clausola REFRESH la vista non sarà mai aggiornata automaticamente.

Nell'esempio seguente viene creata una vista materializzata che ha come START WITH l'istante di creazione più un minuto e come NEXT l'istante di creazione più tre minuti. Di conseguenza la vista materializzata viene aggiornata dopo un minuto dalla creazione e poi ogni due minuti.

La sintassi con cui si esprime l'istante di creazione della vista fa uso della funzione SYSDATE che ritorna l'istante corrente (al secondo). A questa vengono aggiunti i minuti come frazioni di giorni 1/(24*60) è pari alla 1440esima parte di un giorno, cioè un minuto.

```
WTO >create materialized view clienti_roma_mat
  2  refresh start with sysdate+1/(24*60)
  3  next sysdate+3/(24*60)
  4  as
  5  select nome, cognome
  6  from clienti
  7  where comune='H501';

Creata vista materializzata.

WTO >select * from clienti_roma_mat;

NOME       COGNOME
---------- ----------
MARCO      ROSSI
GIOVANNI   BIANCHI
```

La vista materializzata è stata creata ed il cliente ROSSI si chiama MARCO. A questo punto modifichiamo la tabella clienti, immediatamente la vista materializzata resta identica, ma dopo un minuto dalla sua creazione si aggiorna automaticamente e ROSSI diventa MASSIMO.

```
WTO >update clienti
  2  set nome='MASSIMO'
  3  where cognome='ROSSI';

Aggiornata 1 riga.

WTO >commit;

Commit completato.

WTO >select * from clienti_roma_mat;

NOME       COGNOME
---------- ----------
MARCO      ROSSI
GIOVANNI   BIANCHI

--Bisogna aspettare che passi almeno un
--minuto dalla creazione della vista
WTO >select * from clienti_roma_mat

NOME       COGNOME
---------- ----------
MASSIMO    ROSSI
GIOVANNI   BIANCHI
```

Quindi l'aggiornamento automatico funziona. Si noti però l'istruzione COMMIT. Questa istruzione conferma le modifiche fatte, in sua assenza la

vista materializzata non sarebbe stata aggiornata. Ad esempio eseguiamo un'altra modifica del nome, trasformiamo MASSIMO in MASS ed aspettiamo più di due minuti:

```
WTO >update clienti
  2  set nome='MASS'
  3  where cognome='ROSSI';

Aggiornata 1 riga.

--Qui aspettiamo più di due minuti in modo
--che l'aggiornamento automatico abbia luogo.
WTO >select * from clienti_roma_mat;

NOME       COGNOME
---------- ----------
MASSIMO    ROSSI
GIOVANNI   BIANCHI
```

Dopo l'aggiornamento automatico nella vista materializzata c'è sempre MASSIMO. Se adesso confermiamo con una COMMIT le modifiche fatte alla tabella CLIENTI ed aspettiamo che passino un paio di minuti in modo da essere certi che l'aggiornamento automatico abbia avuto luogo:

```
WTO >commit;

Commit completato.

--Qui aspettiamo più di due minuti in modo
--che l'aggiornamento automatico abbia luogo.
WTO >select * from clienti_roma_mat;

NOME       COGNOME
---------- ----------
MASS       ROSSI
GIOVANNI   BIANCHI
```

Dunque le viste materializzate acquisiscono solo le modifiche confermate con un comando di COMMIT. Di questo comando parleremo diffusamente nella sezione dedicata ai comandi TCL.

Creazione di una sequenza

Una sequenza è un oggetto che fornisce un progressivo numerico ad ogni chiamata. È usata normalmente per creare automaticamente i valori della chiave in tabelle che non hanno una chiave primaria naturale. Ad esempio il codice cliente potrebbe essere una colonna di questo tipo.

Il modo più semplice per creare la sequenza è

```
CREATE SEQUENCE <nome schema>.<nome sequenza>;
```

Questo comando crea una sequenza che comincia da uno e si incrementa di uno ad ogni chiamata, senza valore massimo.

```
WTO >create sequence seq_test;

Sequenza creata.
```

Per ottenere un valore dalla sequenze basta utilizzare in SQL o PL/SQL la pseudocolonna NEXTVAL come segue:

```
WTO >select seq_test.nextval from dual;

   NEXTVAL
----------
         1

WTO >select seq_test.nextval from dual;

   NEXTVAL
----------
         2

WTO >select seq_test.nextval from dual;

   NEXTVAL
----------
         3
```

La tabella DUAL è una tabella di sistema che ha una sola riga ed una sola colonna. Torna molto utile in varie situazioni di cui si parlerà nel prosieguo. Il valore corrente della sequenza si ottiene con la pseudocolonna CURRVAL

```
WTO >select seq_test.currval from dual;

   CURRVAL
----------
         3
```

Attenzione: CURRVAL non può essere utilizzata in una sessione di lavoro in cui non sia stata già utilizzata almeno una volta NEXTVAL. Aprendo una nuova sessione di SQL*Plus, ed utilizzando lo stesso comando si ottiene:

```
WTO >select seq_test.currval from dual;
select seq_test.currval from dual
                             *
ERRORE alla riga 1:
ORA-08002: sequence SEQ_TEST.CURRVAL is not yet defined in this session
```

Una sequenza può partire da un valore diverso da uno, può incrementarsi con un intervallo a piacere può essere discendente anziché ascendente, può avere un valore massimo o un minimo, può essere ciclica. Di tutti questi casi si mostrano esempi nel seguito.

La sequenza seguente estrae solo i numeri pari: comincia con 2, si incrementa di due unità ad ogni lettura, arriva al massimo a 50 e poi ricomincia da capo (cicla):

```
WTO >create sequence seq_test_pari
  2  start with 2
  3  increment by 2
  4  maxvalue 50
  5  cycle;

Sequenza creata.

WTO >select seq_test_pari.nextval from dual;

   NEXTVAL
----------
```

```
         2

WTO >select seq_test_pari.nextval from dual;

   NEXTVAL
----------
         4

WTO >select seq_test_pari.nextval from dual;

   NEXTVAL
----------
         6
--ALTRE CHIAMATE PER PORTARE LA SEQUENZA A 46 SONO STATE OMESSE
WTO >select seq_test_pari.nextval from dual

   NEXTVAL
----------
        48

WTO >;

   NEXTVAL
----------
        50

WTO >select seq_test_pari.nextval from dual;

   NEXTVAL
----------
         1

WTO >select seq_test_pari.nextval from dual;

   NEXTVAL
----------
         3
```

Attenzione: nelle sequenze cicliche una volta raggiunto il massimo la sequenza riprende da uno. Dunque al secondo giro di numeri la sequenza dei pari diventa una sequenza di numeri dispari! Per evitare questo problema si può impostare 2 come valore minimo per la sequenza:

```
WTO >create sequence seq_test_pari
  2  start with 2
  3  increment by 2
  4  maxvalue 50
  5  cycle
  6  minvalue 2;

Sequenza creata.

WTO >select seq_test_pari.nextval from dual;

   NEXTVAL
----------
         2

WTO > select seq_test_pari.nextval from dual

   NEXTVAL
```

```
----------
         4
-- DOPO ALTRE CHIAMATE:
WTO > select seq_test_pari.nextval from dual

   NEXTVAL
----------
        48

WTO > select seq_test_pari.nextval from dual

   NEXTVAL
----------
        50

WTO > select seq_test_pari.nextval from dual

   NEXTVAL
----------
         2

WTO > select seq_test_pari.nextval from dual

   NEXTVAL
----------
         4
```

Un esempio di sequenza discendente:

```
WTO >create sequence seq_desc
  2  start with 30
  3  increment by -1
  4  maxvalue 30;

Sequenza creata.

WTO >select seq_desc.nextval from dual;

   NEXTVAL
----------
        30

WTO >select seq_desc.nextval from dual

   NEXTVAL
----------
        29

WTO >select seq_desc.nextval from dual

   NEXTVAL
----------
        28

-- DOPO UN PO' DI CHIAMATE...
WTO >select seq_desc.nextval from dual

   NEXTVAL
----------
         1

WTO >select seq_desc.nextval from dual
```

```
   NEXTVAL
----------
         0

WTO >select seq_desc.nextval from dual

   NEXTVAL
----------
        -1

WTO >select seq_desc.nextval from dual

   NEXTVAL
----------
        -2
```

Poiché la sequenza non ha un MINVALUE continua con i numeri negativi…

In Oracle 12c è stata introdotta la possibilità di avere una sequence che gestisce i propri valori all'interno della sessione e non globalmente.

Per creare una *session sequence* è sufficiente utilizzare la parola chiave SESSION nel comando di creazione:

```
O12c>Create sequence sess_seq session;

Sequenza creata.
```

A questo punto chiamiamo tre volte in NEXTVAL in questa sessione:

```
O12c>Select sess_seq.nextval from dual;

   NEXTVAL
----------
         1

O12c>Select sess_seq.nextval from dual;

   NEXTVAL
----------
         2

O12c>Select sess_seq.nextval from dual;

   NEXTVAL
----------
         3
```

Poi apriamo una nuova sessione e la sequence riparte da uno:

```
O12c>conn corso/corso@corsopdb
Connesso.
O12c>Select sess_seq.nextval from dual;

   NEXTVAL
----------
         1
```

Creazione di un sinonimo

Un sinonimo è solo un nome alternativo che scegliamo di utilizzare per un oggetto del db. La sintassi è banale

```
CREATE SYNONYM <nome schema>.<nome sinonimo>
FOR <nome schema>.<nome oggetto>;
```

Se dunque vogliamo chiamare C la tabella dei CLIENTI possiamo fare:

```
WTO >create synonym C for clienti;

Sinonimo creato.

WTO >desc c
 Nome                     Nullo?   Tipo
 ------------------------ -------- ---------------
 COD_CLIENTE              NOT NULL NUMBER(8)
 NOME                     NOT NULL VARCHAR2(30)
 COGNOME                  NOT NULL VARCHAR2(30)
 COD_FISC                          CHAR(16)
 INDIRIZZO                NOT NULL VARCHAR2(30)
 CAP                      NOT NULL CHAR(5)
 COMUNE                   NOT NULL CHAR(4)
```

Da questo momento in poi scrivere C oppure scrivere CLIENTI è esattamente la stessa cosa.

Come le viste, i sinonimi possono essere creati con la clausola OR REPLACE che elimina il sinonimo eventualmente esistente con lo stesso nome prima di procedere alla creazione.

Un sinonimo può essere pubblico, cioè utilizzabile da qualunque utente connesso al db senza dover specificare il nome dello schema. Per ottenere un sinonimo pubblico basta aggiungere la parola chiave PUBLIC subito dopo la CREATE.

Nell'esempio seguente viene creato il sinonimo pubblico C2 per la tabella CLIENTI. Dopo la creazione l'utente può utilizzare il comando DESC su C2. Connettendosi con l'utente scott, un altro utente del database, questo può utilizzare il sinonimo C2 ma non il sinonimo C che era stato in precedenza creato senza utilizzare la parola chiave PUBLIC.

```
WTO >create public synonym c2 for clienti;

Sinonimo creato.

WTO >desc c2
 Nome               Nullo?   Tipo
 ------------------ -------- ------------
 COD_CLIENTE        NOT NULL NUMBER(8)
 NOME               NOT NULL VARCHAR2(30)
 COGNOME            NOT NULL VARCHAR2(30)
 COD_FISC                    CHAR(16)
 INDIRIZZO          NOT NULL VARCHAR2(30)
 CAP                NOT NULL CHAR(5)
 COMUNE             NOT NULL CHAR(4)

WTO >conn scott/tiger
```

```
Connesso.

WTO >desc c
ERROR:
ORA-04043: object c does not exist


WTO >desc c2
 Nome                Nullo?   Tipo
 ------------------ -------- ------------
 COD_CLIENTE        NOT NULL NUMBER(8)
 NOME               NOT NULL VARCHAR2(30)
 COGNOME            NOT NULL VARCHAR2(30)
 COD_FISC                    CHAR(16)
 INDIRIZZO          NOT NULL VARCHAR2(30)
 CAP                NOT NULL CHAR(5)
 COMUNE             NOT NULL CHAR(4)
```

Creazione di un database link

Un database link consente di leggere i dati presenti in un altro database. La forma più semplice che assume l'istruzione per la creazione di un DB Link è la seguente:

```
CREATE DATABASE LINK <nome schema>.<nome dblink>
CONNECT TO <nome utente> IDENTIFIED BY <password>
USING '<stringa di connessione>';
```

Sostanzialmente è esattamente come se dal database su cui siamo collegati aprissimo una sessione SQL*Plus per collegarci ad un altro db. Abbiamo bisogno che sul db di partenza ci sia un tnsnames.ora in cui è definita una stringa di connessione che sintetizza i dati del database remoto e poi per connetterci utilizziamo semplicemente utente, password e stringa di connessione.

Ipotizziamo che nel db a cui siamo collegati ci sia un tnsnames.ora in cui è definita la stringa di connessione *altro_db* che fornisce le informazioni di connessione ad un altro database Oracle. Ipotizziamo di voler, restando sempre collegati al nostro DB, di voler leggere il contenuto della tabella DEPT che si trova nello schema SCOTT nell'altro database Oracle.

Sul nostro database creiamo un database link:

```
WTO >CREATE DATABASE LINK scott_remoto
  2  CONNECT TO scott IDENTIFIED BY tiger
  3  USING 'altro_db';

Creato database link.
```

Il link si chiama scott_remoto ed al suo interno abbiamo già definito sia i dati che servono per raggiungere il database remoto (mediante la stringa di connessione altro_db) sia utente e password da usare per la connessione.

Per leggere la tabella DEPT che si trova sullo schema SCOTT del database remoto dobbiamo solo aggiungere @scott_remoto dopo il nome della tabella:

```
WTO >select * from dept@scott_remoto;

    DEPTNO DNAME          LOC
```

```
---------- ------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

Nella medesima istruzione SQL è possibile leggere contemporaneamente i dati da tabelle locali e da tabelle remote.

7.2.2 ALTER

Il comando ALTER consente di modificare alcuni dei parametri degli oggetti già creati senza modificarne il contenuto. Nel seguito sono discusse ed esemplificate alcune delle attività per cui, più comunemente, si ricorre al comando ALTER.

Aggiungere, modificare, rinominare, eliminare le colonne di una tabella

Per aggiungere colonne ad una tabella si utilizza il comando

```
ALTER TABLE <nome schema>.<nome tabella> ADD (
  <nome colonna> <tipo colonna> <null/not null>,
  <nome colonna> <tipo colonna> <null/not null>,
  ...
  <nome colonna> <tipo colonna> <null/not null>
);
```

Ovviamente non è possibile aggiungere una colonna NOT NULL se in tabella sono già presenti dei record. Per tali record, infatti, la nuova colonna non sarebbe valorizzata e violerebbe il constraint NOT NULL.

Come esempio partendo dalla tabella TEST definita come segue:

```
WTO >create table test (
  2  A number not null,
  3  B varchar2(30),
  4  C date
  5  );

Tabella creata.

WTO >desc test
 Nome                Nullo?   Tipo
 ----------------- -------- ------------
 A                  NOT NULL NUMBER
 B                           VARCHAR2(30)
 C                           DATE
```

Si aggiungono le colonne D, un numerico, ed E, una data.

```
WTO >alter table test add (
  2  D number,
  3  E date);

Tabella modificata.

WTO >desc test
 Nome                Nullo?   Tipo
 ----------------- -------- ------------
 A                  NOT NULL NUMBER
 B                           VARCHAR2(30)
```

```
 C                          DATE
 D                          NUMBER
 E                          DATE
```

Le nuove colonne vengono aggiunte in coda, non è possibile aggiungerle in altre posizioni.

La modifica delle colonne già presenti in tabella si realizza con la clausola MODIFY.

```
ALTER TABLE <nome schema>.<nome tabella> MODIFY (
  <nome colonna> <tipo colonna> <null/not null>,
  <nome colonna> <tipo colonna> <null/not null>,
  ...
  <nome colonna> <tipo colonna> <null/not null>
);
```

Ad esempio, nella tabella TEST, si decide di trasformare la colonna C in un campo numerico obbligatorio e la colonna D in una data:

```
WTO >alter table test modify (
  2  c number not null,
  3  d date
  4  );

Tabella modificata.

WTO >desc test
 Nome              Nullo?   Tipo
 ----------------- -------- ------------
 A                 NOT NULL NUMBER
 B                          VARCHAR2(30)
 C                 NOT NULL NUMBER
 D                          DATE
 E                          DATE
```

Chiaramente la modifica del tipo della colonna può essere effettuata solo se in tale colonna non esistono dati, cioè se la tabella è vuota oppure tutti i record presenti in tabella non hanno valorizzata la colonna che si intende modificare.

Se si modifica solo la dimensione massima del campo senza cambiarne il tipo, ad esempio trasformando un VARCHAR2(30) in VARCHAR2(10) o viceversa, si ottiene errore solo quando la dimensione viene diminuita ed esistono nella colonna che si sta modificando dati che eccedono la nuova lunghezza. Ad esempio ipotizziamo che nella colonna B della tabella TEST la stringa più lunga presente sia lunga due caratteri. Possiamo tranquillamente modificare la colonna di VARCHAR2(30) a VARCHAR2(10), ma se cerchiamo di modificare la colonna a VARCHAR2(1) otteniamo un errore.

```
WTO >alter table test modify b varchar2(10);

Tabella modificata.

WTO >alter table test modify b varchar2(1);
alter table test modify b varchar2(1)
                        *
ERRORE alla riga 1:
ORA-01441: cannot decrease column
```

```
length because some value is too big
```

La modifica dell’obbligatorietà da NULL a NOT NULL può essere eseguita solo se la tabella è vuota, mentre da NOT NULL a NULL solo se la colonna non fa parte della PRIMARY KEY della tabella.

Per rinominare una colonna si può utilizzare il comando

```
ALTER TABLE <nome schema>.<nome tabella> RENAME
COLUMN <nome colonna> to <nuovo nome colonna>
```

Ad esempio:

```
WTO >desc test
 Nome                Nullo?   Tipo
 ----------------- -------- ------------
 A                  NOT NULL NUMBER
 B                           VARCHAR2(10)
 C                  NOT NULL NUMBER
 D                           NUMBER
 E                           DATE

WTO >alter table test rename column A to NUOVA_A;

Tabella modificata.

WTO >desc test
 Nome                Nullo?   Tipo
 ----------------- -------- ------------
 NUOVA_A            NOT NULL NUMBER
 B                           VARCHAR2(10)
 C                  NOT NULL NUMBER
 D                           NUMBER
 E                           DATE
```

È possibile eliminare una colonna mediante la clausola DROP:

```
ALTER TABLE <nome schema>.<nome tabella> DROP COLUMN <nome colonna>
```

Ad esempio:

```
WTO >alter table test drop column d;

Tabella modificata.

WTO >desc test
 Nome                Nullo?   Tipo
 ----------------- -------- ------------
 NUOVA_A            NOT NULL NUMBER
 B                           VARCHAR2(10)
 C                  NOT NULL NUMBER
 E                           DATE
```

Chiaramente tutti i dati presenti in D prima dell’esecuzione del comando andranno persi.

Tornando all’aggiunta di colonne, se si ha la necessità di aggiungere una nuova colonna in una posizione specifica, anziché come ultima colonna, si può procedere con l’eliminazione delle come nell’esempio seguente, dove si inserisce la colonna X prima della colonna E:

```
WTO >desc test
 Nome                Nullo?   Tipo
 ----------------- -------- ------------
```

```
 NUOVA_A           NOT NULL NUMBER
 B                          VARCHAR2(10)
 C                 NOT NULL NUMBER
 E                          DATE

WTO >alter table test drop column e;

Tabella modificata.

WTO >alter table test add (X number, e date);

Tabella modificata.

WTO >desc test;
 Nome              Nullo?   Tipo
 ----------------- -------- ------------
 NUOVA_A           NOT NULL NUMBER
 B                          VARCHAR2(10)
 C                 NOT NULL NUMBER
 X                          NUMBER
 E                          DATE
```

Ovviamente tale procedimento causa la perdita dei dati presenti in colonna E, dunque può essere eseguito a tabella vuota oppure dopo avere salvato opportunamente i dati presenti in tabella in modo da poterli recuperare dopo la modifica strutturale. Nell'esempio seguente si utilizzano il comando CREATE…AS SELECT, per salvare i dati prima della modifica, ed il comando INSERT…SELECT, per ripristinare i dati dopo la modifica.

```
--IL COMANDO SEGUENTE CREA UNA NUOVA TABELLA IDENTICA A TEST
WTO >create table test_pre_modifica as select * from test;

Tabella creata.

--IL COMANDO SEGUENTE ELIMINA LA COLONNA E, PERDENDONE I DATI
WTO >alter table test drop column e;

Tabella modificata.

--IL COMANDO SEGUENTE AGGIUNGE LE COLONNE Y ed E
WTO >alter table test add (Y number, e date);

Tabella modificata.

--IL COMANDO SEGUENTE ELIMINA TUTTI I DATI PRESENTI IN TEST
WTO >truncate table test;

Tabella troncata.

--IL COMANDO SEGUENTE POPOLA NUOVAMENTE LA TABELLA TEST CON I -----DATI
CHE ERANO STATI SALVATI IN TEST_PRE_MODIFICA
--LA NUOVA COLONNA Y NON È INCLUSA NEL COMANDO PERCHÈ ESSENDO
--NUOVA NON DEVE ESSERE POPOLATA.
WTO >insert into test (NUOVA_A, B, C, X, E)
  2  select * from test_pre_modifica;

Creata 1 riga.

WTO >desc test;
 Nome              Nullo?   Tipo
 ----------------- -------- ------------
```

```
 NUOVA_A            NOT NULL NUMBER
 B                           VARCHAR2(10)
 C                  NOT NULL NUMBER
 X                           NUMBER
 Y                           NUMBER
 E                           DATE
```

Il procedimento appena mostrato potrebbe non essere utilizzabile in caso di presenza di FOREIGN KEY nel db che impediscono l'eliminazione temporanea di tutti i dati presenti nella tabella TEST. In tal caso bisogna anche disabilitare tali constraint prima di procedere con l'attività e poi ripristinarli al termine della modifica.

Colonne invisibili

In Oracle 12c l'utente può decidere che una colonna debba essere invisibile. Ciò significa che la colonna non sarà estratta da una SELECT * o dal commando DESCRIBE.

Le colonne invisibili vengono regolarmente estratte quando sono esplicitamente menzionate nella select list di una query.

Partiamo dalla tabella TEST_INVISIBILI così fatta:

```
O12c>desc test_invisibili;
 Nome                    Nullo?   Tipo
 ----------------------- -------- -----------
 A                       NOT NULL NUMBER
 B                                VARCHAR2(30)

O12c>select * from test_invisibili;

         A B
---------- ------------------------------
         1 sabato
         2 domenica
         3 lunedì
         4 martedì
         5 mercoledì
         6 giovedì
         7 venerdì

7 righe selezionate.
```

Rendiamo invisibile la colonna B:

```
O12c>alter table test_invisibili modify b invisible;

Tabella modificata.

O12c>desc test_invisibili
 Nome                 Nullo?   Tipo
 ----------------- -------- ------------
 A                  NOT NULL NUMBER

O12c>select * from test_invisibili;

         A
----------
         1
         2
         3
```

```
         4
         5
         6
         7
7 righe selezionate.
```

La colonna B è comunque sempre al suo posto, bisogna solo esplicitarla in una select per vederla:

```
O12c>select a,b from test_invisibili;

         A B
---------- --------------------
         1 sabato
         2 domenica
         3 lunedì
         4 martedì
         5 mercoledì
         6 giovedì
         7 venerdì
7 righe selezionate.
```

Per riportare tutto alla norma:

```
O12c>alter table test_invisibili modify b visible;

Tabella modificata.

O12c>select * from test_invisibili;

         A B
---------- --------------------
         1 sabato
         2 domenica
         3 lunedì
         4 martedì
         5 mercoledì
         6 giovedì
         7 venerdì

7 righe selezionate.
```

Aggiungere, abilitare e disabilitare, rinominare, eliminare i constraint di una tabella

L'aggiunta di un constraint ad una tabella segue lo schema generale

```
ALTER TABLE <nome schema>.<nome tabella> ADD
CONSTRAINT <nome constraint>
<tipo constraint> <specifica constraint>
```

dove il *tipo constraint* può essere PRIMARY KEY, FOREIGN KEY, UNIQUE oppure CHECK e la *specifica constraint* dipende dal tipo come segue:

Tipo constraint	**Specifica constraint**

PRIMARY KEY	`(`
	`<nome colonna>,`
	`<nome colonna>,`
	`...`
	`<nome colonna>`
	`)`

FOREIGN KEY	(<nome colonna>, <nome colonna>, ... <nome colonna>) REFERENCES <nome tabella> (<nome colonna>, <nome colonna>, ... <nome colonna>)
UNIQUE	(<nome colonna>, <nome colonna>, ... <nome colonna>)
CHECK	(<regola>)

Quattro esempi per chiarire:

Aggiunta della PRIMARY KEY

```
WTO >ALTER TABLE TEST ADD
  2  CONSTRAINT TEST_PK PRIMARY KEY (NUOVA_A);

Tabella modificata.
```

La colonna NUOVA_A è PRIMARY KEY.

Aggiunta di una FOREIGN KEY

```
--PRIMA VIENE CREATA LA TABELLA A CUI
--LA FOREIGN KEY DEVE FARE RIFERIMENTO
WTO >create table test2 (chiave number primary key);

Tabella creata.

WTO >alter table test add
  2  constraint test_fk foreign key (c) references test2(chiave);

Tabella modificata.
```

La colonna C della tabella TEST fa riferimento alla colonna CHIAVE della tabella TEST2.

Aggiunta di una UNIQUE KEY

```
WTO >alter table test add constraint test_uk
  2  unique (x);

Tabella modificata.
```

La colonna X non ammette valori duplicati.

Aggiunta di un CHECK constraint

```
WTO >alter table test add
  2  constraint test_ck check (Y>3);

Tabella modificata.
```

La colonna Y ammette solo valori maggiori di tre.

Non è possibile modificare la definizione di un constraint. Per cambiarne lo stato (abilitato/disabilitato) si utilizza la clausola MODIFY ENABLE/DISABLE. Nell'esempio seguente si crea una tabella TEST_DUP contenente una sola colonna X. X è primary key della tabella e quindi univoca. Il tentativo di inserire due valori uguali causa chiaramente un errore. A questo punto il constraint viene disabilitato e si procede senza errori all'inserimento di valori duplicati. Quando si cerca di abilitare il constraint si ottiene un errore, successivamente i valori duplicati vengono rimossi ed il constraint viene correttamente riabilitato.

```
--CREAZIONE DELLA TABELLA
WTO >create table test_dup (x number);

Tabella creata.

--CREAZIONE DELLA PRIMARY KEY
WTO >alter table test_dup add constraint x_pk primary key (x);

Tabella modificata.

WTO >desc test_dup
 Nome                 Nullo?   Tipo
 ------------------ -------- ------------
 X                  NOT NULL NUMBER

--INSERIMENTO DEL VALORE 1
WTO >insert into test_dup values (1);

Creata 1 riga.

--NUOVO INSERIMENTO DEL VALORE 1 VA IN ERRORE
WTO >insert into test_dup values (1);
insert into test_dup values (1)
*
ERRORE alla riga 1:
ORA-00001: unique constraint
(WTO_ESEMPIO.X_PK) violated

--DISABILITAZIONE DELLA PRIMARY KEY
WTO >alter table test_dup modify constraint x_pk disable;

Tabella modificata.

--NUOVO INSERIMENTO DEL VALORE 1 VA BENE
WTO >insert into test_dup values (1);

Creata 1 riga.

--SELEZIONE DI TUTTI I DATI PRESENTI IN
--TABELLA IL VALORE 1 È PRESENTE DUE VOLTE.
WTO >select * from test_dup;

         X
----------
         1
         1

--ABILITAZIONE DEL CONSTRAINT.
--ERRORE PERCHÈ CI SONO VALORI DUPLICATI
WTO >alter table test_dup modify constraint x_pk enable;
```

```
alter table test_dup modify constraint x_pk enable
*
ERRORE alla riga 1:
ORA-02437: cannot validate
(WTO_ESEMPIO.X_PK) - primary key
violated

--CANCELLAZIONE DI UN SOLO RECORD DALLA TABELLA
WTO >delete test_dup where rownum=1;

Eliminata 1 riga.

--INFATTI ADESSO C'E' UNA SOLA RIGA
WTO >select * from test_dup;

         X
----------
         1

--ABILITAZIONE DEL CONSTRAINT.
--QUESTA VOLTA VA A BUON FINE.
WTO >alter table test_dup modify constraint x_pk enable;
Tabella modificata.
```

Per rinominare un constraint basta utilizzare il comando

```
ALTER TABLE <nome schema>.<nome tabella>
RENAME CONSTRAINT <nome constraint> TO <nuovo nome constraint>
```

Ad esempio

```
WTO >alter table TEST_DUP rename constraint X_PK to NUOVO_X_PK;
Tabella modificata.
```

Per eliminare un constraint basta utilizzare il comando

```
ALTER TABLE <nome schema>.<nome tabella>
DROP CONSTRAINT <nome constraint>
```

Ad esempio

```
WTO >alter table TEST_DUP drop constraint NUOVO_X_PK;
Tabella modificata.
```

Modificare un cluster di tabelle

Non è possibile modificare la chiave di clusterizzazione di un cluster di tabelle. Il parametro SIZE può essere modificato solo per gli INDEXED CLUSTER e non per gli HASH CLUSTER.

```
--CREAZIONE DI UN HASH CLUSTER
WTO >create cluster COMUNI_PROVINCE (
  2  cod_provincia char(2)
  3  ) SIZE 16384 HASHKEYS 103;

Creato cluster.

--TENTATIVO DI MODIFICARNE LA SIZE.
--ORACLE SOLLEVA UN ERRORE
WTO >alter cluster comuni_province SIZE 8192;
alter cluster comuni_province SIZE 8192
*
ERRORE alla riga 1:
ORA-02466: The SIZE and INITRANS
options cannot be altered for HASH CLUSTERS.
```

```
--CREAZIONE DI UN INDEXED CLUSTER
WTO >create cluster COMUNI_PROVINCE_2 (
  2  cod_provincia char(2)
  3  ) SIZE 16384;

Creato cluster.

--TENTATIVO DI MODIFICARNE LA SIZE.
--TUTTO OK.
WTO >alter cluster comuni_province_2 SIZE 8192;

Modificato cluster.
```

Modificare un indice

Nessuno dei parametri degli indici analizzati in precedenza può essere modificato con l'ALTER INDEX. Il comando seguente può essere utilizzato per rinominare l'indice.

```
WTO >create index test_idx on test (c,x);

Indice creato.

WTO >alter index test_idx rename to test_idx_nuovo;

Modificato indice.
```

Modificare una IOT

Una IOT è una normale tabella ed in quanto tale le si applicano tutte le opzioni di ALTER già viste sopra. L'organizzazione di una tabella non può essere modificata con il comando ALTER. Non è possibile passare da una tabella classica ad una IOT o viceversa con questo comando. Per realizzare tale attività si può utilizzare il package di sistema DBMS_REDEFINITION. Si tratta di un'attività di livello avanzato che non sarà approfondita in questo corso.

Modificare una vista

Nessuno dei parametri delle viste analizzati in precedenza può essere modificato con il comando ALTER VIEW. Per ridefinire una vista si utilizza la parola chiave OR REPLACE dopo la CREATE nel comando di creazione come già visto sopra.

Modificare una vista materializzata

La clausola REFRESH può essere modificata a piacere per cambiare la configurazione dell'aggiornamento automatico. Ad esempio l'istruzione seguente imposta l'aggiornamento automatico in modo che sia eseguito ogni giorno a mezzanotte.

```
WTO >alter materialized view clienti_roma_mat
  2  refresh start with trunc(sysdate+1)
  3  next trunc(sysdate+2);

Modificata vista materializzata.
```

Modificare un sinonimo

Non esiste il comando ALTER SYNONYM, si utilizza la clausola OR REPLACE dopo la CREATE.

Modificare una sequenza

Il comando ALTER SEQUENCE (senza parole chiave aggiuntive) consente di modificare tutti i parametri già descritti in fase di creazione delle sequenze ad eccezione del valore di partenza, che era stato definito mediante la clausola START WITH. Ad esempio nell'istruzione che segue si modifica la sequenza SEQ_TEST, originariamente impostata per fornire un progressivo numerico crescente senza buchi, per ottenere un progressivo di numeri che si incrementi di tre unità ad ogni chiamata.

```
WTO >select seq_test.nextval from dual;
   NEXTVAL
----------
         5

WTO >select seq_test.nextval from dual;

   NEXTVAL
----------
         6

WTO >alter sequence seq_test
  2  increment by 3;

Sequenza modificata.

WTO >select seq_test.nextval from dual;

   NEXTVAL
----------
         9
```

Modificare un database link

Nel comando ALTER DATABASE LINK è possibile specificare la clausola CONNECT esattamente come in fase di creazione del db link, senza altre parole chiave. Nell'esempio seguente si cambia la password utilizzata per accedere al database remoto.

```
WTO >ALTER DATABASE LINK scott_remoto
  2  CONNECT TO scott IDENTIFIED BY newpwd
  3  USING 'altro_db';

Modificato database link.
```

7.2.3 RENAME

Il comando RANAME consente di rinominare un oggetto di database. Questo comando si applica solo a tabelle, viste, sequenze e sinonimi privati. La sintassi è banale.

```
RENAME <nome schema>.<vecchio nome> to <nome schema>.<nuovo nome>
```

Di seguito gli esempi d'uso:

```
--UNA TABELLA
```

```
WTO >rename test to new_test;

Tabella rinominata.

--UNA VISTA
WTO >rename clienti_roma to new_clienti_roma;

Tabella rinominata.

--UNA SEQUENZA
WTO >rename seq_test to seq_test_new;

Tabella rinominata.

--UN SINONIMO PRIVATO
WTO >rename C to C_new;

Tabella rinominata.
```

Il messaggio di feedback è fuorviante perché parla sempre di “tabella rinominata” anche quando l’oggetto rinominato non è una tabella.

7.2.4 DROP

Il comando generico per eliminare un oggetto di database è il seguente:

```
DROP <tipo oggetto> <nome schema>.<nome oggetto>
```

Di seguito alcuni esempi:

```
--UNA TABELLA
WTO >drop table NEW_TEST;

Tabella eliminata.

--UN INDICE
WTO >drop index TEST_IDX;

Indice eliminato.

--UNA VISTA
WTO >drop view CLIENTI_ROMA;

Vista eliminata.

--UNA VISTA MATERIALIZZATA
WTO >drop materialized view CLIENTI_ROMA_MAT;

Vista materializzata eliminata.

--UN CLUSTER VUOTO
WTO >drop cluster COMUNI_PROVINCE;

Eliminato cluster.

--UNA IOT
WTO >drop table TEST_IOT;

Tabella eliminata.
```

```
--UN SINONIMO PRIVATO
WTO >drop synonym C_NEW;

Sinonimo eliminato.

--UN SINONIMO PUBBLICO
WTO >drop public synonym C2;

Sinonimo eliminato.

--UN DATABASE LINK
WTO >drop database link scott_remoto;

Eliminato database link.
```

In aggiunta a questo comando generale, valido per tutti gli oggetti del database, si possono utilizzare alcune clausole specifiche per alcuni tipi di oggetto.

Eliminazione di una tabella referenziata in una foreign key

Nel caso di eliminazione di una tabella, può succedere che essa sia referenziata da un constraint di foreign key definito su un'altra tabella. Negli esempi precedenti si era definita, ad esempio, una foreign key tra la tabella TEST e la tabella TEST2.

In questa situazione non è possibile eliminare la tabella TEST2 con il comando standard.

```
WTO >drop table test2;

drop table test2
           *
ERRORE alla riga 1:
ORA-02449: unique/primary keys in table referenced by foreign keys
```

Per forzare l'eliminazione della tabella referenziata bisogna aggiungere la clausola CASCADE CONSTRAINTS.

```
WTO >drop table test2 cascade constraints;

Tabella eliminata.
```

Oppure, ovviamente, eliminare prima il constraint e poi la tabella.

```
WTO >drop table test2;
drop table test2
           *
ERRORE alla riga 1:
ORA-02449: unique/primary keys in table referenced by foreign keys


WTO >alter table test drop constraint test_fk;

Tabella modificata.

WTO >drop table test2;

Tabella eliminata.
```

Eliminazione di un cluster non vuoto

Se in un cluster di tabelle è stata aggiunta almeno una tabella non sarà possibile eliminarlo.

```
WTO >drop cluster COMUNI_PROVINCE;
drop cluster COMUNI_PROVINCE
*
ERRORE alla riga 1:
ORA-00951: cluster not empty
```

Per superare quest'ostacolo è possibile utilizzare la clausola INCLUDING TABLES

```
WTO >drop cluster COMUNI_PROVINCE including tables;

Eliminato cluster.
```

Oppure eliminare prima le tabelle e poi il cluster.

```
WTO >drop cluster COMUNI_PRVINCE;
drop cluster COMUNI_PRVINCE
*
ERRORE alla riga 1:
ORA-00951: cluster not empty


WTO >drop table comuni;

Tabella eliminata.

WTO >drop table province;

Tabella eliminata.

WTO >drop cluster COMUNI_PRVINCE;

Eliminato cluster.
```

7.2.5 TRUNCATE

Il comando TRUNCATE consente di eliminare in modo rapido l'intero contenuto di una tabella o di un cluster di tabelle senza modificarne la struttura. Il comando non è annullabile ma è automaticamente confermato. Il comando TRUNCATE è generalmente più rapido del comando DML DELETE che sarà illustrato più avanti.

Troncamento di una tabella

L'istruzione di troncamento di una tabella è

```
TRUNCATE TABLE <nome schema>.<nome tabella>
```

Ad esempio:

```
WTO >truncate table test;

Tabella troncata.
```

Troncamento di un cluster di tabelle

L'istruzione di troncamento di un cluster di tabelle è

```
TRUNCATE CLUSTER <nome schema>.<nome cluster>
```

Un HASH cluster non può essere troncato.

```
--CREO UN HASH CLUSTER...
WTO >create cluster COMUNI_PROVINCE (
  2  cod_provincia char(2)
  3  ) SIZE 16384 HASHKEYS 103;

Creato cluster.

--...E NON POSSO TRONCARLO
WTO >truncate cluster COMUNI_PROVINCE;
truncate cluster COMUNI_PROVINCE
                 *
ERRORE alla riga 1:
ORA-03293: Cluster to be truncated is a HASH CLUSTER


WTO >drop cluster COMUNI_PROVINCE;

Eliminato cluster.

--CREO UN INDEXED CLUSTER...
WTO >create cluster COMUNI_PROVINCE (
  2  cod_provincia char(2)
  3  ) SIZE 16384;

Creato cluster.

--...E POSSO TRONCARLO
WTO >truncate cluster COMUNI_PROVINCE;

Cluster troncato.
```

In Oracle 12c è possibile troncare una tabella andando in cascata su tutte le tabelle figlie.

Questa truncate in cascata si realizza mediante la parola chiave CASCADE e richiede l'esistenza di una foreign key definita ON DELETE CASCADE.

Creiamo la tabella padre:

```
O12c>create table padre (id number primary key);

Tabella creata.

O12c>insert into padre values (1);

Creata 1 riga.
```

Poi la figlia:

```
O12c>create table figlio (id number, id_pa number);

Tabella creata.

O12c>insert into figlio values (1,1);

Creata 1 riga.
```

Aggiungiamo la foreign key:

```
O12c>alter table figlio add constraint
  2  pa_fi foreign key (id_pa) references padre(id);

Tabella modificata.
```

Proviamo a troncare il padre, errore:

```
O12c>truncate table padre;
truncate table padre
               *
ERRORE alla riga 1:
ORA-02266: La tabella referenziata da chiavi esterne abilitate dispone
di chiavi uniche/primarie
```

Proviamo a troncare il padre con l'opzione CASCADE, ancora errore:

```
O12c>truncate table padre cascade;
truncate table padre cascade
               *
ERRORE alla riga 1:
ORA-14705: chiavi esterne abilitate fanno riferimento a chiavi univoche
o primarie nella tabella "CORSO"."FIGLIO"
```

Perché la foreign key non era ON DELETE CASCADE. Ridefiniamola:

```
O12c>alter table figlio drop constraint pa_fi;

Tabella modificata.

O12c>alter table figlio add constraint
  2  pa_fi foreign key (id_pa) references padre(id)
  3  on delete cascade;

Tabella modificata.
```

E poi rifacciamo la truncate, errore:

```
O12c>truncate table padre;
truncate table padre
               *
ERRORE alla riga 1:
ORA-02266: La tabella referenziata da chiavi esterne abilitate dispone
di chiavi uniche/primarie
```

Ma con la clausola CASCADE questa volta ci riusciamo:

```
O12c>truncate table padre cascade;
Tabella troncata.

O12c>select * from padre;
Nessuna riga selezionata

O12c>select * from figlio;
Nessuna riga selezionata
```

7.2.6 PURGE

Quando si elimina un oggetto segment Oracle lo conserva nel cestino, come quando si cancella un file in windows.

L'oggetto viene rinominato con un nome di sistema e conservato.

L'effetto delle cancellazioni eseguite negli esempi precedenti è il seguente:

```
WTO >select * from cat;

TABLE_NAME                     TABLE_TYPE
------------------------------ -----------
TEST_COPY                      TABLE
COMUNI                         TABLE
SEQ_TEST_PARI                  SEQUENCE
X                              SEQUENCE
SEQ_DESC                       SEQUENCE
CLIENTI                        TABLE
SEQ_CLIENTI                    SEQUENCE
PRODOTTI                       TABLE
ORDINI                         TABLE
SEQ_ORDINI                     SEQUENCE
PRODOTTI_ORDINATI              TABLE
FATTURE                        TABLE
SEQ_FATTURE                    SEQUENCE
TEST_PRE_MODIFICA              TABLE
TEST_DUP                       TABLE
123TesTVirgo@à++               TABLE
TEST                           TABLE
NEW_CLIENTI_ROMA               VIEW
SEQ_TEST_NEW                   SEQUENCE
BIN$HzDBX3YoS3m9RX5c+xg2/w==$0 TABLE
BIN$H30bMxKuQput74HcB1GXcA==$0 TABLE
BIN$d8Hp3XYlS1G0xuBp0rYIhA==$0 TABLE
BIN$Sw7y9bgbQLSkwfEM+SLAnA==$0 TABLE

Selezionate 23 righe.
```

Le quattro tabelle poste alla fine dell'elenco sono appunto oggetti finiti nel cestino a fronte di comandi DROP.

Un elenco più preciso degli oggetti presenti nel cestino può essere ottenuto con il comando che si vede nell'esempio seguente.

Le informazioni sono estratte dalla tabella di dizionario USER_RECYCLEBIN.

```
WTO >select OBJECT_NAME, original_name, type
  2  from user_recyclebin;

OBJECT_NAME                    ORIGINAL_NAME      TYPE
------------------------------ --------------------------------
BIN$212cBxw4R1C02fElJrct4A==$0 SYS_C0047891       INDEX
BIN$d8Hp3XYlS1G0xuBp0rYIhA==$0 TEST2              TABLE
BIN$HLqiwXS6TeaxkQaIxSJm8w==$0 TEST_IDX_NUOVO     INDEX
BIN$mnplvDa8R6COBKMxs5z5lQ==$0 TEST_PK            INDEX
BIN$3xs4UCdPSdCmQc2JHIvJVA==$0 TEST_UK            INDEX
BIN$Sw7y9bgbQLSkwfEM+SLAnA==$0 NEW_TEST           TABLE
BIN$H30bMxKuQput74HcB1GXcA==$0 TEST2              TABLE
BIN$+6dYH6AjTkajFsq9sw370w==$0 SYS_C0047978       INDEX
```

Per svuotare il cestino parzialmente o totalmente e recuperare definitivamente lo spazio si utilizza il comando PURGE.

In particolare il comando

```
PURGE TABLE <nome tabella>
```

Elimina una tabella dal cestino.

Il comando

```
PURGE INDEX <nome indice>
```

Elimina un indice dal cestino.

Il comando

```
PURGE RECYCLEBIN
```

Svuota il cestino. Di seguito alcuni esempi:

```
WTO >purge table TEST2;

Svuotata tabella.

WTO >purge index TEST_IDX_NUOVO;

Svuotato indice.

WTO >purge recyclebin;

Svuotato cestino.

WTO >select OBJECT_NAME, original_name, type
  2  from user_recyclebin;

Nessuna riga selezionata
```

7.2.7 FLASHBACK TABLE

Visto cos'è il cestino e come può essere svuotato bisogna capire come si possa "riportare in vita" una tabella precedentemente eliminata. Si utilizza il comando

```
FLASHBACK TABLE <nome schema>.<nome tbella> TO BEFORE DROP
```

Nell'esempio seguente viene create una tabella, inserito un dato, eliminata la tabella e, finalmente, ripescata dal cestino.

```
-- CREAZIONE DELLA TABELLA
WTO >create table TEST_FLASHBACK (a number);
Tabella creata.

--INSERIMENTO DI UN RECORD
WTO >insert into TEST_FLASHBACK values (123);

Creata 1 riga.

--VISUALIZZAZIONE DEI DATI IN ESSA CONTENUTI
WTO >select * from TEST_FLASHBACK;

         A
----------
       123
```

```
--ELIMINAZIONE DELLA TABELLA
WTO >drop table TEST_FLASHBACK;

Tabella eliminata.

--VISUALIZZAZIONE DEL CONTENUTO DEL CESTINO
WTO >select OBJECT_NAME, original_name, type
  2  from user_recyclebin;

OBJECT_NAME                    ORIGINAL_NAME            TYPE
------------------------------ ------------------------ ---------
BIN$td/Hy7kQTguCDcGazvTB+w==$0 TEST_FLASHBACK           TABLE

--RECUPERO DELLA TABELLA DAL CESTINO
WTO >flashback table TEST_FLASHBACK to before drop;

Completato flashback.

--VISUALIZZAZIONE DEI DATI IN ESSA CONTENUTI
WTO >select * from TEST_FLASHBACK;

         A
----------
       123

--IL CESTINO E' VUOTO
WTO >select OBJECT_NAME, original_name, type
  2  from user_recyclebin;

Nessuna riga selezionata
```

7.3 Comandi DML

I comandi DML consentono di inserire, modificare e cancellare i dati presenti nelle tabelle senza cambiarne la struttura.

7.3.1 Valori fissi

Durante l'esecuzione dei comandi DML è spesso necessario utilizzare valori fissi specifici (in inglese *literals*) per i dati. La sintassi con cui tali valori vengono specificati dipende dal tipo di dato.

Per indicare una stringa alfanumerica costante si utilizza l'apice singolo prima e dopo la stringa ad esempio

```
'Esempio di stringa'
```

Nel caso in cui la stringa contenga a sua volta un apice singolo questo deve essere raddoppiato:

```
'Quest''esempio è OK.'
```

I valori numerici si esprimono utilizzando le cifre ed eventualmente il solo punto decimale, non la virgola perché questa assume, nei comandi SQL, il ruolo di separatore dei valori negli elenchi. Nessun carattere può essere utilizzato come separatore delle migliaia. Due esempi corretti sono

```
1234.4
```

e

```
133
```

Una data può essere indicata con la parola chiave DATE seguita da un apice singolo, dalla data nel formato anno-mese-giorno e da un altro apice singolo, come nell'esempio seguente

```
DATE'2011-01-28'
```

Con la parola chiave DATE non è possibile specificare ore, minuti e secondi. Utilizzando la parola chiave TIMESTAMP, invece, è possibile arrivare ai secondi ed anche scendere fino al dettaglio delle frazioni di secondo

```
TIMESTAMP'2011-01-28 23:59:59'
```

```
TIMESTAMP'2011-01-28 23:59:59.999'
```

7.3.2 INSERT

Il comando INSERT consente di inserire record in una tabella. Il comando può essere utilizzato nella forma più semplice seguente:

```
INSERT INTO <nome schema>.<nome tabella>
VALUES (<valore>, <valore>, ..., <valore>);
```

Dove la clausola VALUES è utilizzate per specificare, separati da virgole, i valori da assegnare ai diversi campi della riga che si sta inserendo. Per cominciare con gli esempi eliminiamo la tabella TEST già utilizzata

```
WTO >drop table test;

Tabella eliminata.
```

E la definiamo nuovamente come segue:

```
WTO >create table test (
  2  a number,
  3  b date,
  4  c varchar2(30)
  5  );

Tabella creata.
```

Come primo esempio inseriamo in TEST una riga contenente il numero 123,4 nella colonna A; la data 13 gennaio 2011 nella colonna B e la dicitura “Stringa d'esempio” nella C.

```
WTO >insert into test
  2  values (123.4, date'2011-01-13', 'Stringa d''esempio');

Creata 1 riga.

WTO >select * from test;

         A B         C
---------- --------- ------------------------------
     123,4 13-GEN-11 Stringa d'esempio
```

Ovviamente tale sintassi ci costringe a fornire valori per tutte le colonne della tabella. Se, infatti, forniamo solo due valori anziché i tre previsti otteniamo un errore:

```
WTO >insert into test values (123.4, date'2011-01-13');
insert into test values (123.4, date'2011-01-13')
            *
ERRORE alla riga 1:
ORA-00947: not enough values
```

È possibile ovviare a tale situazione specificando nella clausola INTO, dopo il nome della tabella, le colonne che si intendono valorizzare. Nell'esempio che segue si dichiara di voler valorizzare solo le colonne A e B e dunque si forniscono solo due valori:

```
WTO >insert into test (A, B)
  2  values (123.4, date'2011-01-13');

Creata 1 riga.

WTO >select * from test;

         A B         C
---------- --------- ------------------------------
     123,4 13-GEN-11 Stringa d'esempio
     123,4 13-GEN-11
```

Le colonne possono essere indicate anche in un ordine differente da quello previsto in tabella:

```
WTO >insert into test (C,A) values ('Test',0);

Creata 1 riga.

WTO >select * from test;

         A B         C
---------- --------- ------------------------------
     123,4 13-GEN-11 Stringa d'esempio
     123,4 13-GEN-11
         0           Test
```

Anche se una colonna è indicata nella clausola INTO la si può non valorizzare usando la parola chiave NULL.

```
WTO >insert into test (A, B, C)
  2  values (null, null,'Test con campi nulli');

Creata 1 riga.

WTO >select * from test;

         A B         C
---------- --------- ------------------------------
     123,4 13-GEN-11 Stringa d'esempio
     123,4 13-GEN-11
         0           Test
                     Test con campi nulli
```

Nell'esempio presedente, sebbene tutte le tre colonne fossero inserite nella clausola INTO, si è scelto di valorizzare la sola colonna C.

Inserimento dati e valori di default

Vediamo cosa succede quando una colonna è munita di valore di default. Definiamo la tabella TEST_DEFAULT avente due colonne, A è un numero, B è una stringa con valore di default.

```
WTO >create table test_default (
  2  a number,
  3  b varchar2(30) default 'Valore di default'
  4  );

Tabella creata.
```

Inserendo in tabella due valori specifici si ottiene un record in cui ovviamente il valore di default è ignorato:

```
WTO >insert into test_default (A, B)
  2  values (1, 'Test');

Creata 1 riga.

WTO >select * from test_default;

         A B
---------- ------------------------------
         1 Test
```

Per utilizzare il valore di default si può escluderà dalla clausola INTO la colonna relativa:

```
WTO >insert into test_default (A)
  2  values (2);

Creata 1 riga.

WTO >select * from test_default;

         A B
---------- ------------------------------
         1 Test
         2 Valore di default
```

Se invece la colonna munita di default è inclusa nella clausola INTO e viene valorizzata con NULL, essa assumerà il valore NULL:

```
WTO >insert into test_default (A,B)
  2  values (3,null);

Creata 1 riga.

WTO >select * from test_default;

         A B
---------- ------------------------------
         1 Test
         2 Valore di default
         3
```

Se si desidera che una colonna munita di valore di default sia anche menzionata nella clausola INTO ed assuma il valore di default bisogna usare la parola chiave DEFAULT al poste del valore:

```
WTO >insert into test_default (A,B)
  2  values (4, default);

Creata 1 riga.

WTO >select * from test_default;

         A B
---------- ------------------------------
         1 Test
         2 Valore di default
         3
         4 Valore di default
```

Fino ad Oracle 11g, il valore di default di una colonna non è mai stato applicato quando l’utente (o l’applicazione) inseriva esplicitamente il valore NULL.

```
SQL >create table test_default (
  2  a number,
  3  b varchar2(30) default 'Valore di default'
  4  );

Tabella creata.

SQL >insert into test_default (A,B)
  2  values (3,null);
Creata 1 riga.

SQL >select * from test_default;

         A B
---------- ------------------------------
         3
```

Oracle 12c continua a comportarsi in questo modo, a meno che non si faccia uso della clausola ON NULL quando si assegna il default. In questo caso, infatti, si indica ad Oracle di utilizzare il valore di default anche se un utente esplicitamente assegna il valore NULL alla colonna:

```
O12c>create table test_default (
  2  a number,
  3  b varchar2(30) default on null 'Valore di default'
  4  );

Tabella creata.

O12c>insert into test_default (A,B)
  2  values (3,null);
Creata 1 riga.

O12c>select * from test_default;

         A B
---------- ------------------------------
         3 Valore di default
```

A partire da Oracle 12c, quando una colonna non è valorizzata, il suo eventuale valore di default è conservato nel data dictionary e non nella tabella stessa, come invece avveniva nelle precedenti versioni di Oracle.

L'aggiunta di nuove colonne munite di default è quindi molto più veloce, visto che non richiede di scrivere fisicamente il valore di default in tutti i record già presenti in tabella.

Partiamo dalla tabella TEST_TAB valorizzata come segue:

```
O12c>select * from test_tab;

        ID DESCR
---------- ------------------------------
         1 riga 1
         2 riga 2
         3 riga 3
         4 riga 4
         5 riga 5
         6 riga 6
         7 riga 7
         8 riga 8
         9 riga 9
        10 riga 10
        11 riga 11
        12 riga 12
        13 riga 13
        14 riga 14
        15 riga 15
        16 riga 16
        17 riga 17
        18 riga 18
        19 riga 19
        20 riga 20

20 righe selezionate.
```

Aggiungiamo una terza colonna avente default 'ABC':

```
O12c>alter table test_tab add
  2  (X varchar2(3) default 'ABC');

Tabella modificata.

O12c>select * from test_tab;

        ID DESCR                          X
---------- ------------------------------ ---
         1 riga 1                         ABC
         2 riga 2                         ABC
         3 riga 3                         ABC
         4 riga 4                         ABC
         5 riga 5                         ABC
         6 riga 6                         ABC
         7 riga 7                         ABC
         8 riga 8                         ABC
         9 riga 9                         ABC
        10 riga 10                        ABC
        11 riga 11                        ABC
        12 riga 12                        ABC
        13 riga 13                        ABC
```

```
        14 riga 14                     ABC
        15 riga 15                     ABC
        16 riga 16                     ABC
        17 riga 17                     ABC
        18 riga 18                     ABC
        19 riga 19                     ABC
        20 riga 20                     ABC

20 righe selezionate.
```

Il valore compare accanto a tutti i record, ma in realtà è memorizzato nel dizionario. Ciò non causa gli effetti collaterali che si potrebbero temere. Modificando un valore della colonna X e poi modificando il valore di default, oppure annullando più valori della colonna X e poi modificando il valore di default, la colonna X resta sempre correttamente valorizzata come al popolamento:

```
O12c>update test_tab set X='XYZ' where id=17;

Aggiornata 1 riga.

O12c>select * from test_ tab;

        ID DESCR                          X
---------- ------------------------------ ---
         1 riga 1                         ABC
         2 riga 2                         ABC
         3 riga 3                         ABC
         4 riga 4                         ABC
         5 riga 5                         ABC
         6 riga 6                         ABC
         7 riga 7                         ABC
         8 riga 8                         ABC
         9 riga 9                         ABC
        10 riga 10                        ABC
        11 riga 11                        ABC
        12 riga 12                        ABC
        13 riga 13                        ABC
        14 riga 14                        ABC
        15 riga 15                        ABC
        16 riga 16                        ABC
        17 riga 17                        XYZ
        18 riga 18                        ABC
        19 riga 19                        ABC
        20 riga 20                        ABC

20 righe selezionate.

O12c>alter table test_tab modify X default 'DFG';
Tabella modificata.

O12c>select * from test_tab;

        ID DESCR                          X
---------- ------------------------------ ---
         1 riga 1                         ABC
         2 riga 2                         ABC
         3 riga 3                         ABC
         4 riga 4                         ABC
```

```
         5 riga 5                         ABC
         6 riga 6                         ABC
         7 riga 7                         ABC
         8 riga 8                         ABC
         9 riga 9                         ABC
        10 riga 10                        ABC
        11 riga 11                        ABC
        12 riga 12                        ABC
        13 riga 13                        ABC
        14 riga 14                        ABC
        15 riga 15                        ABC
        16 riga 16                        ABC
        17 riga 17                        XYZ
        18 riga 18                        ABC
        19 riga 19                        ABC
        20 riga 20                        ABC
20 righe selezionate.

O12c>update test_tab set X=null where id<7;
6 righe aggiornate.
O12c>select * from test_tab;

        ID DESCR                          X
---------- ------------------------------ ---
         1 riga 1
         2 riga 2
         3 riga 3
         4 riga 4
         5 riga 5
         6 riga 6
         7 riga 7                         ABC
         8 riga 8                         ABC
         9 riga 9                         ABC
        10 riga 10                        ABC
        11 riga 11                        ABC
        12 riga 12                        ABC
        13 riga 13                        ABC
        14 riga 14                        ABC
        15 riga 15                        ABC
        16 riga 16                        ABC
        17 riga 17                        XYZ
        18 riga 18                        ABC
        19 riga 19                        ABC
        20 riga 20                        ABC

20 righe selezionate.

O12c>alter table test_tab modify X default 'HJK';
Tabella modificata.

O12c>select * from test_tab;

        ID DESCR                          X
---------- ------------------------------ ---
         1 riga 1
         2 riga 2
         3 riga 3
         4 riga 4
         5 riga 5
         6 riga 6
         7 riga 7                         ABC
         8 riga 8                         ABC
```

```
         9 riga 9                        ABC
        10 riga 10                       ABC
        11 riga 11                       ABC
        12 riga 12                       ABC
        13 riga 13                       ABC
        14 riga 14                       ABC
        15 riga 15                       ABC
        16 riga 16                       ABC
        17 riga 17                       XYZ
        18 riga 18                       ABC
        19 riga 19                       ABC
        20 riga 20                       ABC

20 righe selezionate.
```

Il valore di default di una Colonna in Oracle 12c può far riferimento direttamente ad una sequence Oracle. Può essere utilizzato sia il CURRVAL che il NEXTVAL.

L'accoppiamento tra una colonna ed una sequence, ampiamente diffuso negli altri database relazionali, è sempre stato escluso in Oracle. Per utilizzare automaticamente una sequence in fase di inserimento era necessario scrivere un db trigger o diffondere il codice nelle applicazioni. Con Oracle 12c finalmente questo limite viene superato. Esempio di utilizzo.

Innanzi tutto creo una sequence:

```
O12c>create sequence test_seq;

Sequenza creata.
```

Poi una tabella in cui il valore di default di un campo (ID) fa riferimento al nextval della sequence:

```
O12c>create table test_seq_tab
  2  (id number default test_seq.nextval,
  3  descr varchar2(30));

Tabella creata.
```

Successivamente inserisco 20 righe in tabella, valorizzando solo la descrizione del record, non l'ID:

```
O12c>insert into test_seq_tab (descr)
  2  select 'riga '||level
  3  from dual
  4  connect by level <= 20;

20 righe create.
```

Con una query verifico che la sequence è stata utilizzata per popolare l'ID:

```
O12c>select * from test_seq_tab;

        ID DESCR
---------- ------------------------------
         1 riga 1
         2 riga 2
```

```
         3 riga 3
         4 riga 4
         5 riga 5
         6 riga 6
         7 riga 7
         8 riga 8
         9 riga 9
        10 riga 10
        11 riga 11
        12 riga 12
        13 riga 13
        14 riga 14
        15 riga 15
        16 riga 16
        17 riga 17
        18 riga 18
        19 riga 19
        20 riga 20

20 righe selezionate.
```

Infatti se leggo il CURRVAL della sequence:

```
O12c>select test_seq.currval from dual;

   CURRVAL
----------
        20
```

In Oracle 12c le colonne di una tabella possono essere definite come IDENTITY. Con questa modifica Oracle si adegua allo standard SQL ANSI.

Le colonne IDENTITY assumono automaticamente un valore progressivo che le rende univoche.

Vediamo un esempio. Prima di tutto creo una tabella che ha una colonna definita come GENERATED AS IDENTITY:

```
O12c>create table test_identity (
  2  a number generated as identity,
  3  b varchar2(30)
  4  );

Tabella creata.
```

Poi inserisco in tabella sette righe, valorizzando solo la colonna B con il nome di un giorno della settimana:

```
O12c>insert into test_identity (b)
  2  select to_char(sysdate+level,'day')
  3  from dual
  4  connect by level<=7;

7 righe create.
```

Effettivamente la colonna A è autogenerata come sequenza:

```
O12c>select * from test_identity;
```

```
         A B
---------- ------------------------------
         1 sabato
         2 domenica
         3 lunedý
         4 martedý
         5 mercoledý
         6 giovedý
         7 venerdý

7 righe selezionate.
```

Quando si definisce una colonna IDENTITY, Oracle essenzialmente definisce una sequence e la associa alla colonna come visto precedentemente.

Oracle infatti ha creato una sequence:

```
O12c>select sequence_name from user_sequences;

SEQUENCE_NAME
---------------------------------------------------
ISEQ$$_92470
TEST_SEQ
```

Che è arrivata al valore sette:

```
O12c>select ISEQ$$_92470.currval from dual;

   CURRVAL
----------
         7
```

Inserimento di record multipli

Con i comandi appena visti è possibile inserire in tabella un solo record per volta. Per inserire in tabella più record ottenuti con una query si utilizza, invece, il comando

```
INSERT INTO <nome schema>.<nome tabella>
SELECT ...;
```

L'istruzione SELECT sarà approfondita nel seguito. Gli esempi che vediamo utilizzano alcune SELECT davvero banali. Cominciamo creando la tabella TEST2 munita anch'essa di tre colonne: D, E ed F.

```
WTO >create table test2 (
  2  d date,
  3  e number,
  4  f varchar2(30)
  5  );

Tabella creata.
```

Nel primo esempio copiamo tutte le righe di TEST in TEST2. Attenzione: poiché la colonna A è numerica come la E, la colonna B è una data come la D e la colonna C è una stringa come la F bisogna far corrispondere per posizione queste coppie:

```
WTO >insert into test2 (E, D, F)
```

```
  2  select * from test;

Create 4 righe.
WTO >select * from test2;

D                   E F
--------- ---------- ------------------------------
13-GEN-11      123,4 Stringa d'esempio
13-GEN-11      123,4
                   0 Test
                     Test con campi nulli
```

Il comando SELECT * FROM TEST estrae le colonne nell'ordine in cui sono definite in tabella: prima A, poi B ed infine C. Quindi nella INTO le colonne di TEST2 devono apparire nell'ordine: E, D ed F.

In alternativa è possibile non menzionare le colonne di TEST2 nella INTO ma bisogna estrarre le colonne di TEST nell'ordine B, A, C.

```
WTO >insert into test2
  2  select b, a, c from test;

Create 4 righe.

WTO >select * from test2;

D                   E F
--------- ---------- ------------------------------
13-GEN-11      123,4 Stringa d'esempio
13-GEN-11      123,4
                   0 Test
                     Test con campi nulli
13-GEN-11      123,4 Stringa d'esempio
13-GEN-11      123,4
                   0 Test
                     Test con campi nulli

Selezionate 8 righe.
```

Definiamo adesso la tabella TEST3 munita delle colonne G, H ed I.

```
WTO >create table test3 (
  2  g number,
  3  h date,
  4  i varchar2(30)
  5  );
Tabella creata.
```

L'opzione ALL del comando INSERT consente di inserire record in diverse tabelle con una singola istruzione:

```
WTO >insert all
  2  into test2(d, e, f)
  3  values (b, a, c)
  4  into test3(g, h, i)
  5  values (a, b, c)
  6  select * from test;
Create 8 righe.

WTO >select * from test2;
```

```
D                  E F
--------- ---------- ------------------------------
13-GEN-11      123,4 Stringa d'esempio
13-GEN-11      123,4
                   0 Test
                     Test con campi nulli
13-GEN-11      123,4 Stringa d'esempio
13-GEN-11      123,4
                   0 Test
                     Test con campi nulli
13-GEN-11      123,4 Stringa d'esempio
13-GEN-11      123,4
                   0 Test
                     Test con campi nulli
Selezionate 12 righe.

WTO >select * from test3;

         G H         I
---------- --------- ------------------------------
     123,4 13-GEN-11 Stringa d'esempio
     123,4 13-GEN-11
         0           Test
                     Test con campi nulli
```

Nell'esempio precedente sono state specificate, dopo la parola chiave ALL, due diverse clausole INTO. La prima per inserire i record in tabella TEST2, la seconda per inserirli in tabella TEST3.

Siccome la SELECT estrae quattro record, vengono inserite in tutto otto righe, quattro in TEST2 e quattro in TEST3.

Lo stesso comando può essere utilizzato per inserire righe diverse nella stessa tabella.

Facciamo prima di tutto un po' di pulizia in TEST3 mediante il comando DELETE, che approfondiremo nei prossimi paragrafi.

```
WTO >delete test3;
Eliminate 4 righe.

WTO >select * from test;

         A B         C
---------- --------- ------------------------------
     123,4 13-GEN-11 Stringa d'esempio
     123,4 13-GEN-11
         0           Test
                     Test con campi nulli
```

Poi eseguiamo il seguente comando.

```
WTO >insert all
  2  into test3 (g,h,i)
  3  values (a+1,b+1,c)
  4  into test3 (g,h,i)
  5  values (a+2,b+2,c)
  6  select * from test;
Create 8 righe.

WTO >select * from test3;
```

```
         G H         I
---------- --------- ------------------------------
     124,4 14-GEN-11 Stringa d'esempio
     124,4 14-GEN-11
         1           Test
                     Test con campi nulli
     125,4 15-GEN-11 Stringa d'esempio
     125,4 15-GEN-11
         2           Test
                     Test con campi nulli

Selezionate 8 righe.
```

Questo comando legge quattro righe da TEST. Con la prima clausola INTO le inseriamo in TEST3 aggiungendo però una unità ai valori di A ed un giorno ai valori di B. Con la seconda clausola INTO le inseriamo aggiungendo 2 unità ai valori di A e due giorni ai valori di B.

In conclusione abbiamo otto record in TEST3.

Inserimento mediante una vista

Come detto quando è stato descritto il constraint “check option” è possibile inserire record in una tabella attraverso una vista. Facciamo proprio l’esempio che era stato accennato per spiegare il comportamento del constraint.

Per prima cosa creiamo la tabella PERSONE ed inseriamo alcuni record:

```
WTO >create table persone (
  2  nome varchar2(30),
  3  data_nascita date,
  4  sesso char(1)
  5  );

Tabella creata.

WTO >insert into persone values
  2  ('Massimo', date'1971-11-01','M');

Creata 1 riga.

WTO >insert into persone values ('Laura', date'2000-03-09','F');

Creata 1 riga.

WTO >insert into persone values ('Marco', date'1990-08-23','M');

Creata 1 riga.

WTO >insert into persone values ('Clara', date'1995-04-18','F');

Creata 1 riga.

WTO >select * from persone;

NOME       DATA_NASC S
---------- --------- -
Massimo    01-NOV-71 M
Laura      09-MAR-00 F
```

```
Marco      23-AGO-90 M
Clara      18-APR-95 F
```

Successivamente definiamo la vista MASCHI.

```
WTO >create view maschi as
  2  select * from persone
  3  where sesso='M';

Vista creata.

WTO >select * from maschi;

NOME       DATA_NASC S
---------- --------- -
Massimo    01-NOV-71 M
Marco      23-AGO-90 M
```

In questo primo momento non abbiamo definito il constraint “check option”, di conseguenza sarà possibile inserire un record “femmina” in tabella attraverso la vista MASCHI.

```
WTO >insert into maschi values ('Veronica',date'1980-02-26','F');

Creata 1 riga.

WTO >select * from maschi;

NOME       DATA_NASC S
---------- --------- -
Massimo    01-NOV-71 M
Marco      23-AGO-90 M

WTO >select * from persone;

NOME       DATA_NASC S
---------- --------- -
Massimo    01-NOV-71 M
Laura      09-MAR-00 F
Marco      23-AGO-90 M
Clara      18-APR-95 F
Veronica   26-FEB-80 F
```

Se invece ridefiniamo la vista MASCHI aggiungendo la clausola WITH CHECK OPTION otterremo un comportamento differente.

```
WTO >create or replace view maschi
  2  as select * from persone
  3  where sesso='M'
  4  with check option;

Vista creata.

WTO >insert into maschi values ('Vittoria',date'1987-05-10','F');
insert into maschi values ('Vittoria',date'1987-05-10','F')
            *
ERRORE alla riga 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Non è possibile inserire un record "femmina" in tabella attraverso la vista MASCHI. È ovviamente sempre possibile inserire direttamente in tabella.

```
WTO >insert into persone values
  2  ('Vittoria',date'1987-05-10','F');

Creata 1 riga.

WTO >select * from persone;

NOME       DATA_NASC S
---------- --------- -
Massimo    01-NOV-71 M
Laura      09-MAR-00 F
Marco      23-AGO-90 M
Clara      18-APR-95 F
Veronica   26-FEB-80 F
Vittoria   10-MAG-87 F

Selezionate 6 righe.
```

7.3.3 UPDATE

Il comando UPDATE consente di modificare i dati presenti in una tabella. La sintassi del comando è la seguente:

```
UPDATE <nome schema>.<nome tabella>
SET <nome colonna> = <valore>,
    <nome colonna> = <valore>,
...
    <nome colonna> = <valore>
WHERE <condizione>
;
```

La condizione da mettere, opzionalmente, dopo la parola chiave WHERE serve ad identificare le righe da aggiornare. In assenza di tale condizione tutte le righe presenti in tabella vengono aggiornate.

Per ogni riga della tabella la condizione nel suo complesso assume il valore VERO oppure il valore FALSO. Vengono aggiornate solo le righe per le quali la condizione vale VERO.

Qualche semplice esempio.

Nell'esempio seguente tutte le righe della tabella PERSONE vengono aggiornate. La colonna DATA_NASCITA assume il valore 12 gennaio 2000. La clausola WHERE è omessa.

```
WTO >select * from persone;

NOME       DATA_NASC S
---------- --------- -
Massimo    01-NOV-71 M
Laura      09-MAR-00 F
Marco      23-AGO-90 M
Clara      18-APR-95 F
Veronica   26-FEB-80 F
Vittoria   10-MAG-87 F

Selezionate 6 righe.
```

```
WTO >update persone
  2  set data_nascita = date'2000-01-12';
Aggiornate 6 righe.
WTO >select * from persone;

NOME       DATA_NASC S
---------- --------- -
Massimo    12-GEN-00 M
Laura      12-GEN-00 F
Marco      12-GEN-00 M
Clara      12-GEN-00 F
Veronica   12-GEN-00 F
Vittoria   12-GEN-00 F

Selezionate 6 righe.
```

Nell'esempio seguente solo la riga relativa alla persona avente NOME='Vittoria' viene aggiornata.

```
WTO >update persone
  2  set data_nascita = date'1987-05-10'
  3  where nome='Vittoria';

Aggiornata 1 riga.

WTO >select * from persone;

NOME       DATA_NASC S
---------- --------- -
Massimo    12-GEN-00 M
Laura      12-GEN-00 F
Marco      12-GEN-00 M
Clara      12-GEN-00 F
Veronica   12-GEN-00 F
Vittoria   10-MAG-87 F

Selezionate 6 righe.
```

Le condizioni saranno approfondite quando si parlerà del comando SELECT.

Per determinare il valore da assegnare ad una colonna in fase di aggiornamento si può fare riferimento anche al valore già presente in tabella. Se, ad esempio si intende aumentare di un giorno la data di nascita di tutti i record, si può scrivere:

```
WTO >update persone
  2  set data_nascita=data_nascita+1;

Aggiornate 6 righe.

WTO >select * from persone;

NOME       DATA_NASC S
---------- --------- -
Massimo    13-GEN-00 M
Laura      13-GEN-00 F
Marco      13-GEN-00 M
Clara      13-GEN-00 F
Veronica   13-GEN-00 F
Vittoria   11-MAG-87 F
```

```
Selezionate 6 righe.
```

Il valore da assegnare alle colonne può essere ottenuto anche con espressioni complesse che coinvolgono i valori di altre colonne dello stesso record modificati e combinati utilizzando funzioni PL/SQL.

Tali capacità di manipolazione, utili anche per ottenere condizioni complesse da inserire nella clausola WHERE, saranno approfondite nel seguito quando si descriverà il comando SELECT.

7.3.4 DELETE

Il comando per cancellare righe da una tabella assume la forma

```
DELETE FROM <nome schema>.<nome tabella>
WHERE <condizione>
;
```

La parola chiave FROM è stata inserita perché prevista nello standard SQL ISO ma in Oracle non è obbligatoria e frequentemente viene omessa.

La clausola WHERE ha lo stesso comportamento già visto nell'UPDATE, serve ad identificare le righe da eliminare.

Ad esempio cancelliamo dalla tabella PERSONE tutte quelle di sesso maschile.

```
WTO >select * from persone;

NOME       DATA_NASC S
---------- --------- -
Massimo    13-GEN-00 M
Laura      13-GEN-00 F
Marco      13-GEN-00 M
Clara      13-GEN-00 F
Veronica   13-GEN-00 F
Vittoria   11-MAG-87 F

Selezionate 6 righe.

WTO >delete persone
  2  where sesso='M';

Eliminate 2 righe.

WTO >select * from persone;

NOME       DATA_NASC S
---------- --------- -
Laura      13-GEN-00 F
Clara      13-GEN-00 F
Veronica   13-GEN-00 F
Vittoria   11-MAG-87 F
```

Omettendo la clausola WHERE eliminiamo tutte le righe presenti in tabella:

```
WTO >delete persone;
Eliminate 4 righe.
```

```
WTO >select * from persone;
Nessuna riga selezionata
```

7.3.5 MERGE

L’istruzione MERGE consente di eseguire in un unico comando un UPDATE ed un INSERT. È possibile infatti specificare una condizione e, a seconda del fatto che questa condizione sia vera o falsa, inserire un nuovo record in tabella oppure aggiornarne uno già esistente.

Come esempio d’uso si ipotizzi di avere la necessità di dover aggiornare una tabella periodicamente leggendo i dati da un'altra tabella.

Si ipotizzi anche che i dati nella tabella sorgente siano completi, cioè che ci siano sia i record che abbiamo già caricato con inserimenti precedenti (e magari ora vanno aggiornati) sia nuovi record.

L'approccio più semplice è sicuramente quello di svuotare la tabella ad ogni aggiornamento e riempirla rileggendo tutti i dati. Non sempre, però, è l'approccio migliore. Se invece vogliamo inserire i nuovi record ed aggiornare soltanto quelli già presenti possiamo utilizzare l'istruzione MERGE, che in pratica racchiude la possibilità di eseguire, a seconda dei record, un INSERT oppure un UPDATE.

Vediamo un esempio. Nello schema SCOTT ipotizziamo di dover aggiornare la lista dei dipendenti (una nuova tabella MYEMP strutturalmente identica alla EMP) leggendo la tabella EMP.

```
WTO> select * from emp;

EMPNO ENAME   JOB         MGR HIREDATE    SAL   COMM  DEPTNO
----- ------- --------- ---- --------- ----- ------ -------
 7369 SMITH   CLERK     7902 17-DIC-80   800            20
 7499 ALLEN   SALESMAN  7698 20-FEB-81  1600    300     30
 7521 WARD    SALESMAN  7698 22-FEB-81  1250    500     30
 7566 JONES   MANAGER   7839 02-APR-81  2975            20
 7654 MARTIN  SALESMAN  7698 28-SET-81  1250   1400     30
 7698 BLAKE   MANAGER   7839 01-MAG-81  2850            30
 7782 CLARK   MANAGER   7839 09-GIU-81  2450            10
 7788 SCOTT   ANALYST   7566 09-DIC-82  3000            20
 7839 KING    PRESIDENT      17-NOV-81  5000            10
 7844 TURNER  SALESMAN  7698 08-SET-81  1500      0     30
 7876 ADAMS   CLERK     7788 12-GEN-83  1100            20
 7900 JAMES   CLERK     7698 03-DIC-81   950            30
 7902 FORD    ANALYST   7566 03-DIC-81  3000            20
 7934 MILLER  CLERK     7782 23-GEN-82  1300            10
Selezionate 14 righe.

WTO> select * from myemp;

EMPNO ENAME   JOB         MGR HIREDATE    SAL   COMM  DEPTNO
----- ------- --------- ---- --------- ----- ------ -------
 7566 JONES   MANAGER   7839 02-APR-81  1000            20
 7654 MARTIN  SALESMAN  7698 28-SET-81  1000   1400     30
 7698 BLAKE   MANAGER   7839 01-MAG-81  1000            30
 7782 CLARK   MANAGER   7839 09-GIU-81  1000            10
 7788 SCOTT   ANALYST   7566 09-DIC-82  3000            20
 7839 KING    PRESIDENT      17-NOV-81  5000            10
```

```
 7844 TURNER   SALESMAN   7698 08-SET-81  1500      0      30
 7876 ADAMS    CLERK      7788 12-GEN-83  1100             20
 7900 JAMES    CLERK      7698 03-DIC-81   950             30
 7902 FORD     ANALYST    7566 03-DIC-81  3000             20
 7934 MILLER   CLERK      7782 23-GEN-82  1300             10

Selezionate 11 righe.
```

Come si vede, nella tabella MYEMP ci sono alcuni record mancanti ed un po' di stipendi differenti rispetto alla tabella EMP.

Allineiamo le due tabelle utilizzando MERGE:

```
WTO> MERGE into MYEMP
  2  USING EMP
  3  ON (emp.empno=myemp.empno)
  4  WHEN MATCHED THEN
  5  UPDATE SET
  6    ename    = emp.ename,
  7    job      = emp.job,
  8    mgr      = emp.mgr,
  9    hiredate = emp.hiredate,
 10    sal      = emp.sal,
 11    comm     = emp.comm,
 12    deptno   = emp.deptno
 13  WHEN NOT MATCHED THEN
 14  INSERT (empno, ename, job, mgr, hiredate, sal, comm, deptno)
 15  VALUES (emp.empno,emp.ename, emp.job, emp.mgr, emp.hiredate,
 16          emp.sal, emp.comm, emp.deptno)
 17  ;

14 di righe unite.

WTO> select * from myemp;

EMPNO ENAME    JOB         MGR HIREDATE     SAL   COMM  DEPTNO
----- -------- --------- ----- --------- ------ ------ -------
 7566 JONES    MANAGER    7839 02-APR-81   2975             20
 7654 MARTIN   SALESMAN   7698 28-SET-81   1250   1400      30
 7698 BLAKE    MANAGER    7839 01-MAG-81   2850             30
 7782 CLARK    MANAGER    7839 09-GIU-81   2450             10
 7788 SCOTT    ANALYST    7566 09-DIC-82   3000             20
 7839 KING     PRESIDENT       17-NOV-81   5000             10
 7844 TURNER   SALESMAN   7698 08-SET-81   1500      0      30
 7876 ADAMS    CLERK      7788 12-GEN-83   1100             20
 7900 JAMES    CLERK      7698 03-DIC-81    950             30
 7902 FORD     ANALYST    7566 03-DIC-81   3000             20
 7934 MILLER   CLERK      7782 23-GEN-82   1300             10
 7369 SMITH    CLERK      7902 17-DIC-80    800             20
 7499 ALLEN    SALESMAN   7698 20-FEB-81   1600    300      30
 7521 WARD     SALESMAN   7698 22-FEB-81   1250    500      30

Selezionate 14 righe.
```

L'aggiornamento ha avuto successo ed ora le due tabelle sono allineate. Rivediamo l'istruzione passo per passo:

- MERGE INTO MYEMP

indica la tabella da aggiornare

- USING EMP

indica la tabella da cui leggere.

- ON (emp.empno=myemp.empno)

è la condizione da controllare, se questa è vera si effettuerà un UPDATE, altrimenti un INSERT.

- WHEN MATCHED THEN

quando la condizione di cui sopra è vera...

- UPDATE SET etc...

esegui questo UPDATE.

- WHEN NOT MATCHED THEN

altrimenti...

- INSERT (empno, ename, job, mgr, hiredate, sal, comm, deptno) etc..

esegui questo INSERT.

C'è qualche clausola della MERGE che non è stata utilizzata nell'esempio.

All'UPDATE può essere aggiunta una clausola WHERE per aggiornare solo determinati record. Se, per esempio, vogliamo aggiornare il record solo se lo stipendio nuovo è maggiore di quello vecchio faremo:

```
MERGE into MYEMP
USING EMP
ON (emp.empno=myemp.empno)
WHEN MATCHED THEN
UPDATE SET
  ename    = emp.ename,
  job      = emp.job,
  mgr      = emp.mgr,
  hiredate = emp.hiredate,
  sal      = emp.sal,
  comm     = emp.comm,
  deptno   = emp.deptno
WHERE sal < emp.sal
WHEN NOT MATCHED THEN
INSERT (empno, ename, job, mgr, hiredate, sal, comm, deptno)
VALUES (emp.empno, emp.ename, emp.job, emp.mgr, emp.hiredate,
        emp.sal, emp.comm, emp.deptno)
;
```

C'è poi la possibilità di specificare una clausola DELETE nell'UPDATE.

Serve per cancellare dalla tabella di destinazione i record che, dopo l'UPDATE, verificano una certa condizione.

Se per esempio vogliamo rifare l'aggiornamento di prima ma cancellare da MYEMP i dipendenti che, dopo l'aggiornamento, hanno stipendio minore di 1000 facciamo:

```
WTO> MERGE into MYEMP
  2  USING EMP
```

```
  3  ON (emp.empno=myemp.empno)
  4  WHEN MATCHED THEN
  5  UPDATE SET
  6    ename    = emp.ename,
  7    job      = emp.job,
  8    mgr      = emp.mgr,
  9    hiredate = emp.hiredate,
 10    sal      = emp.sal,
 11    comm     = emp.comm,
 12    deptno   = emp.deptno
 13  DELETE WHERE sal < 1000
 14  WHEN NOT MATCHED THEN
 15  INSERT (empno, ename, job, mgr, hiredate, sal, comm, deptno)
 16  VALUES (emp.empno,emp.ename, emp.job, emp.mgr, emp.hiredate,
 17          emp.sal, emp.comm, emp.deptno)
 18  ;

14 di righe unite.


WTO>  select * from myemp;

EMPNO ENAME    JOB         MGR HIREDATE     SAL   COMM  DEPTNO
----- -------- --------- ----- --------- ------ ------ -------
 7566 JONES    MANAGER    7839 02-APR-81   2975             20
 7654 MARTIN   SALESMAN   7698 28-SET-81   1250   1400      30
 7698 BLAKE    MANAGER    7839 01-MAG-81   2850             30
 7782 CLARK    MANAGER    7839 09-GIU-81   2450             10
 7788 SCOTT    ANALYST    7566 09-DIC-82   3000             20
 7839 KING     PRESIDENT       17-NOV-81   5000             10
 7844 TURNER   SALESMAN   7698 08-SET-81   1500      0      30
 7876 ADAMS    CLERK      7788 12-GEN-83   1100             20
 7902 FORD     ANALYST    7566 03-DIC-81   3000             20
 7934 MILLER   CLERK      7782 23-GEN-82   1300             10
 7369 SMITH    CLERK      7902 17-DIC-80    800             20
 7499 ALLEN    SALESMAN   7698 20-FEB-81   1600    300      30
 7521 WARD     SALESMAN   7698 22-FEB-81   1250    500      30

Selezionate 13 righe.
```

Prima sono state aggiornate le 14 righe, poi la riga del dipendente JAMES (che prima dell'update aveva uno stipendio di 1000 e dopo passa ad uno stipendio di 950) è stata cancellata.

Attenzione: non è stata cancellata la riga di SMITH che, pur avendo uno stipendio di 800, non era presente nella tabella MYEMP prima della MERGE e dunque è entrato con l'INSERT, non con l'UPDATE. La DELETE si applica solo ai record che sono stati aggiornati, non a tutti i record della tabella.

Anche la INSERT può avere una clausola WHERE per inserire solo i record che verificano una certa condizione, ma non può avere una clausola DELETE.

Ad esempio per inserire in MYEMP solo i dipendenti con stipendio maggiore di 1000:

```
WTO> MERGE into MYEMP
  2  USING EMP
  3  ON (emp.empno=myemp.empno)
```

```
  4  WHEN MATCHED THEN
  5  UPDATE SET
  6    ename    = emp.ename,
  7    job      = emp.job,
  8    mgr      = emp.mgr,
  9    hiredate = emp.hiredate,
 10    sal      = emp.sal,
 11    comm     = emp.comm,
 12    deptno   = emp.deptno
 13  WHEN NOT MATCHED THEN
 14  INSERT (empno, ename, job, mgr, hiredate, sal, comm, deptno)
 15  VALUES (emp.empno,emp.ename, emp.job, emp.mgr, emp.hiredate,
 16          emp.sal, emp.comm, emp.deptno)
 17  WHERE emp.sal>1000
 18  ;
13 righe unite.

WTO>  select * from myemp;

EMPNO ENAME      JOB         MGR HIREDATE      SAL   COMM  DEPTNO
----- ---------- --------- ---- --------- ------- ------- -------
 7566 JONES      MANAGER    7839 02-APR-81    2975              20
 7654 MARTIN     SALESMAN   7698 28-SET-81    1250   1400       30
 7698 BLAKE      MANAGER    7839 01-MAG-81    2850              30
 7782 CLARK      MANAGER    7839 09-GIU-81    2450              10
 7788 SCOTT      ANALYST    7566 09-DIC-82    3000              20
 7839 KING       PRESIDENT       17-NOV-81    5000              10
 7844 TURNER     SALESMAN   7698 08-SET-81    1500      0       30
 7876 ADAMS      CLERK      7788 12-GEN-83    1100              20
 7900 JAMES      CLERK      7698 03-DIC-81     950              30
 7902 FORD       ANALYST    7566 03-DIC-81    3000              20
 7934 MILLER     CLERK      7782 23-GEN-82    1300              10
 7499 ALLEN      SALESMAN   7698 20-FEB-81    1600    300       30
 7521 WARD       SALESMAN   7698 22-FEB-81    1250    500       30

Selezionate 13 righe.
```

Questa volta a restare fuori è SMITH che ha uno stipendio di 800. Attenzione: nella clausola WHERE della INSERT non è possibile referenziare colonne della tabella che si sta aggiornando, ma solo colonne della tabella sorgente.

7.3.6 Gestione degli errori in SQL con LOG ERRORS

Quest'argomento è considerato di livello avanzato. Per comprendere pienamente gli esempi è necessario conoscere argomenti di SQL e PL/SQL che in questa fase del manuale ancora non sono stati introdotti. Il lettore principiante può saltare questo paragrafo e tornarci alla fine del libro quando avrà acquisito gli strumenti necessari ad affrontarlo.

Spesso capita di dover effettuare un INSERT, UPDATE, DELETE o MERGE massivo su una tabella e di aver bisogno di gestire gli errori a livello di singolo record, quindi scartare i record che per qualche motivo non possono essere inseriti (o aggiornati o cancellati) ma lavorare quelli che non hanno problemi.

Negli esempi che seguono si parlerà sempre di INSERT, ma tutto ciò che sarà mostrato è valido alla stessa maniera anche per le altre istruzioni DML.

Connettendoci con lo schema SCOTT, ipotizziamo di voler travasare i record della tabella EMP in una nuova tabella EMP2 fatta come segue:

```
WTO> desc emp
 Name                          Null?    Type
 ----------------------------- -------- ----------------------------
 EMPNO                         NOT NULL NUMBER(4)
 ENAME                                  VARCHAR2(10)
 JOB                                    VARCHAR2(9)
 MGR                                    NUMBER(4)
 HIREDATE                               DATE
 SAL                                    NUMBER(7,2)
 COMM                                   NUMBER(7,2)
 DEPTNO                                 NUMBER(2)

WTO> create table emp2
  2  as select * from emp where 1=2;
Table created.

WTO> desc emp2;
 Name                          Null?    Type
 ----------------------------- -------- ----------------------------
 EMPNO                                  NUMBER(4)
 ENAME                                  VARCHAR2(10)
 JOB                                    VARCHAR2(9)
 MGR                                    NUMBER(4)
 HIREDATE                               DATE
 SAL                                    NUMBER(7,2)
 COMM                                   NUMBER(7,2)
 DEPTNO                                 NUMBER(2)

WTO> alter table emp2 modify (
  2  COMM not null,
  3  ENAME varchar2(5));
Table altered.

WTO> desc emp2
 Name                           Null?    Type
 ------------------------------ -------- ---------------------------
 EMPNO                                   NUMBER(4)
 ENAME                                   VARCHAR2(5)
 JOB                                     VARCHAR2(9)
 MGR                                     NUMBER(4)
 HIREDATE                                DATE
 SAL                                     NUMBER(7,2)
 COMM                           NOT NULL NUMBER(7,2)
 DEPTNO                                  NUMBER(2)
```

Ovviamente alcuni record di EMP non possono essere travasati in EMP2 perché hanno ENAME troppo lungo oppure COMM non valorizzato.

Fino ad Oracle 10gR1 (10.1.*) eravamo costretti a scrivere un programma PL/SQL per elaborare una ad una le righe da inserire e gestire in un blocco EXCEPTION gli errori, qualcosa del tipo:

```
BEGIN
  FOR r in (Select * from emp) loop
    begin
```

```
      insert into emp2 values (...)
    exception
      -- i record che non possono essere inseriti in EMP2
      -- finiscono per essere gestiti qui...
    end;
  end loop;
END;
/
```

In Oracle 10gR2 (10.2.*) è stata introdotta la clausola LOG ERRORS nei comandi di INSERT, UPDATE, DELETE e MERGE che consente di definire una tabella di log in cui inserire i record scartati durante l'inserimento (o le altre operazioni).

Vediamo un esempio.

Analizzando le lunghezze della colonna ENAME in EMP e la valorizzazione del campo COMM si vede quanto segue:

```
WTO> select length(ename), count(0)
  2  from emp
  3  group by length(ename);

LENGTH(ENAME)   COUNT(0)
------------- ----------
            6          3
            5          8
            4          3

WTO> select count(*), count(comm)
  2  from emp;

  COUNT(*) COUNT(COMM)
---------- -----------
        14           4

WTO> select count(0) from emp
  2  where LENGTH(ENAME)>5 or comm is null;

  COUNT(0)
----------
        12
```

Ci sono in pratica solo due record "buoni" che hanno lunghezza di ENAME non maggiore di 5 e COMM valorizzato.

Procediamo con l'inserimento:

```
WTO> insert into emp2
  2  select * from emp
  3  log errors;
insert into emp2
*
ERROR at line 1:
ORA-00942: table or view does not exist
```

La tabella che non esiste è quella in cui inserire i record errati.

La clausola LOG ERRORS consente di specificare il nome della tabella degli errori, nel caso in cui questo non venga specificato il nome di

default è ERR$_ seguito dai primi 25 caratteri del nome della tabella in cui si cerca di inserire.

Nel nostro caso, dunque, Oracle cerca di inserire gli errori nella tabella ERR$_EMP2 che non esiste.

Come si crea la tabella di gestione degli errori?

Utilizzando il package DBMS_ERRLOG che ha un'unica procedura CREATE_ERROR_LOG:

```
WTO> desc DBMS_ERRLOG
PROCEDURE CREATE_ERROR_LOG
 Argument Name                  Type        In/Out Default?
 ------------------------------ ----------- ------ --------
 DML_TABLE_NAME                 VARCHAR2    IN
 ERR_LOG_TABLE_NAME             VARCHAR2    IN     DEFAULT
 ERR_LOG_TABLE_OWNER            VARCHAR2    IN     DEFAULT
 ERR_LOG_TABLE_SPACE            VARCHAR2    IN     DEFAULT
 SKIP_UNSUPPORTED               BOOLEAN     IN     DEFAULT

WTO> exec DBMS_ERRLOG.CREATE_ERROR_LOG('EMP2');

PL/SQL procedure successfully completed.

WTO> desc ERR$_EMP2
 Name                 Null?    Type
 -------------------- -------- ----------------
 ORA_ERR_NUMBER$               NUMBER
 ORA_ERR_MESG$                 VARCHAR2(2000)
 ORA_ERR_ROWID$                ROWID
 ORA_ERR_OPTYP$                VARCHAR2(2)
 ORA_ERR_TAG$                  VARCHAR2(2000)
 EMPNO                         VARCHAR2(4000)
 ENAME                         VARCHAR2(4000)
 JOB                           VARCHAR2(4000)
 MGR                           VARCHAR2(4000)
 HIREDATE                      VARCHAR2(4000)
 SAL                           VARCHAR2(4000)
 COMM                          VARCHAR2(4000)
 DEPTNO                        VARCHAR2(4000)
```

La tabella di gestione degli errori possiede cinque colonne di sistema (dopo ne vedremo il contenuto) ed una colonna VARCHAR2(4000) per ogni colonna della tabella applicativa.

Proviamo nuovamente con l'insert:

```
WTO> insert into emp2
  2  select * from emp
  3  log errors;
select * from emp
       *
ERROR at line 2:
ORA-01400: cannot insert NULL into ("SCOTT"."EMP2"."COMM")
```

Ancora errore, il problema è che la clausola LOG ERRORS prevede la possibilità di specificare il numero massimo di record che possono essere scartati (REJECT LIMIT), superato quel numero di errori lo statement va in errore.

Il default di REJECT LIMIT è zero, quindi se non si specifica basta un solo record errato per mandare in errore l'intera istruzione.

Specifichiamo allora REJECT LIMIT:

```
WTO> insert into emp2
  2  select * from emp
  3  log errors reject limit 20
  4  ;

2 rows created.

WTO> select count(0) from ERR$_EMP2;

  COUNT(0)
----------
        12
```

Due righe sono state correttamente travasate, dodici errori sono stati loggati nella tabella ERR$_EMP2.

Vediamo il contenuto della tabella degli errori, cominciamo con le colonne di sistema. Visualizziamo in quest'esempio solo il messaggio d'errore:

```
WTO> select ORA_ERR_MESG$
  2* from ERR$_EMP2

ORA_ERR_MESG$
------------------------------------------------------------
ORA-12899: value too large for column "SCOTT"."EMP2"."ENAME"
ORA-01400: cannot insert NULL into ("SCOTT"."EMP2"."COMM")
ORA-01400: cannot insert NULL into ("SCOTT"."EMP2"."COMM")
ORA-01400: cannot insert NULL into ("SCOTT"."EMP2"."COMM")
ORA-01400: cannot insert NULL into ("SCOTT"."EMP2"."COMM")
ORA-12899: value too large for column "SCOTT"."EMP2"."ENAME"
ORA-01400: cannot insert NULL into ("SCOTT"."EMP2"."COMM")
ORA-01400: cannot insert NULL into ("SCOTT"."EMP2"."COMM")
ORA-01400: cannot insert NULL into ("SCOTT"."EMP2"."COMM")
ORA-12899: value too large for column "SCOTT"."EMP2"."ENAME"
ORA-01400: cannot insert NULL into ("SCOTT"."EMP2"."COMM")
ORA-01400: cannot insert NULL into ("SCOTT"."EMP2"."COMM")

12 rows selected.
```

Sono però disponibili le seguenti informazioni:

- ORA_ERR_NUMBER$: il codice di errore Oracle
- ORA_ERR_MESG$: il messaggio d'errore Oracle
- ORA_ERR_ROWID$: il rowid della riga che è andata in errore, in caso di INSERT è sempre null perché la riga è andata in errore prima di essere inserita, quando non essendo ancora presente nel DB non aveva ancora un rowid, per UPDATE e DELETE è sempre valorizzato. Per la MERGE dipende dal fatto che la riga era in fase di inserimento o di aggiornamento.

- ORA_ERR_OPTYP$: il tipo di operazione (insert (I), update (U), delete (D))

- ORA_ERR_TAG$: un tag che opzionalmente è possibile indicare nella clausola LOG ERRORS per identificare meglio i record nella tabella degli errori

Per ogni record rigettato c'è un solo errore, anche su quelli che avrebbero avuto più di un motivo per essere scartati.

Per valorizzare il campo ORA_ERR_TAG$ si fa come segue:

```
WTO> insert into emp2
  2  select * from emp
  3  log errors ('ins2') reject limit 20;

2 rows created.

WTO> select ORA_ERR_TAG$, count(0)
  2  from ERR$_EMP2
  3* group by ORA_ERR_TAG$

ORA_ERR_TAG$                     COUNT(0)
------------------------------ ----------
                                       12
ins2                                   12
```

Specificando quindi un tag diverso per ogni statement è poi facile identificare gli scarti associandoli a diverse istruzioni.

Il tag può essere una generica espressione, come ad esempio:

```
WTO> insert into emp2
  2  select * from emp
  3  log errors (to_char(sysdate,'yyyy-mm-dd hh24:mi:ss'))
  4  reject limit 20;

2 rows created.

WTO> select ORA_ERR_TAG$, count(0)
  2  from ERR$_EMP2
  3  group by ORA_ERR_TAG$
  4  ;

ORA_ERR_TAG$                     COUNT(0)
------------------------------ ----------
                                       12
2010-02-05 18:54:31                    12
ins2                                   12
```

In modo da avere in tabella l'orario dell'errore.

Nelle altre colonne, invece, ci sono i dati che abbiamo cercato di inserire in tabella:

```
WTO> select EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO
  2  from ERR$_EMP2
  3  where ORA_ERR_TAG$ = 'ins2';

EMPNO ENAME      JOB         MGR   HIREDATE   SAL     COMM    DEPTNO
```

```
----- ---------- ---------- ----- ---------- ------ ------ ------
7369  SMITH      CLERK      7902  17-DEC-80  800           20
7566  JONES      MANAGER    7839  02-APR-81  2975          20
7654  MARTIN     SALESMAN   7698  28-SEP-81  1250   1400   30
7698  BLAKE      MANAGER    7839  01-MAY-81  2850          30
7782  CLARK      MANAGER    7839  09-JUN-81  2450          10
7788  SCOTT      ANALYST    7566  19-APR-87  3000          20
7839  KING       PRESIDENT        17-NOV-81  5000          10
7844  TURNER     SALESMAN   7698  08-SEP-81  1500      0   30
7876  ADAMS      CLERK      7788  23-MAY-87  1100          20
7900  JAMES      CLERK      7698  03-DEC-81  950           30
7902  FORD       ANALYST    7566  03-DEC-81  3000          20
7934  MILLER     CLERK      7782  23-JAN-82  1300          10

12 rows selected.
```

7.4 SELECT, il comando di ricerca

Il comando SELECT consente di estrarre i dati presenti nel DB. La forma più semplice del comando è

```
SELECT * FROM <nome schema>.<nome oggetto>;
```

Dove l'asterisco significa "tutte le colonne" e l'oggetto può essere una tabella, una vista, una vista materializzata oppure un sinonimo di uno di questi.

Da questo momento in poi parleremo semplicemente di SELECT da una o più tabelle. Resta inteso, però, che tutto ciò che sarà detto resta valido se al posto delle tabelle si utilizzano viste, viste materializzate o sinonimi di questi oggetti.

Nella forma semplice sopra esposta, il comando SELECT non include alcuna condizione che consenta di scegliere le righe da estrarre, quindi saranno estratte tutte le righe presenti in tabella. Ad esempio per estrarre tutti i dati presenti nella tabella CLIENTI si procederà come segue:

```
WTO >select * from clienti;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964

Selezionate 8 righe.
```

Il comando base può essere complicato con una serie molto ampia di clausole aggiuntive. Nel seguito saranno illustrate le principali.

7.4.1 Proiezioni

Partendo dal comando base sopra illustrato, il primo raffinamento dell'istruzione è la scelta delle colonne da visualizzare. Tale query, detta

anche *proiezione* della tabella, si realizza specificando le colonne che si desiderano estrarre, separate da virgola.

Nell'esempio seguente si estraggono solo le colonne COGNOME ed INDIRIZZO della tabella CLIENTI.

```
WTO >select cognome, indirizzo from clienti;

COGNOME  INDIRIZZO
-------- ----------------------
ROSSI    VIA LAURENTINA, 700
BIANCHI  VIA OSTIENSE, 850
VERDI    VIA DEL MARE, 8
NERI     VIA TORINO, 30
COLOMBO  PIAZZA DUOMO, 1
ESPOSITO VIA CARACCIOLO, 100
RUSSO    VIA GIULIO CESARE, 119
AMATO    VIA NAPOLI, 234

Selezionate 8 righe.
```

Ad ogni colonna può essere assegnato un alias, cioè un nome alternativo che sarà utilizzato da Oracle nella visualizzazione dei dati e può essere utilizzato da chi scrive la query nella clausola di ordinamento, di cui parleremo più avanti.

Per assegnare un alias ad una colonna è sufficiente far seguire il nome della colonna dalla parola chiave AS e dall'alias.

```
WTO >select cognome as c, indirizzo as i
  2  from clienti;

C                              I
------------------------------ -----------------------------
ROSSI                          VIA LAURENTINA, 700
BIANCHI                        VIA OSTIENSE, 850
VERDI                          VIA DEL MARE, 8
NERI                           VIA TORINO, 30
COLOMBO                        PIAZZA DUOMO, 1
ESPOSITO                       VIA CARACCIOLO, 100
RUSSO                          VIA GIULIO CESARE, 119
AMATO                          VIA NAPOLI, 234

Selezionate 8 righe.
```

La parola chiave AS è opzionale, quindi si può scrivere anche

```
WTO >select cognome c, indirizzo i
  2  from clienti;

C                              I
------------------------------ -----------------------------
ROSSI                          VIA LAURENTINA, 700
BIANCHI                        VIA OSTIENSE, 850
VERDI                          VIA DEL MARE, 8
NERI                           VIA TORINO, 30
COLOMBO                        PIAZZA DUOMO, 1
ESPOSITO                       VIA CARACCIOLO, 100
RUSSO                          VIA GIULIO CESARE, 119
AMATO                          VIA NAPOLI, 234

Selezionate 8 righe.
```

Alcune volte capita di dover leggere dal database un dato che non è presente in alcuna tabella, ad esempio una stringa di testo fissa, il risultato di un calcolo aritmetico oppure la data di sistema (SYSDATE). Tali dati particolari possono essere estratti da qualunque tabella

```
WTO >select sysdate from clienti;

SYSDATE
---------
02-FEB-11
02-FEB-11
02-FEB-11
02-FEB-11
02-FEB-11
02-FEB-11
02-FEB-11
02-FEB-11

Selezionate 8 righe.

WTO >select 3*5 from clienti;

       3*5
----------
        15
        15
        15
        15
        15
        15
        15
        15

Selezionate 8 righe.

WTO >select 'Una stringa fissa' from clienti;

'UNASTRINGAFISSA'
-----------------
Una stringa fissa
Una stringa fissa
Una stringa fissa
Una stringa fissa
Una stringa fissa
Una stringa fissa
Una stringa fissa
Una stringa fissa

Selezionate 8 righe.

WTO >select sysdate, 3*5, 'Una stringa fissa'
  2  from clienti;

SYSDATE          3*5 'UNASTRINGAFISSA'
--------- ---------- -----------------
02-FEB-11         15 Una stringa fissa
02-FEB-11         15 Una stringa fissa
02-FEB-11         15 Una stringa fissa
02-FEB-11         15 Una stringa fissa
02-FEB-11         15 Una stringa fissa
02-FEB-11         15 Una stringa fissa
```

```
02-FEB-11         15 Una stringa fissa
02-FEB-11         15 Una stringa fissa
Selezionate 8 righe.
```

In tali casi l'utilizzo dell'alias è particolarmente utile poiché consente di assegnare un nome significativo ad una colonna che altrimenti avrebbe un'intestazione poco gestibile:

```
WTO >select cognome, 6*8 prodotto, sysdate oggi
  2  from clienti;

COGNOME    PRODOTTO OGGI
-------- ---------- ---------
ROSSI            48 02-FEB-11
BIANCHI          48 02-FEB-11
VERDI            48 02-FEB-11
NERI             48 02-FEB-11
COLOMBO          48 02-FEB-11
ESPOSITO         48 02-FEB-11
RUSSO            48 02-FEB-11
AMATO            48 02-FEB-11
Selezionate 8 righe.
```

7.4.2 La tabella DUAL

Torniamo alla query precedente che estraeva da CLIENTI dei dati non presenti in tabella.

Ovviamente, poiché nel comando non si è specificato quanti record si desiderano estrarre dalla tabella CLIENTI, viene estratta una riga per ogni record presente in tabella. Se la tabella è vuota non viene estratto nulla

```
WTO >delete test;
Eliminate 4 righe.

WTO >select sysdate, 3*5, 'Una stringa fissa'
  2  from test;

Nessuna riga selezionata
```

Anche per leggere il prossimo valore di una sequenza può essere utilizzata una qualunque tabella. In questo caso c'è da tener presente la particolarità che Oracle leggerà un nuovo valore diverso per ogni record estratto:

```
WTO >create sequence test_seq;
Sequenza creata.

WTO >select test_seq.nextval from clienti;

   NEXTVAL
----------
         1
         2
         3
         4
         5
         6
         7
         8
Selezionate 8 righe.
```

```
WTO >select test_seq.currval from clienti;
   CURRVAL
----------
         8
         8
         8
         8
         8
         8
         8
         8
Selezionate 8 righe.
```

Dagli esempi precedenti risulta evidente che, prima di eseguire una SELECT di valori non contenuti nel DB ,dobbiamo preoccuparci di sapere se la tabella utilizzata nella clausola FROM contiene delle righe e quante.

Per ovviare a questo problema Oracle mette a disposizione una tabella d'appoggio: DUAL. La tabella DUAL, creata per default nello schema SYS ed accessibile da tutti gli utenti, contiene una sola colonna (DUMMY) ed una sola riga. L'unico dato della tabella è una stringa 'X'.

```
WTO >desc dual
 Nome                 Nullo?    Tipo
 ------------------ --------- ------------
 DUMMY                          VARCHAR2(1)

WTO >select * from dual;
D
-
X
```

Il nome potrebbe trarre in inganno visto che DUAL in inglese significa "doppio" oppure "duplice". DUAL non ha nulla di doppio. Il nome deriva infatti dal motivo originale della creazione della tabella, che serviva, messa in prodotto cartesiano con qualunque altra tabella, a raddoppiarne i record. Il concetto di prodotto cartesiano sarà chiarito nel seguito, basti per ora dire che leggendo senza condizioni i record di due tabelle insieme si ottiene un numero di record pari al prodotto dei numeri di record presenti nelle singole tabelle. Per raddoppiare i record di una tabella, dunque, è sufficiente leggerli insieme a quelli di un'altra tabella che contenga esattamente due righe. La tabella DUAL conteneva originariamente due record, non uno. Ciò ne giustifica il nome. Successivamente la tabella non è stata utilizzata per il suo scopo originario ma è utilissima come tabella d'appoggio.

```
WTO >select sysdate, 3*5, 'Una stringa fissa', test_seq.nextval
  2  from dual;

SYSDATE          3*5 'UNASTRINGAFISSA'    NEXTVAL
--------- ---------- ----------------- ----------
02-FEB-11         15 Una stringa fissa          9
```

7.4.3 Pseudo colonne

Una pseudo colonna si comporta esattamente come una colonna ma non è archiviata nella tabella. Si tratta di campi di sola lettura che forniscono specifiche informazioni particolarmente importanti in alcuni tipi di query.

Tre pseudo colonne sono utilizzabili esclusivamente nell'ambito delle query gerarchiche e le introdurremo, dunque, quando sarà trattato questo argomento. Si tratta di:

- CONNECT_BY_ISCYCLE
- CONNECT_BY_ISLEAF
- LEVEL

Due pseudo colonne sono state già descritte quando si è parlato delle sequenze:

- CURRVAL (ritorna il valore corrente della sequenza)
- NEXTVAL (incrementa la sequenza ritornando il nuovo valore)

Le pseudo colonne

- COLUMN_VALUE
- XMLDATA

Saranno introdotte quando si discuterà delle funzionalità per la gestione dei dati XML.

La pseudocolonna ROWID indica la collocazione fisica nei datafile di ciascun record presente nel database.

Si guardi ad esempio la query seguente:

```
WTO >select cognome, rowid
  2  from clienti;

COGNOME  ROWID
-------- ------------------
ROSSI    AAAlsNAAEAAA7gkAAA
BIANCHI  AAAlsNAAEAAA7gkAAB
VERDI    AAAlsNAAEAAA7gkAAC
NERI     AAAlsNAAEAAA7gkAAD
COLOMBO  AAAlsNAAEAAA7gkAAE
ESPOSITO AAAlsNAAEAAA7gkAAF
RUSSO    AAAlsNAAEAAA7gkAAG
AMATO    AAAlsNAAEAAA7gkAAH

Selezionate 8 righe.
```

Per ciascuna riga è stato estratto un identificatore di riga che include quattro informazioni nel formato OOOOOOFFFBBBBBBRRR.

Ogni informazione contenuta nel ROWID è un numero in base 64 dove le sessantaquattro cifre sono: le 26 lettere maiuscole da "A" a "Z", le 26 lettere minuscole da "a" a "z", le cifre da "0" a "9", , i simboli "+" e "/". Nel dettaglio il ROWID deve essere letto come segue

- OOOOOO è l'identificativo dell'oggetto: tabella, vista materializzata etc di cui il record fa parte.
- FFF è il numero del datafile

- BBBBBB è il numero del blocco all’interno del datafile
- RRR è il numero di riga all’interno del blocco.

Nell’esempio precedente tutti i record appartengono all’oggetto AAAIsN, tutti si trovano nel blocco AAA7gk del file AAE. L’unica cosa che cambia di record in record è ovviamente la posizione nel blocco: ROSSI è il primo record nel blocco (AAA in base 64 con la codifica definita equivale a zero in base 10), BIANCHI ha indice uno (AAB) ed è quindi il secondo record e così via.

Per ogni riga estratta da una query, la pseudocolonna ROWNUM restituisce un numero che indica l'ordine in cui Oracle seleziona la riga dalla tabella. La prima riga selezionata ha ROWNUM uguale a 1, la seconda 2 e così via.

```
WTO >select cognome, rownum
  2  from clienti;

COGNOME      ROWNUM
-------- ----------
ROSSI             1
BIANCHI           2
VERDI             3
NERI              4
COLOMBO           5
ESPOSITO          6
RUSSO             7
AMATO             8

Selezionate 8 righe.
```

Il concetto è molto semplice ma genera spesso confusione quando ROWNUM è utilizzata in combinazione con altre clausole (WHERE ed ORDER BY) della SELECT che ancora non abbiamo introdotto.

Ci si limiterà a questo, dunque, per il momento per poi tornare sulla questione più avanti.

7.4.4 Selezioni

Per estrarre solo una parte delle righe presenti in tabella si utilizza la clausola WHERE.

```
SELECT <elenco colonne>
  FROM <nome schema>.<nome oggetto>
 WHERE <condizione>;
```

Una condizione è un’espressione che per ogni riga della tabella può assumere il valore VERO oppure FALSO (10.9). Le righe per le quali la condizione assume il valore VERO vengono estratte.

In una condizione si possono utilizzare:

- Le colonne delle tabelle coinvolte nella query
- Operatori di confronto

- Valori fissi
- Operatori aritmetici
- Operatori logici
- Pseudo colonne

La condizione più semplice assume la forma

```
<nome colonna> <operatore di confronto> <valore>
```

Possono essere utilizzati i seguenti operatori di confronto.

- Uguaglianza e differenza (=; !=, <>, ^=)

I simboli !=, <> e ^= sono perfettamente equivalenti.

Nell'esempio seguente sono estratti i comuni che si trovano in provincia di Roma.

```
WTO >select * from comuni
  2  where provincia='RM';

COD_ DES_COMUNE                                           PR
---- -------------------------------------------------- --
H501 ROMA                                                 RM
G811 POMEZIA                                              RM
M297 FIUMICINO                                            RM
```

Mentre nei tre esempi seguenti si estraggono i comuni avente provincia diversa da 'RM'.

```
WTO >select * from comuni
  2  where provincia!='RM';

COD_ DES_COMUNE                                           PR
---- -------------------------------------------------- --
F205 MILANO                                               MI
G686 PIOLTELLO                                            MI
E415 LAINATE                                              MI
F839 NAPOLI                                               NA
G964 POZZUOLI                                             NA
G902 PORTICI                                              NA

Selezionate 6 righe.

WTO >select * from comuni
  2  where provincia<>'RM';

COD_ DES_COMUNE                                           PR
---- -------------------------------------------------- --
F205 MILANO                                               MI
G686 PIOLTELLO                                            MI
E415 LAINATE                                              MI
F839 NAPOLI                                               NA
G964 POZZUOLI                                             NA
G902 PORTICI                                              NA

Selezionate 6 righe.

WTO >select * from comuni
```

```
  2  where provincia^='RM';

COD_ DES_COMUNE                                        PR
---- -------------------------------------------------- --
F205 MILANO                                            MI
G686 PIOLTELLO                                         MI
E415 LAINATE                                           MI
F839 NAPOLI                                            NA
G964 POZZUOLI                                          NA
G902 PORTICI                                           NA

Selezionate 6 righe.
```

- Confronto con NULL (IS NULL, IS NOT NULL)

Gli esempi appena visti chiariscono il comportamento degli operatori di diseguaglianza quando tutti i dati sono valorizzati. Quando alcuni dati sono assenti, invece, bisogna fare un po' d'attenzione.

Oracle assegna ai dati assenti un valore speciale: NULL.

Nell'esempio che segue si estraggono tutti i record dalla tabella clienti e poi si cerca, senza successo, di estrarre i soli record muniti di codice fiscale e quelli invece privi di codice fiscale.

```
WTO >select * from clienti;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964

Selezionate 8 righe.

WTO >select * from clienti
  2  where cod_fisc = NULL;

Nessuna riga selezionata

WTO >select * from clienti
  2  where cod_fisc != NULL;

Nessuna riga selezionata
```

Tutti gli operatori di confronto visti finora, nonché quelli che saranno introdotti successivamente, restituiscono il valore UNKNOWN (sconosciuto) quando sono applicati ad una colonna di valore NULL.

Le condizioni che restituiscono UNKNOWN sono trattate come se fossero FALSE. Il record non viene estratto.

Questo vuol dire che in presenza di valori NULL non vale la regola intuitiva che estraendo i record che verificano la condizione:

```
<nome colonna> = <valore>
```

E quelli che verificano la condizione:

```
<nome colonna> != <valore>
```

Si estraggono tutti i record della tabella.

L'esempio che segue ne è dimostrazione.

```
WTO >select * from clienti;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964

Selezionate 8 righe.

WTO >select * from clienti
  2  where cod_fisc = 'RSSMRC70R20H501X';

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501

WTO >select * from clienti
  2  where cod_fisc != 'RSSMRC70R20H501X';

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
```

Mancano i record del tutto privi di codice fiscale.

Per verificare se un dato è NULL oppure no esistono due operatori di confronto specifici: IS NULL ed IS NOT NULL.

Per estrarre tutti i clienti privi di codice fiscale bisognerà dunque scrivere

```
WTO >select * from clienti
  2  where cod_fisc is null;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964
```

Mentre per estrarre quelli muniti di codice fiscale si scriverà

```
WTO >select * from clienti
  2  where cod_fisc is not null;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
```

```
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
```

La regola intuitiva di cui si diceva prima può essere dunque enunciata come segue: data una tabella T munita di una colonna C che può contenere valori nulli e dato un valore fisso V l'insieme complessivo di tutti i record di T si ottiene unendo i tre insiemi estratti dalle istruzioni:

```
SELECT * FROM T WHERE C = V;
SELECT * FROM T WHERE C != V;
SELECT * FROM T WHERE C IS NULL;
```

La stessa regola vale per tutti gli operatori di confronto che seguono.

- Disuguaglianza (>, >=, <, <=)

L'utilizzo è molto intuitivo e dunque si procede direttamente con gli esempi.

```
WTO >select * from fatture
  2  where importo>500;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N

WTO >select * from fatture
  2  where importo>=500;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S

WTO >select * from fatture
  2  where importo<500;
NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
WTO >select * from fatture
  2  where importo<=500;
NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          5          5 01-DIC-10        500 S
```

Gli operatori possono essere utilizzati anche con le date

```
WTO >select * from fatture
  2  where DATA_FATTURA > date'2010-12-01';

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          4          3 01-FEB-11       1000 N

WTO >select * from fatture
  2  where DATA_FATTURA >= date'2010-12-01';

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
```

```
----------- ---------- --------- ---------- -
          2          1 01-DIC-10        500 N
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S

WTO >select * from fatture
  2  where DATA_FATTURA < date'2010-12-01';

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          3          2 20-OTT-10        700 S

WTO >select * from fatture
  2  where DATA_FATTURA <= date'2010-12-01';

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          5          5 01-DIC-10        500 S
```

Come già detto, il tipo DATE include sempre anno, mese, giorno, ore, minuti e secondi. Due date sono uguali quando tutti gli elementi sono uguali, fino al secondo. Negli esempi precedenti sia le date presenti in tabella che quelle utilizzate per il confronto hanno ore, minuti e secondi pari alle 00:00:00. Se invece in tabella ci fossero state date comprensive di un orario differente il comportamento sarebbe cambiato sensibilmente. La data 01-12-2010 alle 10:00:00 infatti è maggiore, non uguale, del valore fisso *date'2010-12-01'*. Quest'ultimo rappresenta la mezzanotte del 01-12-2010.

Per osservare questo comportamento cambiamo momentaneamente il formato di visualizzazione di default delle date con il comando

```
WTO >ALTER SESSION SET NLS_DATE_FORMAT='dd/mm/yyyy hh24:mi:ss';

Modificata sessione.
```

I formati di rappresentazione delle date saranno spiegati più avanti, per il momento basti dire che in questo modo per ogni data sarà visualizzata anche la parte relativa ad ore, minuti e secondi, come si vede di seguito.

```
WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATTURA           IMPORTO P
----------- ---------- ------------------- ---------- -
          1          1 01/10/2010 00:00:00        300 S
          2          1 01/12/2010 00:00:00        500 N
          3          2 20/10/2010 00:00:00        700 S
          4          3 01/02/2011 00:00:00       1000 N
          5          5 01/12/2010 00:00:00        500 S
```

In tabelle tutte le date sono sprovviste di ora, minuti e secondi.

Anticipiamo un comportamento delle date che riprenderemo più avanti: aggiungendo un numero x ad una data si ottiene una nuova data uguale alla data di partenza più x giorni. Per aggiungere un giorno basta fare DATA+1,

per aggiungere una settimana DATA+7, per aggiungere solo sei ore DATA+1/4, essendo sei ore pari ad un quarto di giorno.

```
WTO >select sysdate, sysdate+1, sysdate+1/4
  2  from dual;

SYSDATE             SYSDATE+1           SYSDATE+1/4
------------------- ------------------- -------------------
05/02/2011 16:03:53 05/02/2011 16:03:53 05/02/2011 22:03:53
```

Modifichiamo tutte le date delle fatture aggiungendo 6 ore mediante l'UPDATE seguente.

```
WTO >update fatture
  2  set data_fattura=data_fattura+1/4
  3  ;

Aggiornate 5 righe.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATTURA           IMPORTO P
----------- ---------- ------------------- ---------- -
          1          1 01/10/2010 06:00:00        300 S
          2          1 01/12/2010 06:00:00        500 N
          3          2 20/10/2010 06:00:00        700 S
          4          3 01/02/2011 06:00:00       1000 N
          5          5 01/12/2010 06:00:00        500 S
```

A questo punto le stesse istruzioni precedenti danno risultati ben diversi.

```
WTO >select * from fatture
  2  where DATA_FATTURA > date'2010-12-01';

NUM_FATTURA NUM_ORDINE DATA_FATTURA           IMPORTO P
----------- ---------- ------------------- ---------- -
          2          1 01/12/2010 06:00:00        500 N
          4          3 01/02/2011 06:00:00       1000 N
          5          5 01/12/2010 06:00:00        500 S

WTO >select * from fatture
  2  where DATA_FATTURA >= date'2010-12-01';

NUM_FATTURA NUM_ORDINE DATA_FATTURA           IMPORTO P
----------- ---------- ------------------- ---------- -
          2          1 01/12/2010 06:00:00        500 N
          4          3 01/02/2011 06:00:00       1000 N
          5          5 01/12/2010 06:00:00        500 S

WTO >select * from fatture
  2  where DATA_FATTURA < date'2010-12-01';

NUM_FATTURA NUM_ORDINE DATA_FATTURA           IMPORTO P
----------- ---------- ------------------- ---------- -
          1          1 01/10/2010 06:00:00        300 S
          3          2 20/10/2010 06:00:00        700 S

WTO >select * from fatture
  2  where DATA_FATTURA <= date'2010-12-01';

NUM_FATTURA NUM_ORDINE DATA_FATTURA           IMPORTO P
----------- ---------- ------------------- ---------- -
```

```
         1          1 01/10/2010 06:00:00       300 S
         3          2 20/10/2010 06:00:00       700 S
```

Gli operatori di disuguaglianza si applicano anche alle stringhe alfanumeriche. In questo caso si intende che una stringa è maggiore di un'altra quando è successiva in ordine alfabetico.

```
WTO >select * from comuni
  2  where des_comune>'MILANO';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
H501 ROMA                                                  RM
G811 POMEZIA                                               RM
G686 PIOLTELLO                                             MI
F839 NAPOLI                                                NA
G964 POZZUOLI                                              NA
G902 PORTICI                                               NA
Selezionate 6 righe.

WTO >select * from comuni
  2  where des_comune>='MILANO';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
H501 ROMA                                                  RM
G811 POMEZIA                                               RM
F205 MILANO                                                MI
G686 PIOLTELLO                                             MI
F839 NAPOLI                                                NA
G964 POZZUOLI                                              NA
G902 PORTICI                                               NA

Selezionate 7 righe.

WTO >select * from comuni
  2  where des_comune<'MILANO';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
M297 FIUMICINO                                             RM
E415 LAINATE                                               MI

WTO >select * from comuni
  2  where des_comune<='MILANO';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
M297 FIUMICINO                                             RM
F205 MILANO                                                MI
E415 LAINATE                                               MI
```

- Appartenenza ad un intervallo (BETWEEN, NOT BETWEEN)

Per verificare se un dato valore è in un intervallo si utilizza l'operatore BETWEEN. Questo operatore accetta ovviamente due valori di confronto, l'estremo inferiore e l'estremo superiore dell'intervallo, separati dalla parola chiave AND.

```
WTO >select * from fatture
  2  where importo between 400 and 750;
```

```
NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          5          5 01-DIC-10        500 S
```

L'intervallo si intende sempre estremi inclusi.

```
WTO >select * from fatture
  2  where importo between 400 and 500;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          2          1 01-DIC-10        500 N
          5          5 01-DIC-10        500 S
```

L'operatore reciproco di BETWEEN è NOT BETWEEN. Esso estrae i record che si trovano al di fuori dell'intervallo.

```
WTO >select * from fatture
  2  where importo not between 400 and 500;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
```

- Appartenenza ad una lista (IN, NOT IN)

Per verificare se un valore è in un elenco si utilizza l'operatore IN. I valori dell'elenco devono essere separati da virgola ed inclusi tra parentesi tonde.

```
WTO >select * from comuni
  2  where provincia in ('RM','MI');

COD_ DES_COMUNE                                         PR
---- -------------------------------------------------- --
H501 ROMA                                               RM
G811 POMEZIA                                            RM
M297 FIUMICINO                                          RM
F205 MILANO                                             MI
G686 PIOLTELLO                                          MI
E415 LAINATE                                            MI

Selezionate 6 righe.

WTO >select * from fatture
  2  where importo in (300,700);

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          3          2 20-OTT-10        700 S

WTO >select * from fatture
  2  where data_fattura in (date'2011-02-01', date'2010-10-20');

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
```

Nella lista possono essere specificati fino a mille valori.

L'operatore reciproco della IN è NOT IN, esso estrae tutti i record per cui la colonna indicata non è nell'elenco.

```
WTO >select * from comuni
  2  where provincia not in ('RM','MI');

COD_ DES_COMUNE                                          PR
---- --------------------------------------------------- --
F839 NAPOLI                                              NA
G964 POZZUOLI                                            NA
G902 PORTICI                                             NA

WTO >select * from fatture
  2  where importo not in (300,700);

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          2          1 01-DIC-10        500 N
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S

WTO >select * from fatture
  2  where data_fattura not in
  3  (date'2011-02-01', date'2010-10-20');

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          5          5 01-DIC-10        500 S
```

- Similitudine (LIKE, NOT LIKE)

L'operatore LIKE consente di verificare se un dato rispetta o meno un modello. Ad esempio con quest'operatore è possibile estrarre tutti i nomi che cominciano per P oppure quelli che hanno una M come terzo carattere.

```
WTO >select * from comuni
  2  where des_comune like 'P%';

COD_ DES_COMUNE                                          PR
---- --------------------------------------------------- --
G811 POMEZIA                                             RM
G686 PIOLTELLO                                           MI
G964 POZZUOLI                                            NA
G902 PORTICI                                             NA

WTO >select * from comuni
  2  where des_comune like '__M%';
COD_ DES_COMUNE                                          PR
---- --------------------------------------------------- --
H501 ROMA                                                RM
G811 POMEZIA                                             RM
```

Per specificare il modello che il campo deve rispettare si possono utilizzare due caratteri speciali:

- ✓ Il carattere % che significa “qualunque stringa di caratteri”.
- ✓ Il carattere _ che significa “qualunque singolo carattere”.

Scrivere dunque il modello 'P%' vuol dire che il nome del comune deve cominciare con P e dopo può avere qualunque stringa di caratteri.

Scrivere il modello '__M%' vuol dire che il nome può cominciare con qualunque carattere, deve avere al secondo posto qualunque carattere, al terzo posto una M e poi una qualunque stringa di caratteri.

Continuando con questa logica vediamo alcuni esempi.

Tutti i comuni che finiscono per O

```
WTO >select * from comuni
  2  where des_comune like '%O';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
M297 FIUMICINO                                             RM
F205 MILANO                                                MI
G686 PIOLTELLO                                             MI
```

Tutti i comuni che contengono nell'ordine una O ed una L

```
WTO >select * from comuni
  2  where des_comune like '%O%L%';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
G686 PIOLTELLO                                             MI
F839 NAPOLI                                                NA
G964 POZZUOLI                                              NA
```

Tutti i comuni che contengono una N

```
WTO >select * from comuni
  2  where des_comune like '%N%';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
M297 FIUMICINO                                             RM
F205 MILANO                                                MI
E415 LAINATE                                               MI
F839 NAPOLI                                                NA
```

Tutti i comuni che contengono almeno due O

```
WTO >select * from comuni
  2  where des_comune like '%O%O%';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
G686 PIOLTELLO                                             MI
G964 POZZUOLI                                              NA
```

L'operatore reciproco di LIKE è NOT LIKE ed estrae tutti i dati che non rispettano il modello.

```
WTO >select * from comuni
  2  where des_comune not like 'P%';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
```

```
H501 ROMA                                                  RM
M297 FIUMICINO                                             RM
F205 MILANO                                                MI
E415 LAINATE                                               MI
F839 NAPOLI                                                NA

WTO >select * from comuni
  2  where des_comune not like '__M%';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
M297 FIUMICINO                                             RM
F205 MILANO                                                MI
G686 PIOLTELLO                                             MI
E415 LAINATE                                               MI
F839 NAPOLI                                                NA
G964 POZZUOLI                                              NA
G902 PORTICI                                               NA

Selezionate 7 righe.

WTO >select * from comuni
  2  where des_comune not like '%O';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
H501 ROMA                                                  RM
G811 POMEZIA                                               RM
E415 LAINATE                                               MI
F839 NAPOLI                                                NA
G964 POZZUOLI                                              NA
G902 PORTICI                                               NA

Selezionate 6 righe.

WTO >select * from comuni
  2  where des_comune not like '%O%L%';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
H501 ROMA                                                  RM
G811 POMEZIA                                               RM
M297 FIUMICINO                                             RM
F205 MILANO                                                MI
E415 LAINATE                                               MI
G902 PORTICI                                               NA
Selezionate 6 righe.
WTO >select * from comuni
  2  where des_comune not like '%N%';

COD_ DES_COMUNE                                            PR
---- -------------------------------------------------- --
H501 ROMA                                                  RM
G811 POMEZIA                                               RM
G686 PIOLTELLO                                             MI
G964 POZZUOLI                                              NA
G902 PORTICI                                               NA

WTO >select * from comuni
  2  where des_comune not like '%O%O%';

COD_ DES_COMUNE                                            PR
```

```
---- ------------------------------------------------- --
H501 ROMA                                              RM
G811 POMEZIA                                           RM
M297 FIUMICINO                                         RM
F205 MILANO                                            MI
E415 LAINATE                                           MI
F839 NAPOLI                                            NA
G902 PORTICI                                           NA

Selezionate 7 righe.
```

L'operatore LIKE pur essendo molto versatile non copre tutte le esigenze di confronto con un modello. Per aumentare tali potenzialità in SQL e PL/SQL sono disponibili funzioni che implementano le espressioni regolari. Se ne parlerà più avanti.

Le singole condizioni viste fin qui possono essere aggregate tra loro mediante operatori logici.

- Congiunzione (AND)

La congiunzione consente di estrarre tutti i record che rispettano contemporaneamente due condizioni. È equivalente all'operazione insiemistica (10.8) di INTERSEZIONE.

Ad esempio l'insieme dei comuni consiste in tutti i record presenti nella tabella COMUNI.

```
WTO >select * from comuni;
COD_ DES_COMUNE                                        PR
---- ------------------------------------------------- --
H501 ROMA                                              RM
G811 POMEZIA                                           RM
M297 FIUMICINO                                         RM
F205 MILANO                                            MI
G686 PIOLTELLO                                         MI
E415 LAINATE                                           MI
F839 NAPOLI                                            NA
G964 POZZUOLI                                          NA
G902 PORTICI                                           NA
Selezionate 9 righe.
```

L'insieme di tutti i comuni il cui nome comincia con la lettera P è identificato dalla condizione

```
DES_COMUNE LIKE 'P%'
```

Eseguendo la query:

```
WTO >select * from comuni
  2  where des_comune like 'P%';

COD_ DES_COMUNE                                        PR
---- ------------------------------------------------- --
G811 POMEZIA                                           RM
G686 PIOLTELLO                                         MI
G964 POZZUOLI                                          NA
G902 PORTICI                                           NA
```

L'insieme di tutti i comuni che si trovano in provincia di Napoli è identificato dalla condizione

```
PROVINCIA = 'NA'
```

Eseguendo la query:

```
WTO >select * from comuni
  2  where provincia = 'NA';
COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
F839 NAPOLI                                              NA
G964 POZZUOLI                                            NA
G902 PORTICI                                             NA
```

L'insieme dei comuni che si trovano in provincia di Napoli ed iniziano per P, intersezione insiemistica dei due insiemi precedenti, si ottiene con la congiunzione delle due condizioni.

```
WTO >select * from comuni
  2  where provincia = 'NA'
  3  and des_comune like 'P%';

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
G964 POZZUOLI                                            NA
G902 PORTICI                                             NA
```

- Disgiunzione (OR)

La disgiunzione di due condizioni consente di estrarre tutti i record che rispettano o l'una o l'altra oppure entrambe le condizioni. È equivalente all'operazione insiemistica di UNIONE.

Nell'esempio precedente la disgiunzione delle due condizioni estrae tutti i comuni che si trovano in provincia di Napoli oppure iniziano per P (o che verificano entrambe le condizioni).

```
WTO >select * from comuni
  2  where provincia = 'NA'
  3  or des_comune like 'P%';

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
G811 POMEZIA                                             RM
G686 PIOLTELLO                                           MI
F839 NAPOLI                                              NA
G964 POZZUOLI                                            NA
G902 PORTICI                                             NA
```

- Negazione (NOT)

La negazione di una condizione consente di estrarre tutti i record che non rispettano la condizione data. È equivalente all'operazione insiemistica di COMPLEMENTO.

Nell'esempio precedente possiamo estrarre tutti i comuni che non si trovano in provincia di Napoli con la condizione

```
WTO >select * from comuni
  2  where not provincia = 'NA';

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
H501 ROMA                                                RM
```

```
G811 POMEZIA                                             RM
M297 FIUMICINO                                           RM
F205 MILANO                                              MI
G686 PIOLTELLO                                           MI
E415 LAINATE                                             MI

Selezionate 6 righe.
```

Che è equivalente perfettamente alla condizione

```
WTO >select * from comuni
  2  where provincia != 'NA';

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
H501 ROMA                                                RM
G811 POMEZIA                                             RM
M297 FIUMICINO                                           RM
F205 MILANO                                              MI
G686 PIOLTELLO                                           MI
E415 LAINATE                                             MI

Selezionate 6 righe.
```

Anche l'operatore di negazione, come visto già con gli operatori di confronto, è soggetto a comportamenti che potrebbero sembrare anomali quando sono coinvolti valori NULL.

Se, ad esempio, si estraggono i clienti aventi codice fiscale che comincia per R.

```
WTO >select * from clienti
  2  where cod_fisc like 'R%';

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
```

E poi si applica la negazione richiedendo i clienti aventi codice fiscale che non comincia per R.

```
WTO >select * from clienti
  2  where not cod_fisc like 'R%';

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
```

Si perde traccia dei clienti aventi codici fiscali NULL.

```
WTO >select * from clienti
  2  where cod_fisc is null;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964
```

Alla negazione di una congiunzione o di una disgiunzione si applicano le leggi di De Morgan della teoria degli insiemi.

```
NOT (<prima condizione> AND <seconda condizione>)
È equivalente a
NOT <prima condizione> OR NOT <seconda condizione>
```

e

```
NOT (<prima condizione> OR <seconda condizione>)
È equivalente a
NOT <prima condizione> AND NOT <seconda condizione>
```

Negli esempi precedenti, sono equivalenti le seguenti tre condizioni

```
WTO >select * from comuni
  2  where not (provincia = 'NA' and des_comune like 'P%');

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
H501 ROMA                                                RM
G811 POMEZIA                                             RM
M297 FIUMICINO                                           RM
F205 MILANO                                              MI
G686 PIOLTELLO                                           MI
E415 LAINATE                                             MI
F839 NAPOLI                                              NA

Selezionate 7 righe.

WTO >select * from comuni
  2  where not provincia = 'NA' or not des_comune like 'P%';

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
H501 ROMA                                                RM
G811 POMEZIA                                             RM
M297 FIUMICINO                                           RM
F205 MILANO                                              MI
G686 PIOLTELLO                                           MI
E415 LAINATE                                             MI
F839 NAPOLI                                              NA

Selezionate 7 righe.

WTO >select * from comuni
  2  where provincia != 'NA' or des_comune not like 'P%';

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
H501 ROMA                                                RM
G811 POMEZIA                                             RM
M297 FIUMICINO                                           RM
F205 MILANO                                              MI
G686 PIOLTELLO                                           MI
E415 LAINATE                                             MI
F839 NAPOLI                                              NA

Selezionate 7 righe.
```

Come sono equivalenti le seguenti tre.

```
WTO >select * from comuni
```

```
  2  where not (provincia = 'NA' or des_comune like 'P%');

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
H501 ROMA                                               RM
M297 FIUMICINO                                          RM
F205 MILANO                                             MI
E415 LAINATE                                            MI

WTO >select * from comuni
  2  where not provincia = 'NA' and not des_comune like 'P%'

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
H501 ROMA                                               RM
M297 FIUMICINO                                          RM
F205 MILANO                                             MI
E415 LAINATE                                            MI

WTO >select * from comuni
  2  where provincia != 'NA' and des_comune not like 'P%';

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
H501 ROMA                                               RM
M297 FIUMICINO                                          RM
F205 MILANO                                             MI
E415 LAINATE                                            MI
```

Anche gli operatori logici, come quelli aritmetici, rispettano un criterio di precedenza. Innanzi tutto vengono risolte le negazioni (NOT), successivamente le congiunzioni (AND) e solo per ultime le disgiunzioni (OR).

Anche in questo caso possono essere utilizzate le parentesi tonde per forzare tali regole di precedenza.

Nei due esempi che seguono, che sono equivalenti data la precedenza dell'AND rispetto all'OR, si richiedono i comuni che si trovano in provincia di Napoli oppure i comuni che, oltre a trovarsi in provincia di Milano, iniziano per P.

```
WTO >select * from comuni
  2  where provincia='NA'
  3  or provincia='MI'
  4  and des_comune like 'P%';

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
G686 PIOLTELLO                                          MI
F839 NAPOLI                                             NA
G964 POZZUOLI                                           NA
G902 PORTICI                                            NA

WTO >select * from comuni
  2  where provincia='NA'
  3  or (provincia='MI'
  4  and des_comune like 'P%');

COD_ DES_COMUNE                                          PR
---- -------------------------------------------------- --
```

```
G686 PIOLTELLO                                          MI
F839 NAPOLI                                             NA
G964 POZZUOLI                                           NA
G902 PORTICI                                            NA
```

Nell'esempio seguente, invece, forzando la precedenza dell'OR con le parentesi, si ottengono i comuni che cominciano per P e che si trovano in provincia di Napoli o Milano.

```
WTO >select * from comuni
  2  where (provincia='NA'
  3  or provincia='MI')
  4  and des_comune like 'P%';

COD_ DES_COMUNE                                         PR
---- -------------------------------------------------- --
G686 PIOLTELLO                                          MI
G964 POZZUOLI                                           NA
G902 PORTICI                                            NA
```

7.4.5 Ordinamento

Per ordinare i record estratti da una query è necessario aggiungere una clausola ORDER BY. In assenza di tale clausola Oracle non garantisce alcun tipo di ordinamento. A prima vista può sembrare che, in assenza della clausola ORDER BY, le righe della tabella siano restituite nello stesso ordine in cui sono state inserite. Non è vero.

La forma più semplice della clausola ORDER BY prevede la seguente sintassi:

```
ORDER BY <discriminante>, <discriminante>, ..., <discriminante>
```

Si possono dunque specificare diverse discriminanti, Oracle ordinerà in base alla prima, poi a parità della prima discriminante ordinerà in base alla seconda e così via.

Le discriminanti di ordinamento valide rientrano nelle seguenti categorie:

- ✓ Nomi di colonna
- ✓ Espressioni che coinvolgono una o più colonne, operatori aritmetici e funzioni di manipolazione (predefinite o definite dall'utente).
- ✓ Alias di colonna
- ✓ Numero d'ordine della colonna nella lista specificata dopo la parola chiave SELECT.

Per iniziare con gli esempi ordiniamo i clienti in base al nome di battesimo.

```
WTO >select * from clienti
  2  order by nome;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964

Selezionate 8 righe.
```

Nell’esempio seguente si ordina per comune e, a parità di comune, per cognome.

```
WTO >select * from clienti
  2  order by comune, cognome;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501

Selezionate 8 righe.
```

La colonna su cui si ordina non deve necessariamente essere estratta dalla query, nell’esempio seguente si ordina su comune e nome ma nessuna delle due colonne è estratta.

```
WTO >select cognome, indirizzo
  2  from clienti
  3  order by comune, nome;

COGNOME  INDIRIZZO
-------- ----------------------
COLOMBO  PIAZZA DUOMO, 1
NERI     VIA TORINO, 30
ESPOSITO VIA CARACCIOLO, 100
RUSSO    VIA GIULIO CESARE, 119
VERDI    VIA DEL MARE, 8
AMATO    VIA NAPOLI, 234
BIANCHI  VIA OSTIENSE, 850
ROSSI    VIA LAURENTINA, 700

Selezionate 8 righe.
```

Come detto la discriminante può essere anche ottenuta manipolando le colonne, nell’esempio seguente si ordina sulla somma di CAP e codice cliente.

```
WTO >select cognome, indirizzo
  2  from clienti
```

```
  3  order by cap+cod_cliente;

COGNOME  INDIRIZZO
-------- ----------------------
VERDI    VIA DEL MARE, 8
ROSSI    VIA LAURENTINA, 700
BIANCHI  VIA OSTIENSE, 850
COLOMBO  PIAZZA DUOMO, 1
NERI     VIA TORINO, 30
AMATO    VIA NAPOLI, 234
ESPOSITO VIA CARACCIOLO, 100
RUSSO    VIA GIULIO CESARE, 119
Selezionate 8 righe.
```

Un'altra possibilità è l'ordinamento in base alla posizione della colonna. Nelle esempio seguente si ordina sulla seconda colonna, cioè sul nome di battesimo.

```
WTO >select *
  2  from clienti
  3  order by 2;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964
Selezionate 8 righe.
```

C'è anche la possibilità di ordinare in base all'alias della colonna.

```
WTO >select nome n, cognome c, indirizzo
  2  from clienti
  3  order by c;

N                          C                       INDIRIZZO
-------------------------- ----------------------- ----------------------
VINCENZO                   AMATO                   VIA NAPOLI, 234
GIOVANNI                   BIANCHI                 VIA OSTIENSE, 850
AMBROGIO                   COLOMBO                 PIAZZA DUOMO, 1
GENNARO                    ESPOSITO                VIA CARACCIOLO, 100
LUCA                       NERI                    VIA TORINO, 30
MARCO                      ROSSI                   VIA LAURENTINA, 700
PASQUALE                   RUSSO                   VIA GIULIO CESARE, 119
MATTEO                     VERDI                   VIA DEL MARE, 8
Selezionate 8 righe.
```

Per ogni discriminante può opzionalmente essere indicato se l'ordinamento deve avvenire in direzione crescente o decrescente aggiungendo la parola chiave ASC o DESC. Per default l'ordinamento è crescente.

Nell'esempio seguente si ordina in senso decrescente sul cognome.

```
WTO >select * from clienti
  2  order by cognome desc;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
```

```
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964
Selezionate 8 righe.
```

Ovviamente si può ordinare su una discriminante in senso decrescente ma su un'altra in senso crescente. Nell'esempio seguente si ordina sul comune decrescente e poi sul nome in senso crescente.

```
WTO >select * from clienti
  2  order by comune desc, nome;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
Selezionate 8 righe.
```

Per ogni discriminante si può indicare se i NULL devono essere ordinati all'inizio oppure alla fine della lista con le parole chiave NULLS FIRST o NULLS LAST.

Nell'esempio seguente si ordina sul codice fiscale. Senza ulteriori specifiche i NULL finiscono alla fine, specificando NULLS FIRST vengono estratti per primi.

```
WTO >select * from clienti
  2  order by cod_fisc;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205

Selezionate 8 righe.

WTO >select * from clienti
  2  order by cod_fisc nulls first;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811

Selezionate 8 righe.
```

7.4.6 Raggruppamenti

Alcune volte è necessario estrarre le righe da una tabella raggruppandole secondo un determinato criterio.

Ipotizziamo ad esempio di voler conoscere tutti i comuni in cui è residente almeno uno dei nostri clienti, banalmente la prima query che viene in mente è

```
WTO >select comune from clienti;

COMU
----
H501
H501
G811
F205
F205
F839
F839
G964
Selezionate 8 righe.
```

Il problema di questa query è che estrae un numero di record pari ai clienti presenti in tabella, in alcuni comuni ci sono più clienti e questi vengono estratti più volte. Abbiamo quindi bisogno di raggruppare il risultato per comune.

Il modo più semplice per estrarre una sola riga per ogni comune è utilizzare la clausola DISTINCT.

```
WTO >select distinct comune from clienti;

COMU
----
F205
F839
G811
G964
H501
```

Questa clausola comprime l'output della query eliminando le righe duplicate. La clausola DISTINCT può essere applicata anche a più di una colonna, nell'esempio seguente le fatture vengono accorpate per data ed importo:

```
WTO >select * from fatture;
NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S

WTO >select distinct data_fattura, importo
  2  from fatture;

DATA_FATT    IMPORTO
--------- ----------
01-OTT-10        300
```

```
20-OTT-10        700
01-DIC-10        500
01-FEB-11       1000
```

Di conseguenza le due righe relative a fatture da 500 euro del 01/10/2010 sono state collassate in una sola.

La clausola DISTINCT non può essere applicata solo ad una parte delle colonne estratte dalla query. La parola chiave deve seguire immediatamente la parola chiave SELECT ed essere seguita da tutte le colonne.

```
WTO >select num_ordine, distinct data_fattura, importo
  2  from fatture;
select num_ordine, distinct data_fattura, importo
                        *
ERRORE alla riga 1:
ORA-00936: missing expression
```

Un altro modo per raggruppare il risultato di una query è utilizzare la clausola GROUP BY.

Questa clausola si aggiunge alla fine dell'istruzione, dopo la WHERE, e richiede di specificare l'elenco delle colonne su cui si intende raggruppare. Segue quindi la sintassi

```
SELECT <colonna>, <colonna>,...<colonna>
  FROM <nome tabella>
 WHERE <condizione>
GROUP BY <colonna>, <colonna>,...<colonna>
```

Per ottenere, dunque, i comuni in cui sono residenti i clienti è possibile anche scrivere.

```
WTO >select comune
  2  from clienti
  3  group by comune;

COMU
----
F205
F839
G811
G964
H501
```

Il vantaggio di utilizzare la clausola GROUP BY invece della DISTINCT consiste nella possibilità di ricorrere alle funzioni di colonna o di gruppo.

Una funzione di gruppo è una funzione che si applica ad un gruppo di dati accorpati con la GROUP BY e restituisce un valore. La funzione di gruppo può essere applicata anche in assenza della clausola GROUP BY. In tal caso la funzione sarà applicata all'unico gruppo costituito da tutte le righe estratte dalla query.

Oracle mette a disposizione le seguenti funzioni di gruppo:

- SUM

Somma un dato numerico presente nelle righe che compongono il gruppo. Ipotizziamo ad esempio di voler sommare gli importi di tutte le fatture. Poiché ci interessa creare un unico gruppo composto da tutte le fatture dobbiamo omettere del tutto la clausola GROUP BY:

```
WTO >select sum(importo) from fatture;

SUM(IMPORTO)
------------
        3000
```

Se invece vogliamo ottenere la somma delle fatture per ogni data avremo la necessità di raggruppare per data fattura

```
WTO >select data_fattura, sum(importo)
  2  from fatture
  3  group by data_fattura;

DATA_FATT SUM(IMPORTO)
--------- ------------
01-OTT-10          300
20-OTT-10          700
01-DIC-10         1000
01-FEB-11         1000
```

Avremmo potuto anche non estrarre la data fattura

```
WTO >select  sum(importo)
  2  from fatture
  3  group by data_fattura;

SUM(IMPORTO)
------------
         300
         700
        1000
        1000
```

Sarebbe però stato difficile associare le righe estratte alle date.

- MIN

Estrae il valore minimo di un dato presente nelle righe che compongono il gruppo. Ad esempio estraiamo il minimo importo di una fattura.

```
WTO >select  min(importo)
  2  from fatture;

MIN(IMPORTO)
------------
         300

```

Mentre il minimo importo di fattura per ogni ordine si estrae come segue:

```
WTO >select num_ordine, min(importo)
  2  from fatture
  3  group by num_ordine;

NUM_ORDINE MIN(IMPORTO)
---------- ------------
```

```
         1          300
         2          700
         3         1000
         5          500
```

- MAX

Estrae il valore massimo di un dato presente nelle righe che compongono il gruppo. Si comporta come la MIN.

```
WTO >select max(importo)
  2  from fatture;

MAX(IMPORTO)
------------
        1000

WTO >select num_ordine, max(importo)
  2  from fatture
  3  group by num_ordine;

NUM_ORDINE MAX(IMPORTO)
---------- ------------
         1          500
         2          700
         3         1000
         5          500
```

- AVG

Calcola il valore medio di un dato presente nelle righe che compongono il gruppo.

```
WTO >select avg(importo)
  2  from fatture;

AVG(IMPORTO)
------------
         600

WTO >select num_ordine, avg(importo)
  2  from fatture
  3  group by num_ordine;

NUM_ORDINE AVG(IMPORTO)
---------- ------------
         1          400
         2          700
         3         1000
         5          500
```

- VARIANCE

Calcola la varianza di un dato presente nelle righe che compongono il gruppo.

```
WTO >select variance(importo)
  2  from fatture;

VARIANCE(IMPORTO)
-----------------
            70000
```

```
WTO >select num_ordine, variance(importo)
  2  from fatture
  3  group by num_ordine;

NUM_ORDINE VARIANCE(IMPORTO)
---------- -----------------
         1             20000
         2                 0
         3                 0
         5                 0

WTO >select stddev(importo)
  2  from fatture;
```

- STDDEV

Calcola la deviazione standard di un dato presente nelle righe che compongono il gruppo

```
STDDEV(IMPORTO)
---------------
     264,575131

WTO >select num_ordine, stddev(importo)
  2  from fatture
  3  group by num_ordine;

NUM_ORDINE STDDEV(IMPORTO)
---------- ---------------
         1      141,421356
         2               0
         3               0
         5               0
```

- COUNT

Conta le righe presenti nel gruppo. È l'unica funzione di gruppo che può prendere in input un singolo dato oppure il carattere jolly *.

Se la funzione COUNT riceve in input un dato conta solo le righe che hanno quel dato valorizzato, non NULL. Se invece riceve in input * conta tutte le righe presenti nel gruppo a prescindere dal fatto che i valori siano NULL oppure no.

Per contare tutte le righe presenti in una tabella si utilizza dunque la sintassi

```
SELECT COUNT(*) FROM <nome tabella>
```

Ad esempio per contare le righe presenti nella tabella CLIENTI

```
WTO >select count(*) from clienti;

  COUNT(*)
----------
         8
```

Se alla COUNT si dà in input la colonna NOME il risultato è lo stesso, perché nella colonna NOME non ci sono dati nulli

```
WTO >select count(nome) from clienti;
```

```
COUNT(NOME)
-----------
          8
```

Se invece si passa in input una colonna che non è sempre valorizzata, come il codice fiscale, si ottiene un risultato diverso.

```
WTO >select count(cod_fisc) from clienti;

COUNT(COD_FISC)
---------------
              5
```

Per sapere quante fatture ci sono per ciascun ordine si deve raggruppare sul numero dell'ordine

```
WTO >select num_ordine, count(*)
  2  from fatture
  3  group by num_ordine;

NUM_ORDINE   COUNT(*)
---------- ----------
         1          2
         2          1
         3          1
         5          1
```

In una query in cui è presente la clausola GROUP BY bisogna rispettare la regola seguente: tutte le espressioni presenti nell'elenco delle colonne che non sono costanti e non sono funzioni di gruppo devono essere inserite anche nella clausola GROUP BY.

Se, infatti, si cerca di eseguire la query

```
WTO >select nome, comune
  2  from clienti
  3  group by comune;
select nome, comune
       *
ERRORE alla riga 1:
ORA-00979: not a GROUP BY expression
```

Si ottiene un errore. Una volta che il risultato della query è stato diviso in gruppi per comune è necessario, per ogni gruppo, estrarre un valore del nome. Ciò ovviamente non è possibile perché in ogni gruppo ci possono essere nomi differenti ed Oracle non saprebbe quale estrarre.

In una query in cui è presente la clausola GROUP BY Oracle segue il seguente flusso. Innanzi tutto applica la clausola WHERE e scarta le righe che non la rispettano.

Successivamente raggruppa le righe rispetto alle colonne indicate nella clausola GROUP BY e calcola le funzioni di gruppo. In questo modo ottiene le righe accorpate.

A questo punto potrebbe essere necessario escludere alcune righe accorpate. Ciò non può essere fatto con la clausola WHERE poiché, come detto, questa viene applicata prima del raggruppamento.

Per questa esigenza esiste la clausola HAVING. Questa clausola accetta una condizione che viene applicata alle righe accorpate al fine di escludere quelle che non la verificano.

Ipotizziamo ad esempio di voler estrarre le fatture accorpate per numero ordine escludendo gli ordini che hanno meno di due fatture.

Innanzi tutto è necessario utilizzare una clausola GROUP BY per numero ordine, successivamente bisogna escludere con una HAVING le righe che hanno COUNT(*) minore di due.

```
WTO >select num_ordine, sum(importo), count(*)
  2  from fatture
  3  group by num_ordine
  4  having count(*)>=2;

NUM_ORDINE SUM(IMPORTO)   COUNT(*)
---------- ------------ ----------
         1          800          2
```

Solo l'ordine numero 1 ha almeno due fatture.

Nella stessa maniera, se volessimo estrarre gli ordini che hanno un importo totale delle fatture emesse maggiore di 700 euro, a prescindere dal numero di fatture potremmo fare:

```
WTO >select num_ordine, sum(importo), count(*)
  2  from fatture
  3  group by num_ordine
  4  having sum(importo)>700;

NUM_ORDINE SUM(IMPORTO)   COUNT(*)
---------- ------------ ----------
         1          800          2
         3         1000          1
```

L'ordine numero 1 arriva ad 800 euro con due fatture mentre l'ordine numero 3 arriva a 1000 euro con una sola fattura.

La condizione della HAVING può essere composta con gli operatori logici e gli operatori di confronto esattamente come la condizione utilizzata nella clausola WHERE. Ad esempio potremmo estratte tutti gli ordini che hanno almeno due fatture oppure una sola fattura di importo almeno 700 euro col comando:

```
WTO >select num_ordine, sum(importo), count(*)
  2  from fatture
  3  group by num_ordine
  4  having count(*)>=2
  5  or (count(*)=1 and sum(importo) >=700);

NUM_ORDINE SUM(IMPORTO)   COUNT(*)
---------- ------------ ----------
         1          800          2
         2          700          1
         3         1000          1
```

La clausola ROLLUP è una funzionalità aggiuntiva alla GROUP BY.

Questa funzionalità consente di ottenere totali parziali all'interno della query, facciamo un esempio.

Ci baseremo sulla tabella EMP dell'utente SCOTT:

```
WTO> desc emp
 Nome                       Nullo?   Tipo
 ------------------------ -------- ----------------
 EMPNO                     NOT NULL NUMBER(4)
 ENAME                              VARCHAR2(10)
 JOB                                VARCHAR2(9)
 MGR                                NUMBER(4)
 HIREDATE                           DATE
 SAL                                NUMBER(7,2)
 COMM                               NUMBER(7,2)
 DEPTNO                             NUMBER(2)
```

Se vogliamo vedere il totale degli stipendi per ogni impiego (JOB) e per ogni dipartimento scriviamo una semplice GROUP BY:

```
WTO> select deptno, job, sum(sal)
  2  from emp
  3  group by deptno,job;

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        20 CLERK           1900
        30 SALESMAN        5600
        20 MANAGER         2975
        30 CLERK            950
        10 PRESIDENT       5000
        30 MANAGER         2850
        10 CLERK           1300
        10 MANAGER         2450
        20 ANALYST         6000

Selezionate 9 righe.
```

La clausola ROLLUP consente di aggiungere con semplicità dei subtotali. Per esempio, volendo conservare la decomposizione per dipartimento-impiego, ma volendo visualizzare totali parziali per dipartimento si può scrivere:

```
WTO> select deptno, job,sum(sal)
  2  from emp
  3  group by deptno,rollup (job);

    DEPTNO JOB         SUM(SAL)
---------- --------- ----------
        10 CLERK           1300
        10 MANAGER         2450
        10 PRESIDENT       5000
        10                 8750
        20 CLERK           1900
        20 ANALYST         6000
        20 MANAGER         2975
        20                10875
        30 CLERK            950
        30 MANAGER         2850
        30 SALESMAN        5600
        30                 9400

Selezionate 12 righe.
```

In pratica rispetto a prima abbiamo scritto nella clausola GROUP BY

```
ROLLUP(JOB)
```

invece di

```
JOB
```

ROLLUP(JOB) implica il raggruppamento su JOB ma, in più, richiede di calcolare il totale dei JOB relativo al gruppo immediatamente più esterno (DEPTNO).

Quindi al risultato della prima query vengono aggiunte tre nuove righe riportanti il totale degli stipendi per tutti i JOB di quel dipartimento.

Per aggiungere un'etichetta di testo al posto di quello spazio vuoto è possibile utilizzare la funzione NVL. Tale funzione e l'operatore di concatenazione (||) saranno trattati più avanti, ne anticipiamo qui un esempio di utilizzo.

```
WTO> select deptno, nvl(job,'    Tot. '||deptno) job,sum(sal)
  2  from emp
  3  group by deptno,rollup (job);

    DEPTNO JOB                    SUM(SAL)
---------- -------------------- ----------
        10 CLERK                      1300
        10 MANAGER                    2450
        10 PRESIDENT                  5000
        10     Tot. 10                8750
        20 CLERK                      1900
        20 ANALYST                    6000
        20 MANAGER                    2975
        20     Tot. 20               10875
        30 CLERK                       950
        30 MANAGER                    2850
        30 SALESMAN                   5600
        30     Tot. 30                9400

Selezionate 12 righe.
```

Dove

```
nvl(job,'    Tot. '||deptno)
```

Significa "se il valore di JOB è NULL scrivi 'Tot. ' e poi il numero del dipartimento, altrimenti scrivi il valore di JOB.

Che succede allora se mettiamo anche DEPTNO dentro la ROLLUP?

```
WTO> select deptno, nvl(job,'    Tot. '||deptno) job,sum(sal)
  2  from emp
  3   group by rollup (deptno,job);

    DEPTNO JOB                    SUM(SAL)
---------- -------------------- ----------
        10 CLERK                      1300
        10 MANAGER                    2450
        10 PRESIDENT                  5000
        10     Tot. 10                8750
        20 CLERK                      1900
        20 ANALYST                    6000
        20 MANAGER                    2975
```

```
        20     Tot. 20              10875
        30 CLERK                      950
        30 MANAGER                   2850
        30 SALESMAN                  5600
        30     Tot. 30               9400
               Tot.                 29025

Selezionate 13 righe.
```

Si ottiene anche il totale generale. Anche in questo caso l'etichetta può essere gestita con la funzione NVL.

```
WTO> select deptno, nvl(job,'   Tot. '||
  2  nvl(to_char(deptno),'Complessivo')) job,sum(sal)
  3  from emp
  4  group by rollup (deptno,job);

    DEPTNO JOB                    SUM(SAL)
---------- -------------------- ----------
        10 CLERK                      1300
        10 MANAGER                    2450
        10 PRESIDENT                  5000
        10    Tot. 10                 8750
        20 CLERK                      1900
        20 ANALYST                    6000
        20 MANAGER                    2975
        20    Tot. 20                10875
        30 CLERK                       950
        30 MANAGER                    2850
        30 SALESMAN                   5600
        30    Tot. 30                 9400
              Tot. Complessivo       29025

Selezionate 13 righe.
```

7.4.7 Tornando su ROWNUM

Si è già detto che, per ogni riga estratta da una query, la pseudocolonna ROWNUM restituisce un numero che indica l'ordine in cui Oracle seleziona la riga dalla tabella. La prima riga selezionata ha ROWNUM uguale a 1, la seconda 2 e così via.

L'esempio (banale) è d'obbligo:

```
WTO >select cognome, rownum
  2  from clienti;

COGNOME      ROWNUM
-------- ----------
ROSSI             1
BIANCHI           2
VERDI             3
NERI              4
COLOMBO           5
ESPOSITO          6
RUSSO             7
AMATO             8
```

```
Selezionate 8 righe.
```

I malintesi cominciano quando si utilizza la pseudocolonna nella condizione di WHERE. Mentre, infatti, il seguente risultato è atteso:

```
WTO >select cognome, rownum
  2  from clienti
  3  where rownum<4;

COGNOME      ROWNUM
-------- ----------
ROSSI             1
BIANCHI           2
VERDI             3
```

Quest'altro spesso sorprende:

```
WTO >select cognome, rownum
  2  from clienti
  3  where rownum<4
  4  order by cognome;

COGNOME      ROWNUM
-------- ----------
BIANCHI           2
ROSSI             1
VERDI             3
```

Molti infatti immaginano che la ROWNUM possa essere utilizzata in questo modo per ottenere una "TOP-N query", cioè l'estrazione dei primi N record di una tabella secondo un particolare ordine. Secondo tale approccio ci si aspetterebbe che l'istruzione precedente torni i primi tre record di CLIENTI in ordine di COGNOME.

Ma non è così. Come si vede, infatti, la clausola di ordinamento è stata applicata solo dopo l'applicazione della ROWNUM.

La query precedente restituisce dunque tre record di CLIENTI presi a caso e poi ordinati per COGNOME.

Per estrarre i primi N record di una tabella in un particolare ordine è necessario ricorrere ad una subquery oppure ad una funzione analitica. Entrambi questi temi saranno trattati più avanti.

La pseudo colonna ROWNUM ha senso in una WHERE solo se confrontata mediante gli operatori "minore" e "minore o uguale".

L'unica uguaglianza che ha senso con ROWNUM è "ROWNUM=1".

Ma perché la query

```
WTO >select cognome, rownum
  2  from clienti
  3  where rownum>3;

Nessuna riga selezionata
```

non estrae record?

Per definizione il primo record estratto ha ROWNUM=1, esso dunque non verifica la condizione e viene scartato. Tocca al secondo record che assume anch'esso ROWNUM=1 (visto che il precedente è stato scartato) e dunque anch'esso viene scartato. Così via per tutti i restanti record.

Hanno invece senso le seguenti:

```
WTO >select cognome, rownum
  2  from clienti
  3  where rownum=1;

COGNOME      ROWNUM
-------- ----------
ROSSI             1

WTO >select cognome, rownum
  2  from clienti
  3  where rownum<4;

COGNOME      ROWNUM
-------- ----------
ROSSI             1
BIANCHI           2
VERDI             3

WTO >select cognome, rownum
  2  from clienti
  3  where rownum<=4;

COGNOME      ROWNUM
-------- ----------
ROSSI             1
BIANCHI           2
VERDI             3
NERI              4
```

7.4.8 Le clausole PIVOT ed UNPIVOT

Quest'argomento è considerato di livello avanzato. Per comprendere pienamente gli esempi è necessario conoscere argomenti di SQL che in questa fase del manuale ancora non sono stati introdotti. Il lettore principiante può saltare questo paragrafo e tornarci alla fine del libro quando avrà acquisito gli strumenti necessari ad affrontarlo.

Uno dei problemi più comuni che ci troviamo ad affrontare in SQL è la necessità di visualizzare per colonne dati che sono contenuti in diverse righe di una tabella. In passato abbiamo risolto questo problema con delle UNION, ma a partire dalla versione 11g di Oracle c'è un nuovo strumento: la clausola PIVOT dell'istruzione di SELECT.

Un esempio può spiegare molto meglio di mille parole: la tabella EMP dei dipendenti (schema scott/tiger) contiene un record per ogni dipendente. Tra i vari attributi sono presenti le colonne ENAME (nome del dipendente), DEPTNO (codice del dipartimento) e SAL (stipendio). Ci proponiamo di estrarre un report a quattro colonne: nella prima vogliamo il nome dei dipendente, nella seconda gli stipendi dei dipendenti appartenenti al dipartimento 10, nella terza di quelli appartenenti al dipartimento 20 e nella

quarta lo stipendio dei dipendenti del dipartimento 30. Fino ad Oracle 10g avremmo risolto con una UNION:

```
SQL> select ename, sal d10, null d20, null d30
  2    from emp where deptno=10
  3    union
  4    select ename, null d10, sal d20, null d30
  5    from emp where deptno=20
  6    union
  7    select ename, null d10, null d20, sal d30
  8    from emp where deptno=30;

ENAME             D10        D20        D30
---------- ---------- ---------- ----------
ADAMS                       1100
ALLEN                                  1600
BLAKE                                  2850
CLARK            2450
FORD                        3000
JAMES                                   950
JONES                       2975
KING             5000
MARTIN                                 1250
MILLER           1300
SCOTT                       3000
SMITH                        800
TURNER                                 1500
WARD                                   1250
```

Dalla versione 11g, invece, abbiamo a disposizione l'operatore PIVOT. Quest'operatore ci consente di definire "al volo" nuove colonne specificando i valori del campo che fa da filtro (DEPTNO nel nostro caso) il campo da totalizzare (SAL per noi) e la funzione di gruppo da utilizzare per la totalizzazione (SUM):

```
SQL> select ename, d10,d20,d30 from emp
  2  pivot (sum(sal) for deptno in
  3           (10 as D10, 20 as d20, 30 as d30));

ENAME             D10        D20        D30
---------- ---------- ---------- ----------
MARTIN                                 1250
BLAKE                                  2850
MILLER           1300
WARD                                   1250
JONES                       2975
TURNER                                 1500
JAMES                                   950
KING             5000
ADAMS                       1100
FORD                        3000
CLARK            2450
SCOTT                       3000
SMITH                        800
ALLEN                                  1600
```

Ovviamente una volta ottenuto questo risultato di partenza possiamo complicare a piacere le cose per derivare altri risultati.

Alcuni esempi:

La somma degli stipendi per dipartimento (con i risultati in colonne distinte):

```
SQL> select sum(d10),sum(d20),sum(d30) from emp
  2  pivot (sum(sal) for deptno
  3          in (10 as D10, 20 as d20, 30 as d30));

  SUM(D10)   SUM(D20)   SUM(D30)
---------- ---------- ----------
      8750      10875       9400
```

La somma degli stipendi per dipartimento (sulle colonne) e per impiego (sulle righe):

```
SQL> select job, sum(d10),sum(d20),sum(d30) from emp
  2  pivot (sum(sal) for deptno
  3           in (10 as D10, 20 as d20, 30 as d30))
  4  group by job;

JOB         SUM(D10)   SUM(D20)   SUM(D30)
--------- ---------- ---------- ----------
SALESMAN                              5600
CLERK           1300       1900        950
PRESIDENT       5000
MANAGER         2450       2975       2850
ANALYST                    6000
```

La clausola UNPIVOT serve a fare esattamente il contrario della PIVOT, cioè a leggere in diversi record i dati che in partenza abbiamo su diverse colonne.

Ipotizziamo ad esempio di avere la tabella VENDITE:

```
WTO> select * from vendite;

    DEPTNO  ANNO_2008  ANNO_2009  ANNO_2010
---------- ---------- ---------- ----------
        10     350000     400000     500000
        20     185000     220000     330000
        30     400000     500000     600000
```

e di voler ottenere nove record, uno per ogni dipartimento e per ogni anno.

```
WTO> select *
  2  from vendite
  3  UNPIVOT
  4  (importo for anno in (
  5           anno_2008 as '2008',
  6           anno_2009 as '2009',
  7           anno_2010 as '2010')
  8  );

    DEPTNO ANNO    IMPORTO
---------- ---- ----------
        10 2008     350000
        10 2009     400000
        10 2010     500000
        20 2008     185000
        20 2009     220000
        20 2010     330000
        30 2008     400000
        30 2009     500000
```

```
        30 2010     600000
9 rows selected.
```

Guardiamo la clausola UNPIVOT. Abbiamo chiesto un campo importo per anno indicando nella clausola IN come trascodificare l'header della colonna in un valore.

Facciamo una piccola modifica ai dati:

```
WTO> update vendite set anno_2010 = null where deptno=20;

1 row updated.

WTO> select * from vendite;

    DEPTNO  ANNO_2008  ANNO_2009  ANNO_2010
---------- ---------- ---------- ----------
        10     350000     400000     500000
        20     185000     220000
        30     400000     500000     600000
```

e ripetiamo la stessa query di prima:

```
WTO> select *
  2  from vendite
  3  UNPIVOT
  4  (importo for anno in (
  5           anno_2008 as 2008,
  6           anno_2009 as 2009,
  7           anno_2010 as 2010)
  8  );

    DEPTNO ANNO    IMPORTO
---------- ---- ----------
        10 2008     350000
        10 2009     400000
        10 2010     500000
        20 2008     185000
        20 2009     220000
        30 2008     400000
        30 2009     500000
        30 2010     600000

8 rows selected.
```

Le righe selezionate sono solo otto perché il valore nullo è stato ignorato.

Se vogliamo che i valori nulli siano invece presi in considerazione basta chiederlo con l'aggiunta dell'opzione "INCLUDE NULLS":

```
WTO> select *
  2  from vendite
  3  UNPIVOT INCLUDE NULLS
  4  (importo for anno in (
  5           anno_2008 as 2008,
  6           anno_2009 as 2009,
  7           anno_2010 as 2010)
  8  );

    DEPTNO ANNO    IMPORTO
---------- ---- ----------
        10 2008     350000
```

```
        10 2009      400000
        10 2010      500000
        20 2008      185000
        20 2009      220000
        20 2010
        30 2008      400000
        30 2009      500000
        30 2010      600000

9 rows selected.
```

Ovviamente questi dati possono essere successivamente manipolati a piacere, ad esempio possiamo calcolare al volo la media annua delle vendite:

```
WTO> select deptno, min(anno), max(anno), avg(importo)
  2  from vendite
  3  UNPIVOT INCLUDE NULLS
  4  (importo for anno in (
  5           anno_2008 as 2008,
  6           anno_2009 as 2009,
  7           anno_2010 as 2010)
  8  )
  9  group by deptno;

    DEPTNO  MIN(ANNO)  MAX(ANNO) AVG(IMPORTO)
---------- ---------- ---------- ------------
        30       2008       2010       500000
        20       2008       2010       202500
        10       2008       2010   416666.667
```

7.4.9 Query gerarchiche

Quest'argomento è considerato di livello avanzato. Per comprendere pienamente gli esempi è necessario conoscere argomenti di SQL che in questa fase del manuale ancora non sono stati introdotti. Il lettore principiante può saltare questo paragrafo e tornarci alla fine del libro quando avrà acquisito gli strumenti necessari ad affrontarlo.

Le query gerarchiche sono una funzionalità dell'SQL che a prima vista può sembrare di scarsa utilità pratica.

Non è così, la clausola CONNECT BY può essere spesso utilizzata per risolvere problemi che a prima vista non sembrano per niente "gerarchici", ad esempio l'anagramma di una parola oppure l'aggregazione di stringhe.

Ma andiamo con ordine, partiremo dai casi più semplici di query gerarchiche per arrivare a questi un po' più articolati alla fine.

Sfrutteremo la tabella EMP dello schema SCOTT che è strutturata già con una relazione gerarchica. Su questa tabella, infatti, c'è un campo MGR che rappresenta il numero di matricola del capo di ogni dipendente, realizzando così una gerarchia tra i record.

I dati in tabella sono i seguenti:

```
WTO> select empno, ename, mgr
```

```
  2  from emp;

     EMPNO ENAME             MGR
---------- ---------- ----------
      7369 SMITH            7902
      7499 ALLEN            7698
      7521 WARD             7698
      7566 JONES            7839
      7654 MARTIN           7698
      7698 BLAKE            7839
      7782 CLARK            7839
      7788 SCOTT            7566
      7839 KING
      7844 TURNER           7698
      7876 ADAMS            7788
      7900 JAMES            7698
      7902 FORD             7566
      7934 MILLER           7782
```

Come si vede il capo di tutti è KING che non ha un capo.

Il primo esempio di query gerarchica è il seguente:

```
WTO> select empno, ename, mgr, prior ename, level
  2  from emp
  3  connect by prior empno = mgr
  4  start with mgr is null;

     EMPNO ENAME             MGR PRIORENAME      LEVEL
---------- ---------- ---------- ---------- ----------
      7839 KING                                      1
      7566 JONES            7839 KING                2
      7788 SCOTT            7566 JONES               3
      7876 ADAMS            7788 SCOTT               4
      7902 FORD             7566 JONES               3
      7369 SMITH            7902 FORD                4
      7698 BLAKE            7839 KING                2
      7499 ALLEN            7698 BLAKE               3
      7521 WARD             7698 BLAKE               3
      7654 MARTIN           7698 BLAKE               3
      7844 TURNER           7698 BLAKE               3
      7900 JAMES            7698 BLAKE               3
      7782 CLARK            7839 KING                2
      7934 MILLER           7782 CLARK               3
```

Analizziamola: la clausola CONNECT BY, indispensabile per realizzare una query gerarchica, ci dice come ogni record è collegato a quello gerarchicamente superiore.

Il padre dei record che hanno MGR=x ha EMPNO=x.

Viceversa dato un record con codice EMPNO=x, tutti i record che hanno MGR=x sono suoi figli.

L'operatore unario PRIOR indica "il padre di".

La clausola START WITH serve ad indicare con quali record cominciare la gerarchia, nel nostro caso vogliamo cominciare la gerarchia con il capo di tutti, cioè il record che ha MGR NULL.

Non è necessario che ci sia un'unica radice dell'albero.

La pseudo colonna LEVEL indica a che livello si trova ogni record nella gerarchia, partendo dal presupposto che le radici dell'albero hanno LEVEL=1.

Detto questo possiamo leggere il nostro albero nel risultato della query appena eseguita: KING è padre di tutti ed ha livello 1.

Sotto di lui ci sono tre dipendenti a livello 2 (JONES,BLAKE e CLARK). Di seguito tutti gli altri.

Come viene realizzata la gerarchia? Prima di tutto si leggono i record. Poi si determinano le radici applicando la clausola START WITH. Successivamente a partire da ogni radice si determinano i figli di primo livello applicando la CONNECT BY e così via per ogni figlio.

Per migliorare un po' la visualizzazione dell'output possiamo ricorrere all'utilizzo della LPAD:

```
WTO> select empno,lpad(' ',level*3,' ')||ename nome,
  2         mgr, prior ename, level
  3    from emp
  4   connect by prior empno = mgr
  5    start with mgr is null;

     EMPNO NOME                      MGR PRIORENAME      LEVEL
---------- ------------------ ---------- ---------- ----------
      7839    KING                                           1
      7566       JONES              7839 KING                2
      7788          SCOTT           7566 JONES               3
      7876             ADAMS        7788 SCOTT               4
      7902          FORD            7566 JONES               3
      7369             SMITH        7902 FORD                4
      7698       BLAKE              7839 KING                2
      7499          ALLEN           7698 BLAKE               3
      7521          WARD            7698 BLAKE               3
      7654          MARTIN          7698 BLAKE               3
      7844          TURNER          7698 BLAKE               3
      7900          JAMES           7698 BLAKE               3
      7782       CLARK              7839 KING                2
      7934          MILLER          7782 CLARK               3
```

Abbiamo aggiunto un po' di spazi alla sinistra del nome in funzione del livello. Adesso la visualizzazione è decisamente migliore.

Come si vede l'ordinamento dei record definisce anche la gerarchia, aggiungendo una clausola ORDER BY perderemmo completamente la visione della gerarchia. C'è però la possibilità di decidere con che criterio ordinare più record che si trovano allo stesso livello (fratelli), come ad esempio JONES, BLAKE e CLARK:

```
WTO> select empno,lpad(' ',level*3,' ')||ename nome,
  2         mgr, prior ename, level
  3    from emp
  4   connect by prior empno = mgr
  5    start with mgr is null
  6   order siblings by ename;

     EMPNO NOME                      MGR PRIORENAME      LEVEL
```

```
---------- ------------------ ---------- ---------- ----------
      7839    KING                                           1
      7698       BLAKE              7839 KING                2
      7499          ALLEN           7698 BLAKE               3
      7900          JAMES           7698 BLAKE               3
      7654          MARTIN          7698 BLAKE               3
      7844          TURNER          7698 BLAKE               3
      7521          WARD            7698 BLAKE               3
      7782       CLARK              7839 KING                2
      7934          MILLER          7782 CLARK               3
      7566       JONES              7839 KING                2
      7902          FORD            7566 JONES               3
      7369             SMITH        7902 FORD                4
      7788          SCOTT           7566 JONES               3
      7876             ADAMS        7788 SCOTT               4
```

Senza rovinare la gerarchia abbiamo ordinato i fratelli per nome.

Su Oracle9i non c'è altro, ciò che segue, infatti, è presente dalla 10g in poi.

In qualunque struttura gerarchica si possono creare dei cicli infiniti. Pensate se KING fosse figlio di un altro dipendente:

```
WTO> update emp set mgr=7369 where ename='KING';
```

Abbiamo detto che KING è figlio di SMITH, che però a sua volta è pronipote di KING. Che succede, dunque, se rieseguiamo la query precedente?

```
WTO> select empno,lpad(' ',level*3,' ')||ename nome,
  2         mgr, prior ename, level
  3    from emp
  4  connect by prior empno = mgr
  5    start with mgr is null
  6  order siblings by ename;

Nessuna riga selezionata
```

Certo, perché non c'è nessun record con MGR NULL. Allora cambiamo la START WITH come segue:

```
WTO> select empno,lpad(' ',level*3,' ')||ename nome,
  2         mgr, prior ename, level
  3    from emp
  4  connect by prior empno = mgr
  5    start with empno=7839;
ERROR:
ORA-01436: CONNECT BY in loop sui dati utente

Nessuna riga selezionata
```

Ecco il loop, è impossibile creare la gerarchia.

Ma Oracle ci ha messo un pezza: la clausola NOCYCLE

```
WTO> select empno,lpad(' ',level*3,' ')||ename nome,
  2         mgr, prior ename, level
  3    from emp
  4  connect by nocycle prior empno = mgr
  5    start with empno=7839;

     EMPNO NOME                            MGR PRIORENAME      LEVEL
```

```
---------- ------------------ ---------- ---------- ----------
      7839    KING                   7369                     1
      7566       JONES               7839 KING                2
      7788          SCOTT            7566 JONES               3
      7876             ADAMS         7788 SCOTT               4
      7902          FORD             7566 JONES               3
      7369             SMITH         7902 FORD                4
      7698       BLAKE               7839 KING                2
      7499          ALLEN            7698 BLAKE               3
      7521          WARD             7698 BLAKE               3
      7654          MARTIN           7698 BLAKE               3
      7844          TURNER           7698 BLAKE               3
      7900          JAMES            7698 BLAKE               3
      7782       CLARK               7839 KING                2
      7934          MILLER           7782 CLARK               3

Selezionate 14 righe.
```

Questa clausola fa sì che appena si determina il ciclo Oracle si interrompa ma continui con gli altri rami. Nel nostro caso l'intero risultato è estratto.

In realtà dopo SMITH ci sarebbe dovuto essere di nuovo KING con tutto ciò che segue, all'infinito, ma Oracle si è accorto del ciclo e si è fermato, andando avanti con gli altri rami.

La pseudo colonna CONNECT_BY_ISCYCLE ci dice in quali record si è verificato il ciclo infinito:

```
WTO> select empno,lpad(' ',level*3,' ')||ename nome,
  2         mgr, prior ename, connect_by_iscycle ciclo
  3    from emp
  4  connect by nocycle prior empno = mgr
  5    start with empno=7839;

     EMPNO NOME                          MGR PRIORENAME      CICLO
---------- ------------------ ---------- ---------- ----------
      7839    KING                   7369                     0
      7566       JONES               7839 KING                0
      7788          SCOTT            7566 JONES               0
      7876             ADAMS         7788 SCOTT               0
      7902          FORD             7566 JONES               0
      7369             SMITH         7902 FORD                1
      7698       BLAKE               7839 KING                0
      7499          ALLEN            7698 BLAKE               0
      7521          WARD             7698 BLAKE               0
      7654          MARTIN           7698 BLAKE               0
      7844          TURNER           7698 BLAKE               0
      7900          JAMES            7698 BLAKE               0
      7782       CLARK               7839 KING                0
      7934          MILLER           7782 CLARK               0
```

Oltre a PRIOR esiste un altro operatore unario molto utile: CONNECT_BY_ROOT. Consente di visualizzare la radice di un certo record:

```
WTO> select empno,lpad(' ',level*3,' ')||ename nome,
  2         connect_by_root ename boss
  3    from emp
  4  connect by prior empno = mgr
  5    start with mgr is null;
```

```
     EMPNO NOME               BOSS
---------- ------------------ ----------
      7839    KING            KING
      7566       JONES        KING
      7788          SCOTT     KING
      7876             ADAMS  KING
      7902          FORD      KING
      7369             SMITH  KING
      7698       BLAKE        KING
      7499          ALLEN     KING
      7521          WARD      KING
      7654          MARTIN    KING
      7844          TURNER    KING
      7900          JAMES     KING
      7782       CLARK        KING
      7934          MILLER    KING

Selezionate 14 righe.
```

Nel nostro caso c'è una sola radice, KING, ma se modifichiamo un po' i dati:

```
WTO> update emp set mgr=null where ename='BLAKE';

Aggiornata 1 riga.
```

Otteniamo quanto segue:

```
WTO> select empno,lpad(' ',level*3,' ')||ename nome,
  2         connect_by_root ename boss
  3    from emp
  4  connect by prior empno = mgr
  5    start with mgr is null;

     EMPNO NOME               BOSS
---------- ------------------ ----------
      7698    BLAKE           BLAKE
      7499       ALLEN        BLAKE
      7521       WARD         BLAKE
      7654       MARTIN       BLAKE
      7844       TURNER       BLAKE
      7900       JAMES        BLAKE
      7839    KING            KING
      7566       JONES        KING
      7788          SCOTT     KING
      7876             ADAMS  KING
      7902          FORD      KING
      7369             SMITH  KING
      7782       CLARK        KING
      7934          MILLER    KING
```

Dove per ogni record abbiamo visto chi è il capo supremo.

Funzione molto utile è la SYS_CONNECT_BY_PATH, prende in input un campo ed un carattere e crea l'intero percorso dalla radice al record corrente utilizzando il carattere dato come separatore:

```
WTO> select empno,lpad(' ',level*3,' ')||ename nome,
  2         sys_connect_by_path(ename,'/') capi
  3    from emp
  4  connect by prior empno = mgr
```

```
  5*   start with mgr is null

     EMPNO NOME                 CAPI
---------- ------------------ --------------------------------
      7839    KING            /KING
      7566       JONES        /KING/JONES
      7788          SCOTT     /KING/JONES/SCOTT
      7876             ADAMS  /KING/JONES/SCOTT/ADAMS
      7902          FORD      /KING/JONES/FORD
      7369             SMITH  /KING/JONES/FORD/SMITH
      7698       BLAKE        /KING/BLAKE
      7499          ALLEN     /KING/BLAKE/ALLEN
      7521          WARD      /KING/BLAKE/WARD
      7654          MARTIN    /KING/BLAKE/MARTIN
      7844          TURNER    /KING/BLAKE/TURNER
      7900          JAMES     /KING/BLAKE/JAMES
      7782       CLARK        /KING/CLARK
      7934          MILLER    /KING/CLARK/MILLER
```

C'è ancora una pseudo colonna molto interessante, CONNECT_BY_ISLEAF.

Ci dice se un record è foglia della gerarchia oppure no:

```
WTO> select empno,lpad(' ',level*3,' ')||ename nome,
  2        connect_by_isleaf foglia
  3    from emp
  4   connect by prior empno = mgr
  5    start with mgr is null;

     EMPNO NOME                   FOGLIA
---------- ------------------ ----------
      7839    KING                     0
      7566       JONES                 0
      7788          SCOTT              0
      7876             ADAMS           1
      7902          FORD               0
      7369             SMITH           1
      7698       BLAKE                 0
      7499          ALLEN              1
      7521          WARD               1
      7654          MARTIN             1
      7844          TURNER             1
      7900          JAMES              1
      7782       CLARK                 0
      7934          MILLER             1
```

Un'altra proprietà interessante della CONNECT BY è la capacità di generare più record da una tabella che ne ha uno solo:

```
WTO> select level from dual
  2  connect by level<=7;

     LEVEL
----------
         1
         2
         3
         4
         5
         6
         7
```

```
Selezionate 7 righe.
```

Possiamo estrarre da DUAL, che come tutti sanno ha una sola riga, tutti i record che ci servono per fare elaborazioni complesse.

Vediamo due esempi pratici.

Il primo esempio riguarda l'aggregazione di stringhe.

```
 SELECT deptno, ltrim(SYS_CONNECT_BY_PATH(ename, ','),',') enames
  FROM (select deptno, ename, rank()
        over(partition by deptno order by rownum) num from emp)
where connect_by_isleaf=1
START WITH num=1
CONNECT BY PRIOR num+1 = num and prior deptno=deptno;

    DEPTNO ENAMES
---------- ----------------------------------------
        10 CLARK,KING,MILLER
        20 SMITH,JONES,SCOTT,ADAMS,FORD
        30 ALLEN,WARD,MARTIN,BLAKE,TURNER,JAMES
```

Usando la SYS_CONNECT_BY_PATH siamo riusciti a mettere in una solo stringa, separati da virgola, i nomi dei dipendenti che lavorano nello stesso dipartimento.

E questo non c'entra assolutamente nulla con il campo MGR.

L'altro esempio estrae tutti gli anagrammi di una parola data:

```
with t as
(select 'pera' name from dual)
select distinct replace(sys_connect_by_path(c,' '),' ',null) str
  from (select substr(name,level,1) c, name,
               level*1000+ascii(substr(name,level,1)) cod
          from t
        connect by level<=length(name))
 where level=length(name)
connect by nocycle cod != prior cod

STR
-------------------------
prae
epar
arpe
rpae
paer
pare
repa
reap
prea
earp
aepr
aerp
pera
erpa
arep
erap
aper
pear
epra
eapr
```

```
rpea
rape
raep
apre

Selezionate 24 righe.
```

In questo caso la parola era "pera".

La query prima estrae tutte le lettere della parola data assegnando un codice univoco ad ognuna:

```
WTO> with t as
  2  (select 'pera' name from dual)
  3  select substr(name,level,1) c, name,
  4                 level*1000+ascii(substr(name,level,1)) cod
  5            from t
  6          connect by level<=length(name);

C NAME        COD
- ---- ----------
p pera       1112
e pera       2101
r pera       3114
a pera       4097
```

E successivamente utilizza la SYS_CONNECT_BY_PATH per costruire tutte le combinazioni possibili di queste lettere.

7.4.10 Paginazione dei record estratti in SQL

In Oracle 12c il comando di SELECT è stato implementato con una nuova clausola che consente di limitare i record estratti.

Questa clausola si basa su due parole chiave: OFFSET e FETCH. Queste parole chiave devono sempre essere utilizzate in combinazione con una ORDER BY o comunque bisogna essere certi che i record estratti siano ordinati in modo deterministico per evitare letture inconsistenti.

La clausola OFFSET consente di specificare il numero di righe da saltare prima della prima riga da estrarre.

Cominciamo popolando una tabella con i numeri da 1 a 20:

```
O12c>Create table test_fetch as
  2  Select level x from dual connect by level<=20;

Tabella creata.

O12c>Select * from test_fetch order by x;

         X
----------
         1
         2
         3
         4
         5
         6
         7
         8
```

```
         9
        10
        11
        12
        13
        14
        15
        16
        17
        18
        19
        20

20 righe selezionate.
```

E saltiamo i primi dieci record

```
O12c>Select * from test_fetch
  2  order by x OFFSET 10 rows;

         X
----------
        11
        12
        13
        14
        15
        16
        17
        18
        19
        20

10 righe selezionate.
```

Se il numero passato come OFFSET è nullo, maggiore o uguale del numero totale dei record, la query non restituirà righe.

```
O12c>Select * from test_fetch
  2  order by x offset 30 rows;

Nessuna riga selezionata
```

La parola chiave FETCH consente di indicare quante righe si vogliono estrarre. Si può indicare il numero esatto di righe da estrarre oppure una percentuale dei record totali:

```
O12c>Select * from test_fetch
  2  order by x FETCH first 10 rows only;

         X
----------
         1
         2
         3
         4
         5
         6
         7
         8
         9
        10
```

```
10 righe selezionate.

O12c>Select * from test_fetch
  2  order by x FETCH first 40 percent rows only;

         X
----------
         1
         2
         3
         4
         5
         6
         7
         8

8 righe selezionate.
```

Se non si specifica un numero di righe né una percentuale viene estratto un solo record:

```
O12c>Select * from test_fetch
  2  order by x FETCH first rows only;

         X
----------
         1
```

La parola chiave FIRST è intercambiabile con NEXT

```
O12c>Select * from test_fetch
  2  order by x FETCH first rows only;

         X
----------
         1

O12c>Select * from test_fetch
  2  order by x FETCH NEXT 10 rows only;

         X
----------
         1
         2
         3
         4
         5
         6
         7
         8
         9
        10

10 righe selezionate.

O12c>Select * from test_fetch
  2  order by x FETCH next 40 percent rows only;

         X
----------
         1
         2
         3
```

```
         4
         5
         6
         7
         8

8 righe selezionate.
```

Se ci sono uguaglianze sull’ordinamento si può decidere, usando le parole chiave ONLY e WITH TIES, se ottenere esattamente il numero di righe presenti nella clausola FETCH oppure aggiungere anche tutte le altre righe che hanno gli stessi valori:

Innanzitutto duplichiamo i record nella tabella di test:

```
O12c>Insert into test_fetch select * from test_fetch;

20 righe create.
```

E poi otteniamo esattamente 5 righe:

```
O12c>Select * from test_fetch
  2  order by x
  3  fetch first 5 rows only;

         X
----------
         1
         1
         2
         2
         3
```

Oppure 5 più gli eventuali ex-aequo

```
O12c>Select * from test_fetch
  2  order by x
  3  fetch first 5 rows with ties;

         X
----------
         1
         1
         2
         2
         3
         3

6 righe selezionate.
```

L’uso combinato di FETCH ed OFFSET consente la paginazione dei record: Pagina 1, da 5 righe

```
O12c>Select * from test_fetch
  2  order by x
  3  FETCH FIRST 5 ROWS ONLY;

         X
----------
         1
         1
```

```
         2
         2
         3
```

Pagina 2, da 5 righe

```
O12c>Select * from test_fetch
  2  order by x
  3  OFFSET 5 ROWS
  4  FETCH FIRST 5 ROWS ONLY;

         X
----------
         3
         4
         4
         5
         5
```

Pagina N, da 15 righe

```
O12c>Select * from test_fetch
  2  order by x
  3  OFFSET 5*(&N-1) ROWS
  4  FETCH FIRST 5 ROWS ONLY;
Immettere un valore per n: 3
vecchio   3: OFFSET 5*(&N-1) ROWS
nuovo   3: OFFSET 5*(3-1) ROWS

         X
----------
         6
         6
         7
         7
         8
```

7.4.11 Pattern matching di righe

Quest'argomento è considerato di livello avanzato. Per comprendere pienamente gli esempi è necessario conoscere argomenti di SQL e PL/SQL che in questa fase del manuale ancora non sono stati introdotti. Il lettore principiante può saltare questo paragrafo e tornarci alla fine del libro quando avrà acquisito gli strumenti necessari ad affrontarlo.

In Oracle 12c è stata aggiunta, nel comando SELECT, la clausola MATCH_RECOGNIZE.

Si tratta di un costrutto molto potente e complesso che consente di ricercare una regola (pattern) in una sequenza di record letti.

La clausola MATCH_RECOGNIZE è basata sui seguenti elementi:

Consente di partizionare ed ordinare i record letti dal DB utilizzando le parole chiave PARTITION BY e ORDER BY.

Consente di definire delle misure, cioè espressioni calcolate utilizzabili in altre parti della clausola, utilizzando la parola chiave MEASURES.

Consente di definire i pattern da ricercare nei record letti mediante la parola chiave PATTERN. In questa parola chiave si fa uso della stessa sintassi delle espressioni regolari.

Consente di specificare le condizioni logiche che devono essere verificate per confrontare un record con un pattern utilizzando la clausola DEFINE.

Vediamo un esempio.

Abbiamo una tabella ANDAMENTO contenente i punti realizzati, giornata per giornata, dalle squadre di un campionato di calcio.

La tabella è fatta così:

```
O12c>create table andamento
  2  (squadra varchar2(10),
  3   giornata number(2),
  4   punti number(1));

Tabella creata.

O12c>desc andamento
 Nome                                     Nullo?   Tipo
 ------------------------------------ -------- ---------------
 SQUADRA                                           VARCHAR2(10)
 GIORNATA                                          NUMBER(2)
 PUNTI                                             NUMBER(1)
```

E contiene questi dati:

```
insert into andamento values ('JUVE',1,3);
insert into andamento values ('JUVE',2,3);
insert into andamento values ('JUVE',3,3);
insert into andamento values ('JUVE',4,1);
insert into andamento values ('JUVE',5,3);
insert into andamento values ('JUVE',6,3);
insert into andamento values ('JUVE',7,1);
insert into andamento values ('JUVE',8,0);
insert into andamento values ('JUVE',9,1);
insert into andamento values ('JUVE',10,3);
insert into andamento values ('JUVE',11,3);
insert into andamento values ('JUVE',12,0);
insert into andamento values ('JUVE',13,1);
insert into andamento values ('JUVE',14,3);
insert into andamento values ('JUVE',15,3);
insert into andamento values ('JUVE',16,0);
insert into andamento values ('JUVE',17,0);
insert into andamento values ('JUVE',18,1);
insert into andamento values ('JUVE',19,3);
insert into andamento values ('NAPOLI',1,3);
insert into andamento values ('NAPOLI',2,3);
insert into andamento values ('NAPOLI',3,0);
insert into andamento values ('NAPOLI',4,1);
insert into andamento values ('NAPOLI',5,3);
insert into andamento values ('NAPOLI',6,3);
insert into andamento values ('NAPOLI',7,3);
insert into andamento values ('NAPOLI',8,0);
insert into andamento values ('NAPOLI',9,3);
insert into andamento values ('NAPOLI',10,3);
insert into andamento values ('NAPOLI',11,1);
insert into andamento values ('NAPOLI',12,1);
```

```
insert into andamento values ('NAPOLI',13,3);
insert into andamento values ('NAPOLI',14,3);
insert into andamento values ('NAPOLI',15,3);
insert into andamento values ('NAPOLI',16,3);
insert into andamento values ('NAPOLI',17,1);
insert into andamento values ('NAPOLI',18,1);
insert into andamento values ('NAPOLI',19,3);

O12c>select * from andamento
  2  order by squadra, giornata;

SQUADRA      GIORNATA      PUNTI
---------- ---------- ----------
JUVE                1          3
JUVE                2          3
JUVE                3          3
JUVE                4          1
JUVE                5          3
JUVE                6          3
JUVE                7          1
JUVE                8          0
JUVE                9          1
JUVE               10          3
JUVE               11          3
JUVE               12          0
JUVE               13          1
JUVE               14          3
JUVE               15          3
JUVE               16          0
JUVE               17          0
JUVE               18          1
JUVE               19          3
NAPOLI              1          3
NAPOLI              2          3
NAPOLI              3          0
NAPOLI              4          1
NAPOLI              5          3
NAPOLI              6          3
NAPOLI              7          3
NAPOLI              8          0
NAPOLI              9          3
NAPOLI             10          3
NAPOLI             11          1
NAPOLI             12          1
NAPOLI             13          3
NAPOLI             14          3
NAPOLI             15          3
NAPOLI             16          3
NAPOLI             17          1
NAPOLI             18          1
NAPOLI             19          3

38 righe selezionate.
```

Ci proponiamo di analizzare l’andamento delle squadre determinando per ognuna i periodi di crisi, cioè quelli in cui per una o più partite consecutive non sono arrivate vittorie.

Per ogni periodo di crisi vogliamo evidenziare la prima giornata, l’ultima e quella in cui la squadra ha ottenuto il risultato peggiore.

```
O12c>select *
```

```
  2  from andamento match_recognize (
  3  partition by squadra
  4  order by giornata
  5  measures r1.giornata+1 as inizio,
  6           last(peggio.giornata) as giornata_basso,
  7           last(meglio.giornata)-1 as fine
  8  pattern (r1 peggio+ meglio+)
  9  define
 10    peggio as peggio.punti <= prev(peggio.punti)
 11              and peggio.punti<3,
 12    meglio as meglio.punti > prev(meglio.punti)
 13              and prev(meglio.punti)<3
 14  ) mr
 15  order by mr.squadra, mr.inizio;

SQUADRA         INIZIO GIORNATA_BASSO       FINE
---------- ---------- -------------- ----------
JUVE                4              4          4
JUVE                7              8          9
JUVE               12             12         13
JUVE               16             17         18
NAPOLI              3              3          4
NAPOLI              8              8          8
NAPOLI             11             12         12
NAPOLI             17             18         18

8 righe selezionate.
```

La parola chiave PARTITION indica di partizionare i dati per squadra.

```
  3  partition by squadra
```

La parola chiave ORDER BY indica di orinare i dati (all'interno della partizione) per giornata.

```
  4  order by giornata
```

Poi definiamo le tre misure. L'inizio della crisi, la giornata in cui si è toccato il fondo, l'ultima giornata della crisi. Per capire bene come sono state definite bisogna vedere il pattern ricercato e le definizioni.

```
  5  measures r1.giornata+1 as inizio,
  6           last(peggio.giornata) as giornata_basso,
  7           last(meglio.giornata)-1 as fine
```

Il pattern ricercato è (r1 peggio+ meglio+). Vuol dire che, scorrendo i record, cominciamo da un certo record r1, poi vogliamo uno o più record di tipo "peggio" e poi uno o più record di tipo "meglio".

```
  8  pattern (r1 peggio+ meglio+)
```

Questa è la nostra definizione di crisi: si parte da una certa situazione, poi si va sempre peggio e si esce dalla crisi quando si va meglio.

Nella parola chiave DEFINE si definisce che vuol dire, per noi, "peggio" e "meglio": "peggio" vuol dire che i punti sono minori o uguali della giornata precedente e sono meno di tre (la squadra non ha vinto e non ha neanche fatto più punti della giornata precedente); "meglio" vuol dire che i punti sono maggiori o uguali della giornata precedente e la giornata precedente non era una vittoria.

```
  9  define
```

```
 10    peggio as peggio.punti <= prev(peggio.punti)
 11              and peggio.punti<3,
 12    meglio as meglio.punti > prev(meglio.punti)
 13              and prev(meglio.punti)<3
```

A questo punto le misure sono più chiare. Per come abbiamo definito il peggioramento, esso comincia con una vittoria. Per come abbiamo definito il miglioramento esso finisce con una vittoria.

Di conseguenza la prima e l'ultima giornata della crisi sono l'inizio del peggioramento+1

```
r1.giornata+1 as inizio
```

e la fine del miglioramento -1

```
last(meglio.giornata)-1
```

Il fondo della crisi è l'ultima giornata del peggioramento:

```
last(peggio.giornata)
```

Rivedendo i dati estratti:

```
SQUADRA        INIZIO GIORNATA_BASSO       FINE
---------- ---------- -------------- ----------
JUVE                4              4          4
JUVE                7              8          9
JUVE               12             12         13
JUVE               16             17         18
NAPOLI              3              3          4
NAPOLI              8              8          8
NAPOLI             11             12         12
NAPOLI             17             18         18
```

La Juve ha avuto una difficoltà alla quarta giornata, poi tre turni difficili tra la settima e la non giornata (con momento peggiore, una sconfitta, all'ottava) e così via…

Indovinate per chi faccio il tifo:

```
O12c>select squadra, sum(punti)
  2  from andamento
  3  group by squadra
  4  order by sum(punti) desc;

SQUADRA    SUM(PUNTI)
---------- ----------
NAPOLI             41
JUVE               35
```

Ma è solo il girone d'andata…

7.4.12 La clausola MODEL

Quest'argomento è considerato di livello avanzato. Per comprendere pienamente gli esempi è necessario conoscere argomenti di SQL e PL/SQL che in questa fase del manuale ancora non sono stati introdotti. Il lettore principiante può saltare questo paragrafo e tornarci alla fine del libro quando avrà acquisito gli strumenti necessari ad affrontarlo.

Oracle lavora continuamente per aumentare la capacità di calcolo complesso delle query. In quest'ottica, a partire dalla versione 10g del DB, è stata aggiunta la clausola MODEL al comando di query.

Questa clausola consente di trattare i dati estratti da una query come un foglio di calcolo. Consente inoltre di referenziare le singole celle per alterarne il valore oppure per utilizzarle come base di calcolo per altre celle.

Per gli esempi che seguono è stata utilizzata la tabella BILANCIO_FAMILIARE, di cui segue lo script di creazione e popolamento:

```
create table bilancio_familiare (
data date,
segno char(1) check (segno in ('D','A')),
categoria varchar2(20)
    check (categoria in
('CASA','PERSONALI','AUTO','ISTRUZIONE','REGALI','STIPENDI')),
descrizione varchar2(30),
importo number
);
insert into bilancio_familiare values
(DATE '2009-12-01', 'D', 'CASA', 'RATA MUTUO',500);
insert into bilancio_familiare values
(DATE '2009-12-10', 'D', 'PERSONALI', 'VACANZA',1000);
insert into bilancio_familiare values
(DATE '2009-12-10', 'D', 'AUTO', 'PIENO',60);
insert into bilancio_familiare values
(DATE '2009-12-20', 'D', 'AUTO', 'PIENO',60);
insert into bilancio_familiare values
(DATE '2009-12-30', 'D', 'AUTO', 'PIENO',60);
insert into bilancio_familiare values
(DATE '2010-01-10', 'D', 'AUTO', 'PIENO',60);
insert into bilancio_familiare values
(DATE '2010-01-20', 'D', 'AUTO', 'PIENO',60);
insert into bilancio_familiare values
(DATE '2010-01-30', 'D', 'AUTO', 'PIENO',60);
insert into bilancio_familiare values
(DATE '2010-01-01', 'D', 'CASA', 'RATA MUTUO',510);
insert into bilancio_familiare values
(DATE '2009-12-15', 'A', 'STIPENDI', '13esima',2000);
insert into bilancio_familiare values
(DATE '2009-12-27', 'A', 'STIPENDI', 'Dicembre',2200);
insert into bilancio_familiare values
(DATE '2010-01-27', 'A', 'STIPENDI', 'Gennaio',2400);
insert into bilancio_familiare values
(DATE '2009-12-15', 'D', 'REGALI', 'Natale',800);
insert into bilancio_familiare values
(DATE '2009-12-25', 'A', 'REGALI', 'Natale',200);
```

Ed ecco dunque i dati presenti in tabella.

```
WTO> select * from bilancio_familiare ;

DATA      S CATEGORIA            DESCRIZIONE          IMPORTO
--------- - -------------------- ------------------ ----------
01-DEC-09 D CASA                 RATA MUTUO                500
10-DEC-09 D PERSONALI            VACANZA                  1000
10-DEC-09 D AUTO                 PIENO                      60
20-DEC-09 D AUTO                 PIENO                      60
```

```
30-DEC-09 D AUTO                 PIENO                 60
10-JAN-10 D AUTO                 PIENO                 60
20-JAN-10 D AUTO                 PIENO                 60
30-JAN-10 D AUTO                 PIENO                 60
01-JAN-10 D CASA                 RATA MUTUO           510
15-DEC-09 A STIPENDI             13esima             2000
27-DEC-09 A STIPENDI             Dicembre            2200
27-JAN-10 A STIPENDI             Gennaio             2400
15-DEC-09 D REGALI               Natale               800
25-DEC-09 A REGALI               Natale               200

14 rows selected.
```

Noi lavoreremo su una sintesi di questi dati ottenuta accorpandoli per ANNO, segno e categoria di spesa:

```
WTO> select to_char(data,'yyyy') anno, segno,
  2         categoria, sum(importo) importo
  3    from bilancio_familiare
  4  group by to_char(data,'yyyy'), segno, categoria
  5  ;

ANNO S CATEGORIA               IMPORTO
---- - -------------------- ----------
2009 D AUTO                        180
2009 D CASA                        500
2009 D PERSONALI                  1000
2009 D REGALI                      800
2010 D CASA                        510
2010 A STIPENDI                   2400
2009 A STIPENDI                   4200
2010 D AUTO                        180
2009 A REGALI                      200

Selezionate 9 righe.
```

Innanzitutto in una query MODEL bisogna individuare tra le colonne disponibili:

- Partizioni
- Dimensioni
- Misure

Le partizioni sono colonne rispetto ai cui valori l'insieme dei dati da elaborare è completamente partizionato.

I dati appartenenti a partizioni differenti non si vedono tra di loro e quindi non possono essere inclusi gli uni delle formule di calcolo degli altri.

Nei nostri esempi la colonna partizione sarà l'anno. Quindi tutti i dati saranno calcolati su base annuale ed i dati di anni differenti non si mescoleranno mai tra loro.

Ovviamente non è obbligatorio avere delle partizioni.

Le dimensioni sono le chiavi di accesso ai dati. L'insieme delle dimensioni deve costituire una chiave primaria all'interno della partizione.

Nel nostro esempio saranno dimensioni il segno e la categoria.

Vuol dire che in uno specifico anno (partizione) le coppie (segno, categoria) non sono mai duplicate.

Le misure sono i valori su cui vengono effettivamente eseguiti i calcoli, sono in genere valori numerici. Nel nostro esempio la misura è l'importo.

Detto questo, da questo momento in poi si utilizzerà il termine cella per indicare uno specifico valore di IMPORTO individuato da ANNO, SEGNO e CATEGORIA.

Si può pensare ad una cartella Excel in cui per ogni anno c'è un foglio, ed in ogni foglio si utilizzano segno e categoria come coordinate (righe e colonne) con cui individuare le singole celle.

Il contenuto della singola cella è un valore di importo.

Calcolo vettoriale simbolico

Questo nome roboante rappresenta la capacità di popolare celle (o vettori di celle) con formule che si basano su altre celle.

Vediamo un semplice esempio.

Vogliamo aggiungere nuove celle al foglio per calcolare

1) il "TOTALE CASA E AUTO" facendo la somma delle celle che hanno segno D e categoria AUTO e CASA

2) il "TOTALE AVERE" facendo la somma delle celle che hanno segno A e categoria STIPENDI e REGALI

```
WTO> select anno, segno, categoria, importo
  2  from (select to_char(data,'yyyy') anno, segno,
  3                categoria, sum(importo) importo
  4           from bilancio_familiare
  5      group by to_char(data,'yyyy'), segno, categoria)
  6  model
  7    PARTITION BY (anno) DIMENSION BY (segno, categoria)
  8    MEASURES (importo)
  9    RULES
 10    (importo['D', 'TOTALE CASA e AUTO'] =
 11     importo['D', 'CASA'] + importo['D', 'AUTO'],
 12     importo['A', 'TOTALE AVERE'] =
 13     importo['A', 'STIPENDI'] + importo['A', 'REGALI']
 14     )
 15  ORDER BY anno, segno,categoria;

ANNO S CATEGORIA               IMPORTO
---- - -------------------- ----------
2009 A REGALI                      200
2009 A STIPENDI                   4200
2009 A TOTALE AVERE               4400
2009 D AUTO                        180
2009 D CASA                        500
2009 D PERSONALI                  1000
2009 D REGALI                      800
2009 D TOTALE CASA e AUTO          680
2010 A STIPENDI                   2400
2010 A TOTALE AVERE
```

```
2010 D AUTO                        180
2010 D CASA                        510
2010 D TOTALE CASA e AUTO          690

13 rows selected.
```

Come si vede, nella clausola MODEL sono state prima di tutto definite PARTIZIONI, DIMENSIONI e MISURE.

Poi si è passati alle regole, indicando che l'importo in Dare con categoria TOTALE CASA E AUTO è dato dalla somma degli importi Dare aventi categorie CASA e AUTO.

Con la stessa logica è stato calcolato il TOTALE AVERE.

Notiamo però che il TOTALE AVERE del 2010 ha importo nullo perché per il 2010 manca la cella (A, REGALI). Come al solito sommando quindi questo valore NULL al valore della cella (A, STIPENDI) si ottiene NULL.

Per ovviare al problema si può banalmente utilizzare la funzione NVL:

```
WTO> select anno, segno, categoria, importo
  2  from (select to_char(data,'yyyy') anno, segno,
  3               categoria, sum(importo) importo
  4          from bilancio_familiare
  5      group by to_char(data,'yyyy'), segno, categoria)
  6  model
  7    PARTITION BY (anno) DIMENSION BY (segno, categoria)
  8    MEASURES (importo)
  9    RULES
 10    (importo['D', 'TOTALE CASA e AUTO'] =
 11     importo['D', 'CASA'] + importo['D', 'AUTO'],
 12     importo['A', 'TOTALE AVERE'] =
 13     nvl(importo['A', 'STIPENDI'],0) +
 14     nvl(importo['A', 'REGALI'],0)
 15     )
 16  ORDER BY anno, segno,categoria;

ANNO S CATEGORIA               IMPORTO
---- - -------------------- ----------
2009 A REGALI                      200
2009 A STIPENDI                   4200
2009 A TOTALE AVERE               4400
2009 D AUTO                        180
2009 D CASA                        500
2009 D PERSONALI                  1000
2009 D REGALI                      800
2009 D TOTALE CASA e AUTO          680
2010 A STIPENDI                   2400
2010 A TOTALE AVERE               2400
2010 D AUTO                        180
2010 D CASA                        510
2010 D TOTALE CASA e AUTO          690

13 rows selected.
```

Adesso tutti gli importi sono valorizzati.

Aggiungiamo anche il “totale dare” utilizzando un operatore più generico:

```
WTO> select anno, segno, categoria, importo
```

```
  2  from (select to_char(data,'yyyy') anno, segno,
  3                categoria, sum(importo) importo
  4           from bilancio_familiare
  5      group by to_char(data,'yyyy'), segno, categoria)
  6  model
  7    PARTITION BY (anno) DIMENSION BY (segno, categoria)
  8    MEASURES (importo)
  9    RULES
 10    (importo['D', 'TOTALE CASA e AUTO'] =
 11     importo['D', 'CASA'] + importo['D', 'AUTO'],
 12     importo['A', 'TOTALE AVERE'] =
 13     nvl(importo['A', 'STIPENDI'],0)
 14     + nvl(importo['A', 'REGALI'],0),
 15     importo['D', 'TOTALE DARE'] =
 16     sum(importo)['D',categoria like '%']
 17     )
 18  ORDER BY anno, segno,categoria;

ANNO S CATEGORIA               IMPORTO
---- - -------------------- ----------
2009 A REGALI                      200
2009 A STIPENDI                   4200
2009 A TOTALE AVERE               4400
2009 D AUTO                        180
2009 D CASA                        500
2009 D PERSONALI                  1000
2009 D REGALI                      800
2009 D TOTALE CASA e AUTO          680
2009 D TOTALE DARE                3160
2010 A STIPENDI                   2400
2010 A TOTALE AVERE               2400
2010 D AUTO                        180
2010 D CASA                        510
2010 D TOTALE CASA e AUTO          690
2010 D TOTALE DARE                1380

15 rows selected.
```

Si è detto che la cella (D,TOTALE DARE) deve essere calcolata sommando gli importi di tutte le celle che hanno segno D e categoria like '%'. Cioè tutte le categorie.

Vedremo più avanti che questa cosa si può fare in maniera più elegante.

UPSERT o UPDATE?

Adesso però ci concentriamo sulla modalità di calcolo delle celle. ce ne sono tre distinte:

- UPDATE

Aggiorna le celle già presenti ma non aggiunge mai nuove celle

- UPSERT

Aggiorna le celle già presenti ed aggiunge nuove celle su un numero limitato di regole

- UPSERT ALL

Aggiorna le celle già presenti ed aggiunge nuove celle su un numero più ampio di regole

Si può specificare una modalità di calcolo per ogni regola ed il default è UPSERT.

Vediamo un esempio.

Tutte le celle che abbiamo calcolato finora sono celle aggiunte, essendo totali. Poiché non abbiamo mai specificato la modalità di calcolo essa vale UPSERT e le celle sono sempre state create.

Utilizziamo invece la modalità UPDATE per il calcolo del TOTALE DARE.

```
WTO> select anno, segno, categoria, importo
  2  from (select to_char(data,'yyyy') anno, segno,
  3               categoria, sum(importo) importo
  4          from bilancio_familiare
  5      group by to_char(data,'yyyy'), segno, categoria)
  6  model
  7    PARTITION BY (anno) DIMENSION BY (segno, categoria)
  8    MEASURES (importo)
  9    RULES
 10    (importo['D', 'TOTALE CASA e AUTO'] =
 11     importo['D', 'CASA'] + importo['D', 'AUTO'],
 12     importo['A', 'TOTALE AVERE'] =
 13     nvl(importo['A', 'STIPENDI'],0)
 14     + nvl(importo['A', 'REGALI'],0),
 15     UPDATE importo['D', 'TOTALE DARE'] =
 16     sum(importo)['D',categoria like '%']
 17     )
 18  ORDER BY anno, segno,categoria;

ANNO S CATEGORIA               IMPORTO
---- - -------------------- ----------
2009 A REGALI                      200
2009 A STIPENDI                   4200
2009 A TOTALE AVERE               4400
2009 D AUTO                        180
2009 D CASA                        500
2009 D PERSONALI                  1000
2009 D REGALI                      800
2009 D TOTALE CASA e AUTO          680
2010 A STIPENDI                   2400
2010 A TOTALE AVERE               2400
2010 D AUTO                        180
2010 D CASA                        510
2010 D TOTALE CASA e AUTO          690

13 rows selected.
```

Rispetto a prima abbiamo perso due righe. I due totali dare per il 2009 ed il 2010.

Siccome la cella (D, TOTALE DARE) non esiste al momento dell'estrazione, la regola in modalità UPDATE non l'aggiunge.

Se invece proviamo ad inserire in tabella un record fatto così:

```
insert into bilancio_familiare values
(DATE '2009-12-25', 'D', 'TOTALE DARE', null,1000);
```

La situazione cambia perché adesso la cella (D, TOTALE DARE) per il 2009 è presente e dunque la regola può essere applicata in modalità UPDATE:

```
WTO> select anno, segno, categoria, importo
  2  from (select to_char(data,'yyyy') anno, segno,
  3                categoria, sum(importo) importo
  4           from bilancio_familiare
  5       group by to_char(data,'yyyy'), segno, categoria)
  6  model
  7    PARTITION BY (anno) DIMENSION BY (segno, categoria)
  8    MEASURES (importo)
  9    RULES
 10    (importo['D', 'TOTALE CASA e AUTO'] =
 11     importo['D', 'CASA'] + importo['D', 'AUTO'],
 12     importo['A', 'TOTALE AVERE'] =
 13     nvl(importo['A', 'STIPENDI'],0)
 14     + nvl(importo['A', 'REGALI'],0),
 15     UPDATE importo['D', 'TOTALE DARE'] =
 16     sum(importo)['D',categoria like '%']
 17     )
 18  ORDER BY anno, segno,categoria;

ANNO S CATEGORIA               IMPORTO
---- - -------------------- ----------
2009 A REGALI                      200
2009 A STIPENDI                   4200
2009 A TOTALE AVERE               4400
2009 D AUTO                        180
2009 D CASA                        500
2009 D PERSONALI                  1000
2009 D REGALI                      800
2009 D TOTALE CASA e AUTO          680
2009 D TOTALE DARE                4160
2010 A STIPENDI                   2400
2010 A TOTALE AVERE               2400
2010 D AUTO                        180
2010 D CASA                        510
2010 D TOTALE CASA e AUTO          690

14 rows selected.
```

Ovviamente il TOTALE DARE per il 2010 continua a non esserci.

UPSERT ALL si comporta come UPSERT (cioè aggiunge le celle) ma su un numero maggiore di regole.

Ci sono regole, infatti, su cui UPSERT non aggiunge le celle.

Un esempio è il seguente:

```
WTO> select anno, segno, categoria, importo
  2  from (select to_char(data,'yyyy') anno, segno,
  3                categoria, sum(importo) importo
  4           from bilancio_familiare
  5       group by to_char(data,'yyyy'), segno, categoria)
  6  model
  7    PARTITION BY (anno) DIMENSION BY (segno, categoria)
  8    MEASURES (importo)
  9    RULES
 10    (importo[segno='A','TOTALE']=
 11    sum(importo)['A',categoria like '%'])
```

```
 12  ORDER BY anno, segno,categoria;

ANNO S CATEGORIA                 IMPORTO
---- - -------------------- ----------
2009 A REGALI                      200
2009 A STIPENDI                   4200
2009 D AUTO                        180
2009 D CASA                        500
2009 D PERSONALI                  1000
2009 D REGALI                      800
2010 A STIPENDI                   2400
2010 D AUTO                        180
2010 D CASA                        510

Selezionate 9 righe.

WTO> select anno, segno, categoria, importo
  2  from (select to_char(data,'yyyy') anno, segno,
  3                categoria, sum(importo) importo
  4           from bilancio_familiare
  5      group by to_char(data,'yyyy'), segno, categoria)
  6  model
  7    PARTITION BY (anno) DIMENSION BY (segno, categoria)
  8    MEASURES (importo)
  9    RULES
 10    ( UPSERT ALL importo[segno='A','TOTALE']=
 11     sum(importo)['A',categoria like '%'])
 12  ORDER BY anno, segno,categoria;

ANNO S CATEGORIA                 IMPORTO
---- - -------------------- ----------
2009 A REGALI                      200
2009 A STIPENDI                   4200
2009 A TOTALE                     4400
2009 D AUTO                        180
2009 D CASA                        500
2009 D PERSONALI                  1000
2009 D REGALI                      800
2010 A STIPENDI                   2400
2010 A TOTALE                     2400
2010 D AUTO                        180
2010 D CASA                        510

Selezionate 11 righe.
```

UPSERT non aggiunge celle quando le dimensioni sono specificate con una condizione (SEGNO='A') anziché con una costante ('A').

UPSERT ALL invece aggiunge le celle anche in questo caso.

La wildcard ANY

Quest'ultimo esempio ci ricorda che dovevamo trovare un modo più elegante per indicare "tutti i valori di categoria".

Lo possiamo fare con la parola chiave ANY:

```
WTO> select anno, segno, categoria, importo
  2  from (select to_char(data,'yyyy') anno, segno,
  3                categoria, sum(importo) importo
  4           from bilancio_familiare
  5      group by to_char(data,'yyyy'), segno, categoria)
```

```
  6  model
  7    PARTITION BY (anno) DIMENSION BY (segno, categoria)
  8    MEASURES (importo)
  9    RULES
 10    (importo['D', 'TOTALE CASA e AUTO'] =
 11     importo['D', 'CASA'] + importo['D', 'AUTO'],
 12     importo['A', 'TOTALE AVERE'] =
 13     nvl(importo['A', 'STIPENDI'],0)
.14     + nvl(importo['A', 'REGALI'],0),
 15     UPDATE importo['D', 'TOTALE DARE'] =
 16     sum(importo)['D',ANY]
 17     )
 18  ORDER BY anno, segno,categoria;

ANNO S CATEGORIA                  IMPORTO
---- - -------------------- ----------
2009 A REGALI                      200
2009 A STIPENDI                   4200
2009 A TOTALE AVERE               4400
2009 D AUTO                        180
2009 D CASA                        500
2009 D PERSONALI                  1000
2009 D REGALI                      800
2009 D TOTALE CASA e AUTO          680
2009 D TOTALE DARE                4160
2010 A STIPENDI                   2400
2010 A TOTALE AVERE               2400
2010 D AUTO                        180
2010 D CASA                        510
2010 D TOTALE CASA e AUTO          690

14 rows selected.
```

Abbiamo specificato il TOTALE DARE senza utilizzare una condizione complessa.

La funzione CV

L'utilizzo della parola chiave ANY apre però un problema.

Se la si utilizza a destra nella regola non ci sono dubbi. Se la si utilizza a sinistra, invece, come si fa a destra della regola a sapere quale valore della dimensione si sta lavorando?

Facciamo un esempio:

```
WTO> select anno, segno, categoria, importo
  2  from (select to_char(data,'yyyy') anno, segno,
  3               categoria, sum(importo) importo
  4          from bilancio_familiare
  5      group by to_char(data,'yyyy'), segno, categoria)
  6  model
  7    PARTITION BY (anno) DIMENSION BY (segno, categoria)
  8    MEASURES (importo)
  9    RULES
 10    (importo['A', 'TOTALE AVERE'] = sum(importo)['A',ANY],
 11     importo['D', 'TOTALE DARE'] = sum(importo)['D',ANY])
 12  ORDER BY anno, segno,categoria;

ANNO S CATEGORIA                  IMPORTO
---- - -------------------- ----------
2009 A REGALI                      200
```

```
2009 A STIPENDI                  4200
2009 A TOTALE AVERE              4400
2009 D AUTO                       180
2009 D CASA                       500
2009 D PERSONALI                 1000
2009 D REGALI                     800
2009 D TOTALE DARE               3480
2010 A STIPENDI                  2400
2010 A TOTALE AVERE              2400
2010 D AUTO                       180
2010 D CASA                       510
2010 D TOTALE DARE                690

13 rows selected.
```

Questa è facile perché calcola semplicemente la somma di tutte le categorie con segno A e con segno D.

Possiamo pensare di scrivere una sola regola che calcoli entrambi i totali usando ANY a sinistra invece che a destra?

Sì, possiamo:

```
WTO> select anno, segno, categoria, importo
  2  from (select to_char(data,'yyyy') anno, segno,
  3               categoria, sum(importo) importo
  4          from bilancio_familiare
  5      group by to_char(data,'yyyy'), segno, categoria)
  6  model
  7    PARTITION BY (anno) DIMENSION BY (segno, categoria)
  8    MEASURES (importo)
  9    RULES
 10    (UPSERT ALL importo[ANY,'TOTALE']=
 11     sum(importo)[CV(segno),ANY])
 12  ORDER BY anno, segno,categoria;
ANNO S CATEGORIA               IMPORTO
---- - -------------------- ----------
2009 A REGALI                     200
2009 A STIPENDI                  4200
2009 A TOTALE                    4400
2009 D AUTO                       180
2009 D CASA                       500
2009 D PERSONALI                 1000
2009 D REGALI                     800
2009 D TOTALE                    2480
2010 A STIPENDI                  2400
2010 A TOTALE                    2400
2010 D AUTO                       180
2010 D CASA                       510
2010 D TOTALE                     690

Selezionate 13 righe.
```

A sinistra abbiamo detto di voler calcolare tutte le celle (Segno, TOTALE) per tutti (ANY) i valori di segno.

A destra però abbiamo bisogno di indicare "La somma di tutti gli importi che hanno come segno QUELLO CHE STO LAVORANDO A SINISTRA".

La funzione CV fa proprio questo. Consente di mettere a destra dell'espressione di una regola un riferimento al valore corrente della dimensione che è attualmente in fase di lavorazione.

Le potenzialità di MODEL si spingono ancora oltre. In particolare è degna di nota segnalare la capacità definire dei cicli iterativi da applicare per il calcolo delle celle. Per tutti gli approfondimenti del caso si rimanda alla documentazione ufficiale.

7.5 Manipolazione dei dati

Oracle mette a disposizione alcuni operatori ed un ampio insieme di funzioni predefinite che consentono di manipolare i dati in fase di estrazione o di confronto con altre informazioni. L'utente del database può poi crearne di proprie utilizzando il PL/SQL.

7.5.1 Operatore di concatenazione

L'operatore di concatenazione consente di fondere in un'unica stringa due o più stringhe di partenza. L'operatore di concatenazione è rappresentato da una doppia barra verticale.

Nel primo esempio si concatenano i campi NOME e COGNOME della tabella clienti.

```
WTO >Select nome||cognome from clienti;

NOME||COGNOME
--------------------------------------------
MARCOROSSI
GIOVANNIBIANCHI
MATTEOVERDI
LUCANERI
AMBROGIOCOLOMBO
GENNAROESPOSITO
PASQUALERUSSO
VINCENZOAMATO

Selezionate 8 righe.
```

Per ottenere un risultato più leggibile è possibile concatenare uno spazio tra nome e cognome:

```
WTO >Select nome||' '||cognome from clienti;

NOME||''||COGNOME
-----------------------------------------------
MARCO ROSSI
GIOVANNI BIANCHI
MATTEO VERDI
LUCA NERI
AMBROGIO COLOMBO
GENNARO ESPOSITO
PASQUALE RUSSO
```

```
VINCENZO AMATO

Selezionate 8 righe.
```

Quando uno dei campi concatenati è NULL l'operatore di concatenazione lo ignora totalmente:

```
WTO >select nome||' '||cod_fisc
  2  from clienti;

NOME||''||COD_FISC
------------------------------------
MARCO RSSMRC70R20H501X
GIOVANNI
MATTEO VRDMTT69S02H501X
LUCA NRILCU77A22F205X
AMBROGIO
GENNARO SPSGNN71B10F839X
PASQUALE RSSPSQ70C14F839X
VINCENZO

Selezionate 8 righe.
```

L'operatore di concatenazione si applica solo a stringhe. Nel caso in cui venga applicato a tipi di dato differenti Oracle provvede autonomamente ad una conversione di tipo. Non vengono sollevati errori.

```
WTO >select data_fattura||'-'||importo
  2  from fatture;

DATA_FATTURA||'-'||IMPORTO
---------------------------------------------
01-OTT-10-300
01-DIC-10-500
20-OTT-10-700
01-FEB-11-1000
01-DIC-10-500
```

7.5.2 Operatori aritmetici

Sono disponibili anche i seguenti operatori aritmetici :

- Addizione (+)

```
WTO >select 3+5 from dual;

       3+5
----------
         8
```

- Sottrazione (–)

```
WTO >select importo, importo-100
  2  from fatture;

   IMPORTO IMPORTO-100
---------- -----------
       300         200
       500         400
       700         600
      1000         900
       500         400
```

- Moltiplicazione (*)

```
WTO >select importo, importo*0.20 iva
```

```
  2  from fatture;

   IMPORTO        IVA
---------- ----------
       300         60
       500        100
       700        140
      1000        200
       500        100
```

- Divisione (/)

```
WTO >select importo, importo/2
  2  from fatture;

   IMPORTO  IMPORTO/2
---------- ----------
       300        150
       500        250
       700        350
      1000        500
       500        250
```

Gli operatori aritmetici rispettano le consuete regole di precedenza, moltiplicazioni e divisioni hanno la precedenza rispetto ad addizioni e sottrazioni. Per forzare la precedenza è possibile fare uso di parentesi tonde.

```
WTO >select 3*(5-2)/4+1 from dual;

3*(5-2)/4+1
-----------
       3,25

WTO >select 3*5-2/4+1 from dual;

 3*5-2/4+1
----------
      15,5
```

7.5.3 Funzioni sulle stringhe

- CONCAT

La funzione CONCAT è equivalente all'operatore di concatenazione.

```
WTO >select concat(nome,cognome) from clienti;

CONCAT(NOME,COGNOME)
------------------------------------------------------------
MARCOROSSI
GIOVANNIBIANCHI
MATTEOVERDI
LUCANERI
AMBROGIOCOLOMBO
GENNAROESPOSITO
PASQUALERUSSO
VINCENZOAMATO

Selezionate 8 righe.
```

L'unica differenza è che questa funzione non accetta più di due parametri.

```
WTO >select concat(nome,' ',cognome) from clienti;
```

```
select concat(nome,' ',cognome) from clienti
       *
ERRORE alla riga 1:
ORA-00909: invalid number of arguments
```

Di conseguenza se fosse necessario concatenare più di due stringe bisognerebbe innestare più volte la funzione in sé stessa.

```
WTO >select concat(nome,concat(' ',cognome)) from clienti;

CONCAT(NOME,CONCAT('',COGNOME))
-------------------------------------------------------------
MARCO ROSSI
GIOVANNI BIANCHI
MATTEO VERDI
LUCA NERI
AMBROGIO COLOMBO
GENNARO ESPOSITO
PASQUALE RUSSO
VINCENZO AMATO

Selezionate 8 righe.
```

- SUBSTR

La funzione SUBSTR estrae una sottostringa da una stringa data. La funzione riceve in input tre parametri: la stringa di partenza; la posizione, all'interno della stringa di partenza, a partire dalla quale bisogna estrarre i caratteri; la lunghezza della stringa da estrarre.

Ipotizziamo ad esempio di voler estrarre i primi tre caratteri del nome dei clienti. La posizione di partenza è uno, la lunghezza della stringa è tre.

```
WTO >select nome, substr(nome,1,3) from clienti;

NOME     SUB
-------- ---
MARCO    MAR
GIOVANNI GIO
MATTEO   MAT
LUCA     LUC
AMBROGIO AMB
GENNARO  GEN
PASQUALE PAS
VINCENZO VIN
```

Se invece vogliamo estrarre il terzo, il quarto ed il quinto carattere dal nome la posizione di partenza sarà tre, la lunghezza sempre tre. In tal caso dal nome LUCA vengono estratti solo due caratteri visto che non esiste un quinto carattere.

```
WTO >select nome, substr(nome,3,3) from clienti;

NOME     SUB
-------- ---
MARCO    RCO
GIOVANNI OVA
MATTEO   TTE
LUCA     CA
AMBROGIO BRO
GENNARO  NNA
PASQUALE SQU
```

```
VINCENZO NCE
```

Se la posizione di partenza è negativa, essa indica una posizione a partire dall'ultimo carattere della stringa e contando verso sinistra. Se, ad esempio, si vogliono estrarre gli ultimi tre caratteri del nome bisognerà partire dalla posizione -3 ed estrarre tre caratteri.

```
WTO >select nome, substr(nome,-3,3) from clienti;

NOME     SUB
-------- ---
MARCO    RCO
GIOVANNI NNI
MATTEO   TEO
LUCA     UCA
AMBROGIO GIO
GENNARO  ARO
PASQUALE ALE
VINCENZO NZO

Selezionate 8 righe.
```

Se la lunghezza della stringa da estrarre è minore di uno, SUBSTR ritorna sempre NULL.

```
WTO >select nome, substr(nome,3,-3) from clienti;

NOME     S
-------- -
MARCO
GIOVANNI
MATTEO
LUCA
AMBROGIO
GENNARO
PASQUALE
VINCENZO

Selezionate 8 righe.
```

Se si omette del tutto la lunghezza della stringa si intende “fino alla fine della stringa”. Di conseguenza per estrarre tutti i caratteri presenti nel nome a partire dal terzo scriveremo:

```
WTO >select nome, substr(nome,3) from clienti;

NOME     SUBSTR(NOME,3)
-------- ----------------------------
MARCO    RCO
GIOVANNI OVANNI
MATTEO   TTEO
LUCA     CA
AMBROGIO BROGIO
GENNARO  NNARO
PASQUALE SQUALE
VINCENZO NCENZO
Selezionate 8 righe.
```

E per estrarre gli ultimi tre caratteri scriveremo:

```
WTO >select nome, substr(nome,-3) from clienti;

NOME     SUB
```

```
-------- ---
MARCO    RCO
GIOVANNI NNI
MATTEO   TEO
LUCA     UCA
AMBROGIO GIO
GENNARO  ARO
PASQUALE ALE
VINCENZO NZO

Selezionate 8 righe.
```

Quest'ultima è equivalente alla SUBSTR(NOME,-3,3).

- INSTR

La funzione INSTR riceve in input due stringhe. Ritorna un numero che rappresenta a che posizione, all'interno della prima stringa, si trova la seconda stringa. La funzione torna zero nel caso in cui la seconda stringa non è contenuta nella prima.

Ipotizziamo ad esempio di voler sapere a che posizione, nei nomi dei clienti, è presente la lettera 'A'

```
WTO >select nome, instr(nome,'A') from clienti;

NOME     INSTR(NOME,'A')
-------- ---------------
MARCO                  2
GIOVANNI               5
MATTEO                 2
LUCA                   4
AMBROGIO               1
GENNARO                5
PASQUALE               2
VINCENZO               0
```

La lettera 'A' è presente in tutti i nomi ad eccezione di 'VINCENZO', solo in quel caso, infatti, INSTR restituisce zero.

Se invece cerchiamo la stringa 'AR' il risultato cambia:

```
WTO >select nome, instr(nome,'AR') from clienti;

NOME     INSTR(NOME,'AR')
-------- ----------------
MARCO                   2
GIOVANNI                0
MATTEO                  0
LUCA                    0
AMBROGIO                0
GENNARO                 5
PASQUALE                0
VINCENZO                0
```

Solo MARCO e GENNARO contengono la stringa 'AR'.

INSTR ammette un terzo parametro, opzionale, mediante il quale è possibile indicare a partire da quale posizione nella prima stringa cominciare la ricerca della seconda. Tornando al primo esempio cerchiamo la lettera 'A' nei nomi ma partendo dal terzo carattere.

```
WTO >select nome, instr(nome,'A',3) from clienti;

NOME     INSTR(NOME,'A',3)
-------- -----------------
MARCO                    0
GIOVANNI                 5
MATTEO                   0
LUCA                     4
AMBROGIO                 0
GENNARO                  5
PASQUALE                 6
VINCENZO                 0

Selezionate 8 righe.
```

MARCO, MATTEO ed AMBROGIO non contengono nessuna 'A' a partire dal terzo carattere.

Quando la lettera 'A' è presente più di una volta INSTR torna la prima posizione in cui essa è stata trovata. Per cambiare questo comportamento è possibile fornire come quarto parametro l'occorrenza della seconda stringa che si desidera trovare nella prima. Tornando al primo esempio avremo:

```
WTO >select nome, instr(nome,'A',1,2) from clienti;

NOME     INSTR(NOME,'A',1,2)
-------- -------------------
MARCO                      0
GIOVANNI                   0
MATTEO                     0
LUCA                       0
AMBROGIO                   0
GENNARO                    0
PASQUALE                   6
VINCENZO                   0

Selezionate 8 righe.
```

L'unico nome che contiene almeno due volte la lettera 'A' è PASQUALE, la seconda occorrenza della 'A' in PASQUALE si trova al sesto carattere.

- LENGTH

La funzione LENGTH riceve in input una stringa e ne ritorna la lunghezza. La LENGTH di una stringa NULL è NULL.

```
WTO >select nome, length(nome) from clienti;

NOME     LENGTH(NOME)
-------- ------------
MARCO               5
GIOVANNI            8
MATTEO              6
LUCA                4
AMBROGIO            8
GENNARO             7
PASQUALE            8
VINCENZO            8

Selezionate 8 righe.
```

```
WTO >select cod_fisc, length(cod_fisc) from clienti;

COD_FISC         LENGTH(COD_FISC)
---------------- ----------------
RSSMRC70R20H501X               16

VRDMTT69S02H501X               16
NRILCU77A22F205X               16

SPSGNN71B10F839X               16
RSSPSQ70C14F839X               16


Selezionate 8 righe.
```

- LOWER, UPPER ed INITCAP

La funzione LOWER riceve in input una stringa e la restituisce tutta al minuscolo.

```
WTO >select lower(nome) from clienti;

LOWER(NOME)
------------------------------
marco
giovanni
matteo
luca
ambrogio
gennaro
pasquale
vincenzo

Selezionate 8 righe.
```

La funzione UPPER riceve in input una stringa e la restituisce tutta al maiuscolo.

```
WTO >select upper('una stringa minuscola') from dual;

UPPER('UNASTRINGAMINU
---------------------
UNA STRINGA MINUSCOLA
```

La funzione INITCAP riceve in input una stringa e la restituisce tutta al minuscolo con le iniziali delle parole maiuscole.

```
WTO >select initcap(nome) from clienti;

INITCAP(NOME)
------------------------------
Marco
Giovanni
Matteo
Luca
Ambrogio
Gennaro
Pasquale
Vincenzo

WTO >select initcap('una stringa minuscola') from dual;
```

```
INITCAP('UNASTRINGAMI
---------------------
Una Stringa Minuscola
```

- LTRIM, RTRIM e TRIM

La funzione LTRIM riceve in input due stringhe. Elimina dalla sinistra della prima tutte le occorrenze della seconda stringa. Ipotizziamo di voler eliminare tutte le 'X' dalla sinistra della stringa 'XXXSTRINGAXXX'.

```
WTO >select ltrim('XXXSTRINGAXXX','X') from dual;

LTRIM('XXX
----------
STRINGAXXX
```

La stringa da eliminare può essere composta anche da più di un carattere.

```
WTO >select ltrim('XYXYXSTRINGAXXX','XY') from dual;

LTRIM('XYX
----------
STRINGAXXX
```

Il caso più frequente è l'eliminazione degli spazi.

```
WTO >select ltrim('     STRINGA    ',' ') from dual;

LTRIM('STRI
-----------
STRINGA
```

Quando il carattere da eliminare è lo spazio il secondo parametro può essere omesso.

```
WTO >select ltrim('     STRINGA    ') from dual;

LTRIM('STRI
-----------
STRINGA
```

La funzione RTRIM si comporta esattamente come la LTRIM ma alle destra della stringa.

```
WTO >select rtrim('XXXSTRINGAXXX','X') from dual;

RTRIM('XXX
----------
XXXSTRINGA

WTO >select rtrim('XYXYXSTRINGAXYXYX','XY') from dual;

RTRIM('XYXYX
------------
XYXYXSTRINGA

WTO >select rtrim('     STRINGA    ',' ') from dual;

RTRIM('STRIN
------------
     STRINGA

WTO >select rtrim('     STRINGA    ') from dual;
```

```
RTRIM('STRIN
------------
     STRINGA
```

La funzione TRIM elimina gli spazi sia dalla destra che dalla sinistra di una stringa:

```
WTO >select trim('     STRINGA    ') from dual;

TRIM('S
-------
STRINGA
```

La funzione TRIM può essere utilizzata solo per eliminare gli spazi, non accetta altre stringhe. Ipotizziamo di voler eliminare tutte le 'X' sia dalla sinistra che dalla destra della stringa 'XXXSTRINGAXXX'.

```
WTO >select trim('XXXSTRINGAXXX','X') from dual;
select trim('XXXSTRINGAXXX','X') from dual
                            *
ERRORE alla riga 1:
ORA-00907: missing right parenthesis
```

Poiché non è possibile farlo con la TRIM sarà necessario innestare in un'unica chiamata le funzioni RTRIM ed LTRIM.

```
WTO >select ltrim(rtrim('XXXSTRINGAXXX','X'),'X') from dual;

LTRIM(R
-------
STRINGA
```

- LPAD ed RPAD

La funzione LPAD riceve in input due stringhe ed una lunghezza. Effettua il riempimento della prima stringa con la seconda stringa fino a raggiungere la lunghezza data. Ipotizziamo ad esempio di voler riempire a sinistra i nomi dei clienti con caratteri asterisco fino ad una lunghezza di dieci caratteri.

```
WTO >select lpad(nome,10,'*') from clienti;

LPAD(NOME,
----------
*****MARCO
**GIOVANNI
****MATTEO
******LUCA
**AMBROGIO
***GENNARO
**PASQUALE
**VINCENZO

Selezionate 8 righe.
```

Se la prima stringa è più corta della lunghezza desiderata essa viene tagliata.

```
WTO >select lpad(nome,7,'*') from clienti;

LPAD(NO
-------
**MARCO
GIOVANN
*MATTEO
***LUCA
AMBROGI
GENNARO
PASQUAL
VINCENZ

Selezionate 8 righe.
```

La funzione RPAD si comporta esattamente come la LPAD ma il riempimento avviene sulla destra della stringa.

```
WTO >select rpad(nome,10,'*') from clienti;

RPAD(NOME,
----------
MARCO*****
GIOVANNI**
MATTEO****
LUCA******
AMBROGIO**
GENNARO***
PASQUALE**
VINCENZO**

WTO >select rpad(nome,7,'*') from clienti;

RPAD(NO
-------
MARCO**
GIOVANN
MATTEO*
LUCA***
AMBROGI
GENNARO
PASQUAL
VINCENZ

Selezionate 8 righe.
```

- REPLACE

La funzione REPLACE riceve in input tre stringhe ed effettua una sostituzione, nella prima stringa tutte le occorrenze della seconda stringa vengono sostituite con occorrenze della terza. Ad esempio se nel nome dei clienti si intende sostituire le lettere ‘A’ con il carattere asterisco si scriverà.

```
WTO >select replace(nome,'A','*') from clienti;

REPLACE(NOME,'A','*')
------------------------------
M*RCO
GIOV*NNI
M*TTEO
LUC*
*MBROGIO
```

```
GENN*RO
P*SQU*LE
VINCENZO

Selezionate 8 righe.
```

La stringa da sostituire e la stringa sostitutiva non devono necessariamente avere la stessa lunghezza.

```
WTO >select replace(nome,'A','<+>') from clienti;

REPLACE(NOME,'A','<+>')
-----------------------------------------------------------
M<+>RCO
GIOV<+>NNI
M<+>TTEO
LUC<+>
<+>MBROGIO
GENN<+>RO
P<+>SQU<+>LE
VINCENZO

Selezionate 8 righe.
```

- TRANSLATE

La funzione TRANSLATE riceve in input tre stringhe ed effettua una sostituzione. La prima stringa è quella in cui viene effettuata la sostituzione. La seconda e la terza stringa rappresentano due alfabeti. Ogni occorrenza del carattere che si trova al primo posto nel primo alfabeto sarà sostituito con un'occorrenza del carattere che si trova al primo posto nel secondo alfabeto e così via. Ad esempio, nei nomi dei clienti, vogliamo sostituire ogni 'A' con un '*', ogni 'M' con un '-' ed ogni 'R' con un '+'. Scriveremo.

```
WTO >select nome, translate(nome,'AMR','*-+') from clienti;

NOME     TRANSLATE(NOME,'AMR','*-+')
-------- ------------------------------
MARCO    -*+CO
GIOVANNI GIOV*NNI
MATTEO   -*TTEO
LUCA     LUC*
AMBROGIO *-B+OGIO
GENNARO  GENN*+O
PASQUALE P*SQU*LE
VINCENZO VINCENZO

Selezionate 8 righe.
```

Un esempio di utilizzo della funzione può essere la pulizia di una stringa dai caratteri speciali. Prendiamo ad esempio la stringa

```
'stringa *+-:con.:/\ caratteri speciali'
```

e ci proponiamo di eliminare in essa i caratteri speciali. La seguente TRANSLATE non assegna alcun carattere di corrispondenza ai caratteri speciali, dunque li elimina dalla stringa.

```
WTO >select translate('stringa *+-:con.:/\ caratteri speciali',
  2  ' *+-:.:/\',' ')
```

```
  3  from dual;

TRANSLATE('STRINGA*+-:CON.:/\C
------------------------------
stringa con caratteri speciali
```

- ASCII e CHR

La funzione ASCII riceve in input un carattere e ne restituisce il codice ASCII (10.4.3) nel set di caratteri del database.

```
WTO >select ascii('A') from dual;

ASCII('A')
----------
        65

WTO >select ascii('a') from dual;

ASCII('A')
----------
        97

WTO >select ascii('{') from dual;

ASCII('{')
----------
       123
```

La funzione CHR, inversa della ASCII, riceve in input un numero e restituisce il carattere che nel set di caratteri del database occupa quella posizione.

```
WTO >select chr(65), chr(97), chr(123) from dual;

C C C
- - -
A a {
```

7.5.4 Funzioni sui numeri

- ABS

La funzione ABS riceve in input un numero e ne restituisce il valore assoluto.

```
WTO >select abs(-3), abs(5) from dual;

   ABS(-3)     ABS(5)
---------- ----------
         3          5
```

- ROUND e TRUNC

La funzione ROUND riceve in input un numero e lo restituisce arrotondato alla parte intera. L'arrotondamento aritmetico prevede che le cifre non significative (i decimali in questo caso) vengano abbandonate, l'ultima cifra significativa viene aumentata di una unità se la prima cifra non significativa va da cinque a nove.

```
WTO >select round(10.3), round(3.5), round(2.4)
  2  from dual;

ROUND(10.3) ROUND(3.5) ROUND(2.4)
----------- ---------- ----------
         10          4          2
```

La funzione accetta opzionalmente in input un secondo parametro che indica a quante cifre decimali bisogna effettuare l'arrotondamento.

```
WTO >select round(2.345, 3), round(2.345,2), round(2.345,1)
  2  from dual;

ROUND(2.345,3) ROUND(2.345,2) ROUND(2.345,1)
-------------- -------------- --------------
         2,345           2,35            2,3

WTO >select round(2.678, 3), round(2.678,2), round(2.678,1)
  2  from dual;

ROUND(2.678,3) ROUND(2.678,2) ROUND(2.678,1)
-------------- -------------- --------------
         2,678           2,68            2,7
```

Nel caso in cui il secondo parametro sia un numero negativo l'arrotondamento sarà effettuato sulla parte intera del numero: decine, centinaia, migliaia…

```
WTO >select round(2345,-1), round(2345,-2), round(2345,-3)
  2  from dual;

ROUND(2345,-1) ROUND(2345,-2) ROUND(2345,-3)
-------------- -------------- --------------
          2350           2300           2000

WTO >select round(2678,-1), ROUND(2678,-2), ROUND(2678,-3)
  2  from dual;

ROUND(2678,-1) ROUND(2678,-2) ROUND(2678,-3)
-------------- -------------- --------------
          2680           2700           3000
```

La funzione TRUNC si comporta come la ROUND ad eccezione del fatto che viene effettuato un troncamento e non un arrotondamento. Le cifre considerate non significative quindi vengono sempre abbandonate senza apportare alcuna modifica alle cifre significative.

```
WTO >select trunc(10.3), trunc(3.5), trunc(2.4)
  2  from dual;

TRUNC(10.3) TRUNC(3.5) TRUNC(2.4)
----------- ---------- ----------
         10          3          2

WTO >select trunc(2.345, 3), trunc(2.345,2), trunc(2.345,1)
  2  from dual;

TRUNC(2.345,3) TRUNC(2.345,2) TRUNC(2.345,1)
-------------- -------------- --------------
         2,345           2,34            2,3

WTO >select trunc(2.678, 3), trunc(2.678,2), trunc(2.678,1)
```

```
  2  from dual;

TRUNC(2.678,3) TRUNC(2.678,2) TRUNC(2.678,1)
-------------- -------------- --------------
         2,678           2,67            2,6

WTO >select trunc(2345,-1), trunc(2345,-2), trunc(2345,-3)
  2  from dual;

TRUNC(2345,-1) TRUNC(2345,-2) TRUNC(2345,-3)
-------------- -------------- --------------
          2340           2300           2000

WTO >select trunc(2678,-1), trunc(2678,-2), trunc(2678,-3)
  2  from dual;

TRUNC(2678,-1) TRUNC(2678,-2) TRUNC(2678,-3)
-------------- -------------- --------------
          2670           2600           2000
```

- CEIL e FLOOR

La funzione CEIL restituisce il più piccolo intero maggiore o uguale del numero ricevuto in input.

```
WTO >select ceil(2.6), ceil(3), ceil(3.3)
  2  from dual;

 CEIL(2.6)    CEIL(3)  CEIL(3.3)
---------- ---------- ----------
         3          3          4

WTO >select ceil(-2.6), ceil(-3), ceil(-3.3)
  2  from dual;

CEIL(-2.6)   CEIL(-3) CEIL(-3.3)
---------- ---------- ----------
        -2         -3         -3
```

La funzione FLOOR restituisce il più grande intero minore o uguale del numero ricevuto in input.

```
WTO >select floor(2.6), floor(3), floor(3.3)
  2  from dual;

FLOOR(2.6)   FLOOR(3) FLOOR(3.3)
---------- ---------- ----------
         2          3          3

WTO >select floor(-2.6), floor(-3), floor(-3.3)
  2  from dual;

FLOOR(-2.6)  FLOOR(-3) FLOOR(-3.3)
----------- ---------- -----------
         -3         -3          -4
```

- EXP, LN e LOG

La funzione esponenziale EXP restituisce il risultato di una potenza avente come base il numero di Nepero (detto anche numero di Eulero 2,718281828459…) e come esponente il numero passato in input alla funzione.

```
WTO >select exp(1), exp(2)
  2  from dual;

      EXP(1)       EXP(2)
------------ ------------
2,7182818285 7,3890560989
```

La funzione Logaritmo naturale LN restituisce il logaritmo del numero ricevuto in input utilizzando come base del logaritmo il numero di Nepero.

```
WTO >select ln(2.7182818285), ln(10)
  2  from dual;

LN(2.7182818285)       LN(10)
---------------- ------------
               1  2,302585093
```

La funzione logaritmo LOG riceve in input due numeri. La funzione restituisce il logaritmo del secondo numero ricevuto in input utilizzando come base il primo numero.

```
WTO >select log(10,10), log(2,8)
  2  from dual;

  LOG(10,10)     LOG(2,8)
------------ ------------
           1            3
```

- POWER e SQRT

La funzione potenza POWER riceve in input due numeri. Restituisce il risultato di una potenza avente come base il primo parametro e come esponente il secondo.

```
WTO >select power(2,3), power(10,4)
  2  from dual;

  POWER(2,3)  POWER(10,4)
------------ ------------
           8        10000
```

La funzione SQRT restituisce la radice quadrata del numero passato in input.

```
WTO >select sqrt(16), sqrt(2)
  2  from dual;

    SQRT(16)      SQRT(2)
------------ ------------
           4 1,4142135624
```

- SIGN

La funzione segno SIGN riceve in input un numero e restituisce:

- 1 se il numero in input è maggiore di zero,
- 0 se il numero in input è zero,
- -1 se il numero in input è minore di zero.

```
WTO >select sign(-23), sign(0), sign(3)
  2  from dual;

   SIGN(-23)      SIGN(0)      SIGN(3)
------------ ------------ ------------
```

```
         -1            0            1
```

- MOD e REMAINDER

La funzione modulo MOD riceve in input due numeri e restituisce il resto che si ottiene dividendo il primo parametro per il secondo.

```
WTO >select mod(10,3), mod(28,7), mod(17,9)
  2  from dual;

   MOD(10,3)    MOD(28,7)    MOD(17,9)
------------ ------------ ------------
           1            0            8
```

La funzione resto REMAINDER è simile alla MOD con la differenza che, per determinare il quoziente della divisione, fa uso al suo interno della funzione ROUND invece che della funzione FLOOR. L'esempio seguente mostra la differenza di risultato di MOD e REMAINDER in un caso particolare.

```
WTO> select mod(10,7), remainder(10,7)
  2  from dual;

 MOD(10,7) REMAINDER(10,7)
---------- ---------------
         3               3

WTO> select mod(10,6), remainder(10,6)
  2  from dual;

 MOD(10,6) REMAINDER(10,6)
---------- ---------------
         4              -2
```

- TRIGONOMETRICHE

Oracle mette a disposizione alcune funzioni trigonometriche che ci limiteremo ad elencare: seno (SIN) e seno iperbolico (SINH); coseno (COS) e coseno iperbolico (COSH); tangente (TAN) e tangente iperbolica (TANH); arcoseno (ASIN), arco coseno (ACOS), arcotangente (ATAN) ed arcotangente a due parametri (ATAN2). Le due funzioni arcotangente sono legate dalla seguente equivalenza:

ATAN2(a,b) = ATAN(a/b).

7.5.5 Funzioni sulle date

Oracle per default presenta le date nel formato DD-MON-YY che significa due cifre per il giorno, tre lettere per il mese e due cifre per l'anno. Come già detto, però, una data in Oracle contiene sempre anche ore, minuti e secondi. Il formato di default dunque nasconde alcune informazioni significative. I formati delle date saranno approfonditi più avanti, per il momento, per comprendere meglio gli esempi presenti in questo paragrafo, conviene eseguire la seguente istruzione SQL:

```
WTO >alter session set nls_date_format='dd-mm-yyyy hh24:mi:ss';
Modificata sessione.
```

In questo modo per tutte le date saranno visualizzati: due cifre per il giorno, due cifre per il mese, quattro cifre per l'anno, due cifre per l'ora nel formato a 24 ore, due cifre per i minuti e due cifre per i secondi.

- SYSDATE e SYSTIMESTAMP

La funzione SYSDATE, senza parametri, restituisce la data corrente.

```
WTO >select sysdate from dual;
SYSDATE
-------------------
13-02-2011 17:39:11
```

La funzione SYSTIMESTAMP, invece, fornisce la data corrente munita di frazioni di secondo ed indicazioni sul fuso orario.

```
WTO >select systimestamp from dual;
SYSTIMESTAMP
-----------------------------------
13-FEB-11 17:40:11,641727 +01:00
```

- Aritmetica sulle date

È possibile modificare una data semplicemente addizionandogli o sottraendogli un numero dei giorni.

Per ottenere la data di domani a quest'ora dunque si può scrivere.

```
WTO >select sysdate+1 from dual;
SYSDATE+1
-------------------
14-02-2011 17:41:59
```

Ovviamente il numero aggiunto o sottratto non deve essere necessariamente intero. Per togliere un minuto alla data corrente si può scrivere.

```
WTO >select sysdate, sysdate-1/(24*60)
  2  from dual;
SYSDATE             SYSDATE-1/(24*60)
------------------- -------------------
13-02-2011 17:43:23 13-02-2011 17:42:23
```

Poiché, se l'unità vale un giorno allora 1/24 vale un ora e 1/(24*60) vale un minuto.

- ADD_MONTHS

La funzione ADD_MONTHS riceve in input una data ed un numero, positivo o negativo, di mesi. Restituisce una data ottenuta addizionando al primo parametro i mesi indicati nel secondo parametro.

```
WTO >select add_months(date'2011-03-13',2)
  2  from dual;
ADD_MONTHS(DATE'201
-------------------
13-05-2011 00:00:00

WTO >select add_months(date'2011-03-13',-5)
  2  from dual;

ADD_MONTHS(DATE'201
-------------------
```

```
13-10-2010 00:00:00
```

La funzione si comporta in modo particolare a fine mese: se la data di partenza è fine mese, infatti, la data calcolata sarà sempre fine mese:

```
WTO >select add_months(date'2011-02-28',1)
  2  from dual;

ADD_MONTHS(DATE'201
-------------------
31-03-2011 00:00:00

WTO >select add_months(date'2011-02-28',2)
  2  from dual;

ADD_MONTHS(DATE'201
-------------------
30-04-2011 00:00:00

WTO >select add_months(date'2011-02-28',-1)
  2  from dual;

ADD_MONTHS(DATE'201
-------------------
31-01-2011 00:00:00

WTO >select add_months(date'2011-02-28',-2)
  2  from dual;

ADD_MONTHS(DATE'201
-------------------
31-12-2010 00:00:00
```

- MONTHS_BETWEEN

La funzione MONTHS_BETWEEN prende in input due date e ritorna la differenza in mesi tra le due. Il risultato è positivo se la prima data è maggiore della seconda, altrimenti è negativo.

```
WTO >select months_between(date'2011-02-28', date'2011-05-07')
  2  from dual;

MONTHS_BETWEEN(DATE'2011-02-28',DATE'2011-05-07')
-------------------------------------------------
                                     -2,322580645

WTO >select months_between(date'2011-05-07', date'2011-02-28')
  2  from dual;

MONTHS_BETWEEN(DATE'2011-05-07',DATE'2011-02-28')
-------------------------------------------------
                                     2,3225806452

```

Anche questa funzione assume un comportamento particolare, e non lineare, a fine mese:

```
WTO >select months_between(date'2011-02-28',date'2011-03-28')
  2  from dual;

MONTHS_BETWEEN(DATE'2011-02-28',DATE'2011-03-28')
-------------------------------------------------
```

```
                                              -1

WTO >select months_between(date'2011-02-28',date'2011-03-30')
  2  from dual;

MONTHS_BETWEEN(DATE'2011-02-28',DATE'2011-03-30')
-------------------------------------------------
                                     -1,064516129

WTO >select months_between(date'2011-02-28',date'2011-03-31')
  2  from dual;

MONTHS_BETWEEN(DATE'2011-02-28',DATE'2011-03-31')
-------------------------------------------------
                                               -1

WTO >select months_between(date'2011-02-28',date'2011-04-01')
  2  from dual;

MONTHS_BETWEEN(DATE'2011-02-28',DATE'2011-04-01')
-------------------------------------------------
                                     -1,129032258
```

- LAST_DAY

La funzione LAST_DAY riceve in input una data e restituisce l'ultimo giorno del mese in cui cade la data in input. La parte ore-minuti-secondi resta inalterata.

```
WTO >select date'2011-03-05', last_day(date'2011-03-05')
  2  from dual;

DATE'2011-03-05'    LAST_DAY(DATE'2011-
------------------- -------------------
05-03-2011 00:00:00 31-03-2011 00:00:00

WTO >select sysdate, last_day(sysdate)
  2  from dual;

SYSDATE             LAST_DAY(SYSDATE)
------------------- -------------------
13-02-2011 17:58:00 28-02-2011 17:58:00
```

- NEXT_DAY

La funzione NEXT_DAY riceve in input una data ed il nome di un giorno della settimana. Restituisce il primo giorno successivo al primo parametro che cada nel giorno della settimana indicato come secondo parametro. Oggi è domenica, chiedendo

```
NEXT_DAY(SYSDATE, 'MARTEDI')
```

si richiede ad Oracle la data del primo martedì successivo ad oggi.

```
WTO >select sysdate, next_day(sysdate, 'martedi')
  2  from dual;

SYSDATE             NEXT_DAY(SYSDATE,'M
------------------- -------------------
13-02-2011 18:00:38 15-02-2011 18:00:38
```

Se la data in input cade già nel giorno indicato nel secondo parametro la funzione NEXT_DAY equivale all'aggiunta di una settimana.

```
WTO >select sysdate, next_day(sysdate, 'domenica')
  2  from dual;

SYSDATE             NEXT_DAY(SYSDATE,'D
------------------- -------------------
13-02-2011 18:09:36 20-02-2011 18:09:36
```

La funzione NEXT_DAY è ovviamente dipendente dalla lingua della sessione di lavoro. Se, infatti, modifichiamo la lingua della sessione di lavoro dall'italiano all'inglese

```
WTO >alter session set nls_language='english';

Session altered.
```

Non è possibile più usare il giorno in italiano ma bisognerà adattarsi all'inglese.

```
WTO >select sysdate, next_day(sysdate, 'martedi')
  2  from dual;
select sysdate, next_day(sysdate, 'martedi')
                                  *
ERROR at line 1:
ORA-01846: not a valid day of the week


WTO >select sysdate, next_day(sysdate, 'tuesday')
  2  from dual;

SYSDATE             NEXT_DAY(SYSDATE,'T
------------------- -------------------
13-02-2011 18:12:21 15-02-2011 18:12:21
```

- ROUND e TRUNC

La funzione ROUND arrotonda una data al giorno più vicino.

```
WTO >select round(sysdate) from dual;

ROUND(SYSDATE)
-------------------
14-02-2011 00:00:00
```

Si può opzionalmente specificare un secondo parametro in cui si indica se arrotondare ad anno, mese, giorno, ora, minuto, secondo.

La sintassi con cui va specificato questo parametro è quella di un "elemento di formato". Gli elementi di formato, già incontrati negli esempio precedenti, saranno trattati nel dettaglio più avanti. Nell'esempio seguente si arrotonda la SYSDATE alle ore.

```
WTO >select sysdate, round(sysdate,'hh')
  2  from dual;

SYSDATE             ROUND(SYSDATE,'HH')
------------------- -------------------
13-02-2011 18:15:05 13-02-2011 18:00:00
```

La funzione TRUNC opera come la ROUND ma applica un troncamento anziché un arrotondamento.

```
WTO >select trunc(sysdate) from dual;

TRUNC(SYSDATE)
-------------------
13-02-2011 00:00:00

WTO >select sysdate, trunc(sysdate,'hh')
  2  from dual;

SYSDATE             TRUNC(SYSDATE,'HH')
------------------- -------------------
13-02-2011 18:19:02 13-02-2011 18:00:00
```

- CURRENT_DATE e CURRENT_TIMESTAMP

La funzione CURRENT_DATE restituisce la data corrente applicando il fuso orario della sessione (laddove invece la SYSDATE applica sempre il fuso orario del database). SYSDATE è diversa da CURRENT_DATE solo quando il client che instanzia la sessione ed il server si trovano in due fusi orari differenti. In una situazione standard (client e server nello stesso fuso orario) si avrà.

```
WTO >select sysdate, current_date
  2  from dual;

SYSDATE             CURRENT_DATE
------------------- -------------------
13-02-2011 18:21:04 13-02-2011 18:21:05
```

Se invece simuliamo di trovarci a Los Angeles (fuso orario -8h rispetto all'ora di Greenwich)

```
WTO >alter session set time_zone='-8:0';

Session altered.
```

Otterremo un risultato differente.

```
WTO >select sysdate, current_date
  2  from dual;

SYSDATE             CURRENT_DATE
------------------- -------------------
13-02-2011 18:22:21 13-02-2011 09:22:22
```

Come CURRENT_DATE anche CURRENT_TIMESTAMP restituisce il TIMESTAMP attuale nel fuso orario del client.

```
WTO >alter session set time_zone='1:0';

Session altered.

WTO >select systimestamp, current_timestamp
  2  from dual;

SYSTIMESTAMP                        CURRENT_TIMESTAMP
----------------------------------- -----------------------------------
13-FEB-11 18:26:35,478507 +01:00    13-FEB-11 18:26:35,478528 +01:00
```

```
WTO >alter session set time_zone='-8:0';

Session altered.

WTO >select systimestamp, current_timestamp
  2  from dual;

SYSTIMESTAMP                        CURRENT_TIMESTAMP
----------------------------------- -----------------------------------
13-FEB-11 18:26:46,593435 +01:00    13-FEB-11 09:26:46,593459 -08:00
```

- EXTRACT

La funzione EXTRACT consente di estrarre da un TIMESTAMP uno degli elementi che lo compone, ad esempio il mese, il giorno oppure il minuto. Gli elementi che possono essere estratti da un TIMESTAMP sono: l'anno (YEAR), il mese (MONTH), il giorno (DAY), l'ora (HOUR), il minuto (MINUTE), il secondo (SECOND), l'ora del fuso orario (TIMEZONE_HOUR), il minuto del fuso orario (TIMEZONE_MINUTE), la regione del fuso orario (TIMEZONE_REGION), la stringa d'abbreviazione del fuso orario (TIMEZONE_ABBR).

Se si prova ad applicare la funzione ad una data anziché un timestamp si ottiene un errore:

```
WTO >select extract(hour from data_fattura) from fatture;
select extract(hour from data_fattura) from fatture
                         *
ERRORE alla riga 1:
ORA-30076: invalid extract field for extract source
```

Per questo motivo creiamo una tabella T contenente una colonna COL di tipo TIMESTAMP.

```
WTO >create table t (col timestamp);

Tabella creata.
```

Inseriamo in tabella il timestamp corrente.

```
WTO >insert into t values (systimestamp);

Creata 1 riga.
```

E finalmente estraiamo dal dato inserito solo l'ora.

```
WTO >select extract(hour from col) from t;

EXTRACT(HOURFROMCOL)
--------------------
                  18
```

Estraendo i secondi vengono estratte anche le frazioni di secondo.

```
WTO >select extract(second from col),
  2  extract(second from systimestamp) from t;

EXTRACT(SECONDFROMCOL) EXTRACT(SECONDFROMSYSTIMESTAMP)
```

```
--------------------- ------------------------------
            54,280232                         1,345504
```

Ovviamente, poiché ogni data può essere vista come un particolare TIMESTAMP avente a zero le frazioni di secondo, è possibile applicare la funzione EXTRACT anche ad un campo di tipo DATE purché esso sia prima esplicitamente convertito ad un TIMESTAMP utilizzando la funzione di conversione TO_TIMESTAMP. Questa funzione sarà descritta meglio più avanti.

```
WTO >select extract(hour from to_timestamp(data_fattura))
  2  from fatture;

EXTRACT(HOURFROMTO_TIMESTAMP(DATA_FATTURA))
-------------------------------------------
                                          0
                                          0
                                          0
                                          0
                                          0

WTO >select extract(day from to_timestamp(data_fattura))
  2  from fatture;

EXTRACT(DAYFROMTO_TIMESTAMP(DATA_FATTURA))
------------------------------------------
                                         1
                                         1
                                        20
                                         1
                                         1

WTO >select extract(month from to_timestamp(data_fattura))
  2  from fatture;

EXTRACT(MONTHFROMTO_TIMESTAMP(DATA_FATTURA))
--------------------------------------------
                                          10
                                          12
                                          10
                                           2
                                          12
```

7.5.6 Funzioni di conversione

Le funzioni di conversione consentono di trasformare un tipo di dato in un altro. Ad esempio una stringa in data oppure un numero in stringa.

Alcune conversioni di tipo, comunque vengono effettuate automaticamente da Oracle, si tratta delle cosiddette “conversioni implicite”. Un esempio di conversione implicita è ciò che avviene quando si utilizza una stringa tra apici per rappresentare una data. Nell’esempio che segue si utilizza la funzione LAST_DAY sulla stringa ’12-NOV-10’. Oracle interpreta la stringa come data e determina l’ultimo giorno di Novembre 2010

```
WTO >select last_day('12-NOV-10') from dual;

LAST_DAY(
```

```
---------
30-NOV-10
```

La stringa ’12-NOV-10’ è stata implicitamente convertita in data.

Allo stesso modo si può sommare un numero ad una stringa. Se tale stringa rappresenta un numero valido Oracle effettua una conversione implicita da stringa a numero ed esegue la somma, viceversa solleva un errore.

```
WTO >select '12,3'+5 from dual;

  '12,3'+5
----------
      17,3

WTO >select '12x'+5 from dual;
select '12x'+5 from dual
       *
ERRORE alla riga 1:
ORA-01722: invalid number
```

È buona pratica di programmazione non fare affidamento sulle conversioni implicite ed usare sempre le funzioni di conversione che saranno presentate nel seguito.

- TO_CHAR

La funzione TO_CHAR consente di convertire in stringa un numero oppure una data.

```
WTO >select to_char(sysdate) from dual;

TO_CHAR(S
---------
22-FEB-11

WTO >select to_char(5.3) from dual;
TO_
---
5,3
```

Quando si utilizza la TO_CHAR è possibile specificare come secondo parametro della funzione il formato grafico in cui il numero o la data deve essere rappresentato. I formati di rappresentazione delle date e dei numeri saranno approfonditi nei due paragrafi successivi. Al momento per un semplice esempio utilizzeremo il formato dd/mm/yyyy per le date (significa: due cifre per il giorno, uno slash, due cifre per il mese, uno slash, quattro cifre per l’anno) ed il formato 999g999d99 per i numeri (vuol dire che bisogna evidenziare il separatore di migliaia e rappresentare i numeri con due decimali).

```
WTO >select to_char(sysdate,'dd/mm/yyyy') from dual;

TO_CHAR(SY
----------
22/02/2011

WTO >select to_char(5.3,'999g999d99') from dual;
```

```
TO_CHAR(5.3
-----------
       5,30

WTO >select to_char(12345.3,'999g999d99') from dual;
TO_CHAR(123
-----------
  12.345,30
```

- TO_DATE

La funzione TO_DATE consente di convertire una stringa in data.

```
WTO >select to_date('12-nov-10') from dual;

TO_DATE('
---------
12-NOV-10
```

Se la funzione viene utilizzata con un solo parametro, la stringa da convertire in data deve essere espressa nel formato di rappresentazione delle date in quel momento utilizzato dal DB. Nel nostro esempio dd-mon-yy cioè due cifre per il giorno, un trattino, tre lettere per il mese, un trattino, due cifre per l’anno. Nel caso in cui si utilizzasse un formato differente la funzione TO_DATE causerebbe un errore.

```
WTO >select to_date('12/11/2010') from dual;
select to_date('12/11/2010') from dual
               *
ERRORE alla riga 1:
ORA-01843: not a valid month
```

Per evitare questo errore e fornire alla TO_DATE una stringa in qualsivoglia formato, è sufficiente passare come secondo parametro il formato di rappresentazione della data in cui il primo parametro è espresso.

```
WTO >select to_date('12/11/2010','dd/mm/yyyy') from dual;

TO_DATE('
---------
12-NOV-10
```

- TO_NUMBER

La funzione TO_NUMBER converte una stringa in numero. Anche in questo caso è opzionalmente possibile indicare il formato numerico in cui la stringa è espressa.

```
WTO >select to_number('123,4') from dual;

TO_NUMBER('123,4')
------------------
             123,4

WTO >select to_number('123.123,4') from dual;
select to_number('123.123,4') from dual
                 *
ERRORE alla riga 1:
ORA-01722: invalid number

WTO >select to_number('123.123,4','999g999d99') from dual;
```

```
TO_NUMBER('123.123,4','999G999D99')
-----------------------------------
                           123123,4
```

- TO_TIMESTAMP

La funzione TO_TIMESTAMP è molto simile alla TO_DATE ma trasforma una stringa in TIMESTAMP.

```
WTO >select to_timestamp('12-nov-10 22:10:33,123456')
  2  from dual;

TO_TIMESTAMP('12-NOV-1022:10:33,123456')
-----------------------------------------------------------------
12-NOV-10 22:10:33,123456000
```

7.5.7 Formati per la rappresentazione delle date

Un formato di rappresentazione di una data è una stringa con cui si comunica ad Oracle come deve essere formattata una data. Un formato di rappresentazione si ottiene combinando in una sola stringa uno o più codici di formato. Ogni codice di formato consente di formattare, in un formato specifico, un elemento della data.

Nel seguito vengono presentati i codici di formato utilizzati più di frequente. Per una lista completa si può fare riferimento al manuale Oracle SQL Reference.

Il formato ‘d’ stampa il giorno della settimana da uno a sette. Uno per il lunedì, sette per la domenica. Tale impostazione dipende dal parametro di sessione NLS_TERRITORY. Per default in una installazione in italiano tale parametro vale ITALY e dunque il primo giorno della settimana è lunedì. Se impostiamo tale parametro al valore AMERICA il primo giorno della settimana sarà la domenica e dunque il martedì è il terzo giorno della settimana.

```
WTO >select sysdate from dual;

SYSDATE
---------
22-FEB-11

WTO >select to_char(sysdate,'d') from dual;

T
-
2

WTO >alter session set nls_territory='AMERICA';
Modificata sessione.

WTO >select to_char(sysdate,'d') from dual;

T
-
3

WTO >alter session set nls_territory='ITALY';
Modificata sessione.
```

Il codice di formato ‘dd’ stampa il giorno del mese a due cifre.

```
WTO >select to_char(sysdate,'dd') from dual;

TO
--
22
```

Il codice di formato ‘ddd’ stampa il giorno dell’anno da uno a 366.

```
WTO >select to_char(sysdate,'ddd') from dual;

TO_
---
053
```

Il codice di formato ‘mm’ stampa il mese dell’anno. Da uno a dodici.

```
WTO >select to_char(sysdate,'mm') from dual;

TO
--
02
```

Il codice ‘mon’ stampa tre lettere che rappresentano il nome abbreviato del mese. Il risultato dipende dal valore del parametro NLS_LANGUAGE.

```
WTO >alter session set nls_language='ENGLISH';

Session altered.

WTO >select to_char(date'2010-05-01','mon') from dual;

TO_
---
may

WTO >alter session set nls_language='ITALIAN';

Modificata sessione.

WTO >select to_char(date'2010-05-01','mon') from dual;

TO_
---
Mag
```

Il codice ‘month’ rappresenta il nome del mese, anche in questo caso il risultato dipende dalla lingua.

```
WTO >alter session set nls_language='ENGLISH';

Session altered.

WTO >select to_char(date'2010-05-01','month') from dual;

TO_CHAR(D
---------
May

WTO >alter session set nls_language='ITALIAN';
```

```
Modificata sessione.

WTO >select to_char(date'2010-05-01','month') from dual;

TO_CHAR(D
---------
maggio
```

Il codice ‘y’ stampa l’ultima cifra dell’anno.

```
WTO >select to_char(sysdate,'y') from dual;

T
-
1
```

Il codice ‘yy’ stampa le ultime due cifre dell’anno.

```
WTO >select to_char(sysdate,'yy') from dual;

TO
--
11
```

Il codice ‘yyy’ stampa le ultime tre cifre dell’anno.

```
WTO >select to_char(sysdate,'yyy') from dual;

TO_
---
011
```

Il codice ‘yyyy’ stampa l’anno a quattro cifre.

```
WTO >select to_char(sysdate,'yyyy') from dual;

TO_C
----
2011
```

Il codice ‘hh’ rappresenta l’ora nel formato 0-12.

```
WTO >select to_char(sysdate,'hh') from dual;

TO
--
10
```

Il codice ‘hh’ rappresenta l’ora nel formato a 24 ore.

```
WTO >select to_char(sysdate,'hh24') from dual;

TO
--
22
```

Il codice ‘mi’ rappresenta i minuti all’interno dell’ora. Da zero a 59.

```
WTO >select to_char(sysdate,'mi') from dual;
```

```
TO
--
42
```

Il codice ‘ss’ rappresenta i secondi all’interno del minuto. Da zero a 59.

```
WTO >select to_char(sysdate,'ss') from dual;

TO
--
42
```

Il codice ‘sssss’ rappresenta i secondi a partire dalla mezzanotte. Da zero a 86399.

```
WTO >select to_char(sysdate,'sssss') from dual;

TO_CH
-----
81769
```

Sulla base dei codici di formato appena elencati è possibile, ad esempio, definire la seguente stringa di formato.

```
WTO >select to_char(sysdate,'dd-mm-yyyy hh24:mi:ss')
  2  from dual;

TO_CHAR(SYSDATE,'DD
-------------------
22-02-2011 22:43:24
```

Oltre ai singoli codici di formato, in una stringa di formato possono apparire anche i caratteri trattino (-), slash (/), virgola (,), punto (.), punto e virgola (;), e due punti(:).

Inoltre, all’interno di una stringa di formato, è possibile aggiungere una qualunque altra stringa, basta metterla all’interno di doppi apici. Ad esempio si può scrivere.

```
WTO >select to_char(sysdate,'"oggi e'' il "dd "di "month.')
  2  from dual;

TO_CHAR(SYSDATE,'"OGGIE''IL
---------------------------
oggi e' il 22 di febbraio .
```

7.5.8 Formati per la rappresentazione dei numeri

Come per le date anche i numeri hanno i loro formati di rappresentazione. La cosa è però più semplice. In una stringa di formato per i numeri si utilizzano: la lettera “g” per rappresentare il separatore di migliaia; la lettera “d” per rappresentare il separatore di decimali; la cifra “9” o la cifra “0” per rappresentare le cifre che compongono il numero.

```
WTO >select to_char(123456.234,'9g999g999d999')
  2  from dual;
```

```
TO_CHAR(123456
--------------
   123.456,234

WTO >select to_char(123456.2,'9g999g999d999')
  2  from dual;

TO_CHAR(123456
--------------
   123.456,200

WTO >select to_char(123456.234,'0g000g000d000')
  2  from dual;

TO_CHAR(123456
--------------
 0.123.456,234

WTO >select to_char(123456.2,'0g000g000d000')
  2  from dual;

TO_CHAR(123456
--------------
 0.123.456,200
```

Come si vede dagli esempi la differenza tra "0" e "9" è la seguente. Quando si usa "0" vengono stampate le cifre intere non significative, quando si usa il "9" le cifre intere non significative sono sostituite con uno spazio. Questa differenza di comportamento è particolarmente utile per gestire la cifra delle unità quando il numero è minore di uno.

```
WTO >select to_char(0.3,'999d99')
  2  from dual;

TO_CHAR
-------
    ,30

WTO >select to_char(0.3,'000d00')
  2  from dual;

TO_CHAR
-------
 000,30

WTO >select to_char(0.3,'990d00')
  2  from dual;

TO_CHAR
-------
   0,30
```

7.5.9 Gestione dei valori nulli

Come è stato già anticipato quando si è parlato degli operatori, bisogna fare particolare attenzione con i valori NULL. Per agevolare il trattamento dell'assenza di valore Oracle mette a disposizione diverse funzioni che vengono di seguito descritte.

- NVL

La funzione NVL consente specificare due parametri. La funzione restituisce il primo se questo è NOT NULL, altrimenti il secondo.

```
WTO >select cod_fisc from clienti;

COD_FISC
----------------
RSSMRC70R20H501X

VRDMTT69S02H501X
NRILCU77A22F205X

SPSGNN71B10F839X
RSSPSQ70C14F839X


Selezionate 8 righe.

WTO >select nvl(cod_fisc,'VALORE NON DISPONIBILE')
  2  from clienti;

NVL(COD_FISC,'VALORENO
----------------------
RSSMRC70R20H501X
VALORE NON DISPONIBILE
VRDMTT69S02H501X
NRILCU77A22F205X
VALORE NON DISPONIBILE
SPSGNN71B10F839X
RSSPSQ70C14F839X
VALORE NON DISPONIBILE

Selezionate 8 righe.
```

La funzione può essere applicata anche a numeri e date.

```
WTO >create table test (a varchar2(10), b number, c date);
Tabella creata.

WTO >insert into test values ('test1',1,sysdate);
Creata 1 riga.

WTO >insert into test values ('test2',null,sysdate);
Creata 1 riga.

WTO >insert into test values ('test3',3,null);
Creata 1 riga.

WTO >insert into test values ('test4',null,null);
Creata 1 riga.

WTO >select * from test;

A                   B C
---------- ---------- ---------
test1               1 22-FEB-11
test2                 22-FEB-11
test3               3
test4

WTO >select a, nvl(b,9), nvl(c,sysdate+3)
```

```
  2  from test;

A            NVL(B,9) NVL(C,SYS
---------- ---------- ---------
test1               1 22-FEB-11
test2               9 22-FEB-11
test3               3 25-FEB-11
test4               9 25-FEB-11
```

- NVL2

La funzione NVL2 riceve in input tre parametri. Se il primo parametro è diverso da NULL ritorna il secondo parametro, altrimenti il terzo.

```
WTO >select nvl2(cod_fisc,'VALORIZZATO','NON VALORIZZATO')
  2  from clienti;

NVL2(COD_FISC,'
---------------
VALORIZZATO
NON VALORIZZATO
VALORIZZATO
VALORIZZATO
NON VALORIZZATO
VALORIZZATO
VALORIZZATO
NON VALORIZZATO

Selezionate 8 righe.

WTO >select nvl2(b,'VALORIZZATO','NON VALORIZZATO')
  2  from test;

NVL2(B,'VALORIZ
---------------
VALORIZZATO
NON VALORIZZATO
VALORIZZATO
NON VALORIZZATO

WTO >select nvl2(c,'VALORIZZATO','NON VALORIZZATO')
  2  from test;

NVL2(C,'VALORIZ
---------------
VALORIZZATO
VALORIZZATO
NON VALORIZZATO
NON VALORIZZATO
```

- COALESCE

La funzione COALESCE è una generalizzazione della NVL. Riceve in input un numero a piacere di espressioni e ritorna la prima che è diversa da NULL.

```
WTO >drop table test;
Tabella eliminata.

WTO >create table test (a varchar2(10), b varchar2(10),
  2  c varchar2(10), d varchar2(10));
Tabella creata.
```

```
WTO >insert into test values ('A','B','C','D');
Creata 1 riga.

WTO >insert into test values (null,'B','C','D');
Creata 1 riga.
WTO >insert into test values (null,null,'C','D');
Creata 1 riga.

WTO >insert into test values (null,null,null,'D');
Creata 1 riga.

WTO >insert into test values (null,null,null,null);
Creata 1 riga.

WTO >set null @
--IL COMANDO "SET NULL @" DICE A SQL*PLUS DI VISUALIZZARE I NULL
--CON LE @.

WTO >select * from test;

A          B          C          D
---------- ---------- ---------- ----------
A          B          C          D
@          B          C          D
@          @          C          D
@          @          @          D
@          @          @          @

WTO >select coalesce(a,b,c,d) from test;

COALESCE(A
----------
A
B
C
D
@
```

Quando le espressioni passate in input alla COALESCE sono due la funzione si comporta esattamente come la NVL.

```
WTO >select coalesce(cod_fisc,'NON VALORIZZATO')
  2  from clienti;

COALESCE(COD_FIS
----------------
RSSMRC70R20H501X
NON VALORIZZATO
VRDMTT69S02H501X
NRILCU77A22F205X
NON VALORIZZATO
SPSGNN71B10F839X
RSSPSQ70C14F839X
NON VALORIZZATO

Selezionate 8 righe.
```

- LNNVL

La funzione LNNVL riceve in input una condizione e ritorna TRUE quando la condizione è vera oppure quando la condizione a falsa a causa di

valori NULL. Se, ad esempio, cerchiamo tutti i codici fiscali che cominciano per R otteniamo

```
WTO >select nome, cod_fisc from clienti;

NOME     COD_FISC
-------- ----------------
MARCO    RSSMRC70R20H501X
GIOVANNI @
MATTEO   VRDMTT69S02H501X
LUCA     NRILCU77A22F205X
AMBROGIO @
GENNARO  SPSGNN71B10F839X
PASQUALE RSSPSQ70C14F839X
VINCENZO @
Selezionate 8 righe.

WTO >select nome from clienti
  2  where cod_fisc like 'R%';

NOME
--------
MARCO
PASQUALE
```

Ci sono solo due codici che cominciano per R. Ce ne sono altri tre che non cominciano per R e ce ne sono tre che sono UNKNOWN, non si sa se cominciano per R oppure no perché il dato non è presente nel sistema. Noi potremmo voler vedere i nomi dei clienti cha hanno il codice fiscale che comincia per R oppure non ha affatto il codice fiscale. Possiamo ottenere questo risultato utilizzando nella WHERE la funzione LNNVL

```
WTO >select nome from clienti
  2  where lnnvl(cod_fisc like 'R%');

NOME
--------
GIOVANNI
MATTEO
LUCA
AMBROGIO
GENNARO
VINCENZO
Selezionate 6 righe.
```

In pratica la LNNVL tratta condizioni UNKNOWN come se fossero TRUE, mentre in generale le condizioni UNKNOWN sono trattate come le FALSE.

- NULLIF

La funzione NULLIF riceve in input due parametri e ritorna NULL quando i due parametri sono uguali.

```
WTO >select nome, nullif(nome,'MARCO')
  2  from clienti;

NOME     NULLIF(NOME,'MARCO')
-------- ------------------------------
MARCO    @
GIOVANNI GIOVANNI
```

```
MATTEO   MATTEO
LUCA     LUCA
AMBROGIO AMBROGIO
GENNARO  GENNARO
PASQUALE PASQUALE
VINCENZO VINCENZO

Selezionate 8 righe.

WTO >select cod_fisc, nullif(cod_fisc,NULL)
  2  from clienti;

COD_FISC         NULLIF(COD_FISC,
---------------- ----------------
RSSMRC70R20H501X RSSMRC70R20H501X
@                @
VRDMTT69S02H501X VRDMTT69S02H501X
NRILCU77A22F205X NRILCU77A22F205X
@                @
SPSGNN71B10F839X SPSGNN71B10F839X
RSSPSQ70C14F839X RSSPSQ70C14F839X
@                @

Selezionate 8 righe.
```

7.5.10 Espressioni regolari

Quest'argomento è considerato di livello avanzato. Per comprendere pienamente gli esempi è necessario conoscere argomenti di SQL e PL/SQL che in questa fase del manuale ancora non sono stati introdotti. Il lettore principiante può saltare questo paragrafo e tornarci alla fine del libro quando avrà acquisito gli strumenti necessari ad affrontarlo.

A partire dalla versione 10 di Oracle sono state aggiunte ad SQL e PL/SQL nuove funzionalità che consentono di utilizzare le espressioni regolari in maniera conforme a quanto definito dallo standard POSIX.

In particolare nella versione 10g sono state introdotte le funzioni REGEXP_INSTR, REGEXP_REPLACE e REGEXP_SUBSTR e la condizione REGEXP_LIKE.

Successivamente, nella versione 11g, è stata aggiunta anche la funzione REGEXP_COUNT. In questo paragrafo si vedranno alcuni esempi d'utilizzo.

- REGEXP_INSTR

La funzione REGEXP_INSTR funziona più o meno come la INSTR: serve a cercare, all'interno di una stringa data, una sottostringa che verifica il pattern della regex data in input.

Se non trova nessuna sottostringa che verifica la regex ritorna 0.

Il prototipo della funzione è

```
REGEXP_INSTR (source_string, pattern, position, occurrence,
return_option, match_parameter)
```

Dove

- source_string è la stringa in cui cercare;
- pattern è l'espressione regolare;
- position è la posizione a partire dalla quale bisogna cercare;
- occurrence è l'occorrenza richiesta;
- return_option può valere 0 se si desidera la posizione del primo carattere della sottostringa trovata, 1 se si desidera la posizione del primo carattere successivo alla sottostringa trovata;
- match_parameter è un flag che può valere:
    - ✓ i (ricerca case-insensitive),
    - ✓ c (ricerca case-sensitive),
    - ✓ n (il carattere jolly '.' trova anche il ritorno a capo),
    - ✓ m (interpreta la stringa in modalità multi-linea)

Solo i primi due parametri sono obbligatori.

Facciamo un esempio: cerchiamo nella stringa 'Questa è una stringa di prova che mi consente di illustrare le regex in Oracle' la seconda parola composta di due lettere:

```
WTO >WITH T AS (
  2  SELECT 'Questa è una stringa di prova che mi consente
  3  di illustrare le regex in Oracle' str from dual)
  4  Select str, REGEXP_INSTR(str,'(^|\ )[[:alpha:]]{2}($|\ )',
  5  1, 2)+1 pos
  6  FROM t;

STR                                                       POS
-------------------------------------------------- ----------
Questa è una stringa di prova che mi consente              35
di illustrare le regex in Oracle
```

- REGEXP_REPLACE

La funzione REGEXP_REPLACE consente di sostituire una stringa trovata mediante espressione regolare con un'altra stringa data:

```
REGEXP_REPLACE(source_string, pattern, replace_string, position,
occurrence, match_parameter)
```

Dove

- source_string è la stringa in cui effettuare la ricerca e la sostituzione;
- pattern è l'espressione regolare;
- replace_string è la stringa da sostituire a quanto trovato mediante il pattern;
- position è la posizione a partire dalla quale bisogna cercare;
- occurrence è l'occorrenza richiesta;
- match_parameter come nella REGEXP_INSTR.

Se quindi vogliamo sostituire, nella stringa dell'esempio precedente, la seconda parola di due caratteri con una X, dobbiamo fare:

```
WTO >WITH T AS (
  2  SELECT 'Questa è una stringa di prova che mi consente
  3  di illustrare le regex in Oracle' str from dual)
  4  Select REGEXP_REPLACE(str,'(^|\ )[[:alpha:]]{2}($|\ )',
  5  ' X ',1, 2) newstr
  6  FROM t;

NEWSTR
--------------------------------------------------------------
Questa è una stringa di prova che X consente
di illustrare le regex in Oracle
```

- REGEXP_SUBSTR

La funzione REGEXP_SUBSTR consente di estrarre una sottostringa:

```
REGEXP_SUBSTR (source_string , pattern, position, occurrence,
match_parameter)
```

Dove

- source_string è la stringa in cui effettuare la ricerca e da cui estrarre la sottostringa;
- pattern è l'espressione regolare;
- position è la posizione a partire dalla quale bisogna cercare;
- occurrence è l'occorrenza richiesta;
- match_parameter come nella REGEXP_INSTR.

Quindi per estrarre la seconda parola da due caratteri:

```
WTO >WITH T AS (
  2  SELECT 'Questa è una stringa di prova che mi
  3  consente di illustrare le regex in Oracle' str from dual)
  4  Select str,
  5  trim(REGEXP_SUBSTR(str,'(^|\ )[[:alpha:]]{2}($|\ )'
  6  ,1, 2)) substr
  7* FROM t

STR                                                 SUBSTR
--------------------------------------------------- ------
Questa è una stringa di prova che mi                di
consente di illustrare le regex in Oracle
```

- REGEXP_COUNT

La funzione REGEXP_COUNT ci consente di contare le occorrenze nella source string che verificano il pattern:

```
REGEXP_COUNT (source_char , pattern, position, match_param)
```

Dove

- source_char è la stringa in cui effettuare la ricerca;
- pattern è l'espressione regolare;
- position è la posizione a partire dalla quale bisogna cercare;

- return_option come nella REGEXP_INSTR.

Quindi per ottenere il numero di parole di due caratteri:

```
WTO> WITH T AS (
  2  SELECT 'Questa è una stringa di prova che mi consente
  3  di illustrare le regex in Oracle' str from dual)
  4  Select str,
  5  REGEXP_COUNT(str,'(^|\ )[[:alpha:]]{2}($|\ )',1) count
  6  FROM t;

STR                                                     COUNT
-------------------------------------------------- ----------
Questa è una stringa di prova che mi consente               4
di illustrare le regex in Oracle
```

Possiamo usare in modalità combinata la REGEXP_COUNT e la REGEXP_SUBSTR (ed un trucchetto della connect by) per estrarre in un colpo solo tutte le parole di 2 caratteri:

```
WTO> WITH T AS (
  2  SELECT 'Questa è una stringa di prova che mi consente
  3   di illustrare le regex in Oracle' str from dual)
  4  Select str,
  5  trim(REGEXP_SUBSTR(str,'(^|\ )[[:alpha:]]{2}($|\ )',1, level)) substr
  6  FROM t
  7  connect by level<=REGEXP_COUNT(str,'(^|\ )[[:alpha:]]{2}($|\ )',1);

STR                                                SUBSTR
-------------------------------------------------- ------
Questa è una stringa di prova che mi consente      di
 di illustrare le regex in Oracle

Questa è una stringa di prova che mi consente      mi
 di illustrare le regex in Oracle

Questa è una stringa di prova che mi consente      di
 di illustrare le regex in Oracle

Questa è una stringa di prova che mi consente      le
 di illustrare le regex in Oracle

Questa è una stringa di prova che mi consente      in
 di illustrare le regex in Oracle
```

- REGEXP_LIKE

La condizione REGEXP_LIKE torna boolean e può essere utilizzata nella WHERE o nella HAVING di una query:

```
REGEXP_LIKE (source_string, pattern, match_parameter)
```

Dove

- source_string è la stringa in cui effettuare la ricerca;
- pattern è l'espressione regolare;
- match_parameter come nella REGEXP_INSTR.

REGEXP_LIKE(str,ptrn,mp) è logicamente equivalente a REGEXP_INSTR(str,ptrn,1,1,mp)>0

Tiriamo per esempio fuori tutte le stringhe che hanno almeno una parola da due caratteri:

```
WTO> WITH T AS (
  2  SELECT 'Stringa senza parole di2' str from dual union
  3  SELECT 'prima Stringa con parole di 2 caratteri' from dual
  4  union
  5  SELECT 'seconda Stringa con parole di 2 caratteri' from dual
  6  union
  7  SELECT 'Altra Stringa senza parole di2' from dual
  8  )
  9  Select str
 10  from t
 11  where REGEXP_LIKE(str,'(^|\ )[[:alpha:]]{2}($|\ )');

STR
--------------------------------------------------
prima Stringa con parole di 2 caratteri
seconda Stringa con parole di 2 caratteri
```

Come detto sarebbe stato del tutto equivalente scrivere:

```
WTO> WITH T AS (
  2  SELECT 'Stringa senza parole di2' str from dual union
  3  SELECT 'prima Stringa con parole di 2 caratteri' from dual
  4  union
  5  SELECT 'seconda Stringa con parole di 2 caratteri' from dual
  6  union
  7  SELECT 'Altra Stringa senza parole di2' from dual
  8  )
  9  Select str
 10  from t
 11  where REGEXP_INSTR(str,'(^|\ )[[:alpha:]]{2}($|\ )')>0;

STR
--------------------------------------------------
prima Stringa con parole di 2 caratteri
seconda Stringa con parole di 2 caratteri
```

7.5.11 DECODE e CASE

- DECODE

La funzione DECODE consente di decodificare un insieme di valori in un insieme di nuovi valori corrispondenti. La funzione riceve in input come primo parametro un dato da controllare. Oltre al dato da controllare è possibile passare alla funzione DECODE un numero indefinito di coppie valore-risultato. Per ogni coppia, se il dato da controllare è uguale al valore la funzione DECODE restituisce il risultato. La funzione si applica a stringhe, numeri e date.

Nella tabella FATTURE c'è il campo PAGATA che vale S o N ed indica se la fattura è stata pagata o no.

```
WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
```

```
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S
```

Immaginiamo di volere estrarre il numero della fattura, la data della fattura e la scritta PAGATA o NON PAGATA.

```
WTO >select num_fattura, data_fattura,
  2  decode(pagata,'S','PAGATA','N','NON PAGATA')"pagata?"
  3  from fatture;

NUM_FATTURA DATA_FATT pagata?
----------- --------- ----------
          1 01-OTT-10 PAGATA
          2 01-DIC-10 NON PAGATA
          3 20-OTT-10 PAGATA
          4 01-FEB-11 NON PAGATA
          5 01-DIC-10 PAGATA
```

Il dato da controllare è il campo PAGATA. Se questo vale 'S' bisogna estrarre 'PAGATA', se questo vale 'N' bisogna estrarre 'NON PAGATA'.

La funzione DECODE consente anche di indicare come ultimo parametro, alla fine delle coppie valore-risultato, un valore di default da ritornare nel caso in cui il dato da controllare non sia uguale a nessuno dei valori. Sempre in riferimento le fatture, ipotizziamo di volere etichettare la fattura numero uno con 'PRIMA', la numero due con 'SECONDA' e tutte le altre con 'SUCCESSIVE'.

```
WTO >select num_fattura, data_fattura,
  2  decode(num_fattura,1,'PRIMA',2,'SECONDA','SUCCESSIVE') posiz
  3  from fatture;

NUM_FATTURA DATA_FATT POSIZ
----------- --------- ----------
          1 01-OTT-10 PRIMA
          2 01-DIC-10 SECONDA
          3 20-OTT-10 SUCCESSIVE
          4 01-FEB-11 SUCCESSIVE
          5 01-DIC-10 SUCCESSIVE
```

Il risultato 'SUCCESSIVE' è utilizzato tutte le volte che il dato da controllare è diverso da tutti i valori gestiti precedentemente.

Anche i valori nulli possono essere decodificati.

```
WTO >select nome, decode(cod_fisc,null,'NO CF','CON CF') "CF?"
  2  from clienti;

NOME     CF?
-------- ------
MARCO    CON CF
GIOVANNI NO CF
MATTEO   CON CF
LUCA     CON CF
AMBROGIO NO CF
GENNARO  CON CF
PASQUALE CON CF
VINCENZO NO CF

Selezionate 8 righe.
```

Sebbene la funzione DECODE consenta solo confronti con insiemi discreti di valori, con un po' di fantasia essa può essere utilizzata anche per gestire insiemi continui. Ipotizziamo ad esempio di voler, per ogni fattura, indicare se essa ha un importo minore, maggiore o uguale a 500 euro.

```
WTO >select num_fattura, importo,
  2  decode(sign(importo-500),-1,'<500',0,'=500',1,'>500') fascia
  3  from fatture;

NUM_FATTURA    IMPORTO FASCIA
----------- ---------- ------
          1        300 <500
          2        500 =500
          3        700 >500
          4       1000 >500
          5        500 =500
```

In questo caso l'utilizzo della funzione SIGN sull'espressione IMPORTO-500 consente di ottenere un insieme discreto di valori (-1, 0 e 1) su cui si basa la DECODE per ottenere risultato.

- CASE

La struttura condizionale CASE (non è una funzione) è una generalizzazione della DECODE. Questa struttura può essere espressa in due modi. Nel primo modo ricalca molto da vicino la DECODE.

```
SELECT CASE <dato da controllare>
       WHEN <valore1> THEN <risultato1>
       WHEN <valore2> THEN <risultato2>
...
       WHEN <valoreN> THEN <risultatoN>
       ELSE <risultato di default>
       END
  FROM <tabella>
```

Consente dunque di valutare l'uguaglianza del dato da controllare con uno dei valori di confronto.

La seconda struttura aggiunge la potenzialità di valutare qualunque condizione.

```
SELECT CASE
       WHEN <condizione1> THEN <risultato1>
       WHEN <condizione2> THEN <risultato2>
...
       WHEN <condizioneN> THEN <risultatoN>
       ELSE <risultato di default>
       END
  FROM <tabella>
```

Ripresentiamo dapprima tutti gli esempi eseguiti con la DECODE nella struttura CASE di primo tipo.

```
WTO >select num_fattura, data_fattura,
  2  CASE pagata when 'S' then 'PAGATA'
  3  when 'N' then 'NON PAGATA' END "pagata?"
  4  from fatture;

NUM_FATTURA DATA_FATT pagata?
----------- --------- ----------
          1 01-OTT-10 PAGATA
          2 01-DIC-10 NON PAGATA
          3 20-OTT-10 PAGATA
          4 01-FEB-11 NON PAGATA
          5 01-DIC-10 PAGATA

WTO >select num_fattura, data_fattura,
  2  CASE num_fattura when 1 then 'PRIMA'
  3  when 2 then 'SECONDA'
  4  else 'SUCCESSIVE' end posiz
  5  from fatture;

NUM_FATTURA DATA_FATT POSIZ
----------- --------- ----------
          1 01-OTT-10 PRIMA
          2 01-DIC-10 SECONDA
          3 20-OTT-10 SUCCESSIVE
          4 01-FEB-11 SUCCESSIVE
          5 01-DIC-10 SUCCESSIVE

WTO >select nome,
  2  case cod_fisc when null then 'NO CF'
  3  else 'CON CF' end "CF?"
  4  from clienti;

NOME     CF?
-------- ------
MARCO    CON CF
GIOVANNI CON CF
MATTEO   CON CF
LUCA     CON CF
AMBROGIO CON CF
GENNARO  CON CF
PASQUALE CON CF
VINCENZO CON CF

Selezionate 8 righe.

WTO >select num_fattura, importo,
  2  case sign(importo-500)
  3  when 1 then '>500'
  4  when 0 then '=500'
  5  when -1 then '<500' end fascia
  6  from fatture;

NUM_FATTURA    IMPORTO FASCIA
----------- ---------- ------
          1        300 <500
          2        500 =500
          3        700 >500
          4       1000 >500
          5        500 =500
```

Come si vede, tre query su quattro hanno dato lo stesso risultato una, invece, è errata. La struttura CASE utilizzata nella modalità appena mostrata,

non è in grado di gestire i valori nulli. O meglio li gestisce come UNKNOWN. Di conseguenza nella terza query la condizione WHEN NULL è sempre falsa.

Per ovviare a questo problema la query può essere riscritta utilizzando la seconda sintassi ammessa dalla struttura CASE.

```
WTO >select nome,
  2  case when (cod_fisc is null) then 'NO CF'
  3  else 'CON CF' end "CF?"
  4  from clienti;

NOME     CF?
-------- ------
MARCO    CON CF
GIOVANNI NO CF
MATTEO   CON CF
LUCA     CON CF
AMBROGIO NO CF
GENNARO  CON CF
PASQUALE CON CF
VINCENZO NO CF

Selezionate 8 righe.
```

Anche l'ultima query può essere scritta in maniera più leggibile utilizzando questa sintassi.

```
WTO >select num_fattura, importo,
  2  case when importo>500 then '>500'
  3       when importo=500 then '=500'
  4       when importo<500 then '<500'
  5  end fascia
  6  from fatture;

NUM_FATTURA    IMPORTO FASCIA
----------- ---------- ------
          1        300 <500
          2        500 =500
          3        700 >500
          4       1000 >500
          5        500 =500
```

7.5.12 GREATEST e LEAST

- GREATEST

La funzione GREATEST riceve in input un indefinito numero di parametri e ritorna il più grande di essi. Può essere applicata a numeri, stringhe e date ma ovviamente tutti i parametri devono essere dello stesso tipo, se non fossero dello stesso tipo Oracle cercherebbe di realizzare una conversione implicita al tipo del primo parametro.

```
WTO >select greatest(-3,5,1,7,2,-4)
  2  from dual;

GREATEST(-3,5,1,7,2,-4)
-----------------------
                      7

WTO >select greatest(sysdate, sysdate-2, sysdate+3, sysdate+1)
  2  from dual;
```

```
GREATEST(
---------
27-FEB-11

WTO >select greatest('MARCO','SANDRO','ALBERTO')
  2  from dual;

GREATE
------
SANDRO
```

- LEAST

La funzione LEAST è identica alla GREATEST con la differenza che ritorna il più piccolo dei parametri ricevuti.

```
WTO >select least(-3,5,1,7,2,-4)
  2  from dual;

LEAST(-3,5,1,7,2,-4)
--------------------
                  -4

WTO >select least(sysdate, sysdate-2, sysdate+3, sysdate+1)
  2  from dual;

LEAST(SYS
---------
22-FEB-11

WTO >select least('MARCO','SANDRO','ALBERTO')
  2  from dual;

LEAST('
-------
ALBERTO
```

7.6 JOIN: ricerche da più tabelle

Tutte le query presentate fino a questo punto attingevano ad una singola tabella per prelevare i dati. Ipotizziamo invece di voler estrarre, per ogni cliente, nome, cognome, indirizzo, nome del comune e provincia in cui risiede.

I dati nome e cognome sono nella tabella CLIENTI mentre i nomi dei comuni e le province sono nella tabella COMUNI. Sarà dunque necessario accedere contemporaneamente a due tabelle eseguendo una JOIN.

7.6.1 Sintassi per la JOIN

Esistono due sintassi distinte per l'esecuzione delle JOIN. Una delle due sintassi prevede di esplicitare, nella clausola FROM della query, non solo da quali tabelle bisogna estrarre i dati, ma anche quali condizioni utilizzare per abbinare correttamente tra loro i record estratti dalle diverse tabelle. Questa sintassi è stata assunta come standard ANSI ma è stata introdotta in Oracle solo nella versione 9.

La seconda sintassi, utilizzata da sempre in Oracle, prevede di indicare nella clausola FROM solo le tabelle da cui estrarre i dati, spostando nella clausola WHERE le condizioni da utilizzare per abbinare correttamente i record.

In questo manuale sarà solo fatto un accenno alla prima sintassi e sarà invece utilizzata sempre la seconda. Questa scelta deriva da tre considerazioni. La prima è che, per coerenza con il resto del manuale, si preferisce il dialetto SQL Oracle allo standard ANSI. La seconda ragione è che la sintassi ANSI risulta, quanto meno nella modesta esperienza dell'autore, più complessa da apprendere per i principianti. L'ultima ragione è che la tardiva introduzione della sintassi standard in Oracle ha fatto in modo che la maggior parte del codice SQL e PL/SQL oggi esistente su DB Oracle sia scritto con la sintassi non standard.

7.6.2 Prodotto cartesiano di tabelle

Come già accennato in precedenza, se si estraggono dati da due tabelle senza specificare alcuna condizione nella clausola WHERE per limitare tale estrazione, si ottiene un numero di righe pari al prodotto del numero di righe presenti nella prima tabella per il numero di righe presenti nella seconda tabella.

In tale risultato, detto "prodotto cartesiano di tabelle" ogni record estratto dalla prima tabella è accoppiato ad ogni record estratto dalla seconda.

A titolo di esempio estraiamo i campi NOME, COGNOME, INDIRIZZO e COMUNE dalla tabella CLIENTI insieme ai dati COD_COMUNE, DES_COMUNE e PROVINCIA della tabella COMUNI.

```
WTO >select nome, cognome, indirizzo, comune, cod_comune,
  2  des_comune, provincia
  3  from clienti, comuni;

NOME     COGNOME  INDIRIZZO              COMU COD_ DES_COMUNE PR
-------- -------- ---------------------- ---- ---- ---------- --
MARCO    ROSSI    VIA LAURENTINA, 700    H501 H501 ROMA       RM
MARCO    ROSSI    VIA LAURENTINA, 700    H501 G811 POMEZIA    RM
MARCO    ROSSI    VIA LAURENTINA, 700    H501 M297 FIUMICINO  RM
MARCO    ROSSI    VIA LAURENTINA, 700    H501 F205 MILANO     MI
MARCO    ROSSI    VIA LAURENTINA, 700    H501 G686 PIOLTELLO  MI
MARCO    ROSSI    VIA LAURENTINA, 700    H501 E415 LAINATE    MI
MARCO    ROSSI    VIA LAURENTINA, 700    H501 F839 NAPOLI     NA
MARCO    ROSSI    VIA LAURENTINA, 700    H501 G964 POZZUOLI   NA
MARCO    ROSSI    VIA LAURENTINA, 700    H501 G902 PORTICI    NA
GIOVANNI BIANCHI  VIA OSTIENSE, 850      H501 H501 ROMA       RM
GIOVANNI BIANCHI  VIA OSTIENSE, 850      H501 G811 POMEZIA    RM
GIOVANNI BIANCHI  VIA OSTIENSE, 850      H501 M297 FIUMICINO  RM
GIOVANNI BIANCHI  VIA OSTIENSE, 850      H501 F205 MILANO     MI
GIOVANNI BIANCHI  VIA OSTIENSE, 850      H501 G686 PIOLTELLO  MI
GIOVANNI BIANCHI  VIA OSTIENSE, 850      H501 E415 LAINATE    MI
GIOVANNI BIANCHI  VIA OSTIENSE, 850      H501 F839 NAPOLI     NA
GIOVANNI BIANCHI  VIA OSTIENSE, 850      H501 G964 POZZUOLI   NA
GIOVANNI BIANCHI  VIA OSTIENSE, 850      H501 G902 PORTICI    NA
MATTEO   VERDI    VIA DEL MARE, 8        G811 H501 ROMA       RM
MATTEO   VERDI    VIA DEL MARE, 8        G811 G811 POMEZIA    RM
```

```
MATTEO   VERDI    VIA DEL MARE, 8        G811 M297 FIUMICINO  RM
MATTEO   VERDI    VIA DEL MARE, 8        G811 F205 MILANO     MI
MATTEO   VERDI    VIA DEL MARE, 8        G811 G686 PIOLTELLO  MI
MATTEO   VERDI    VIA DEL MARE, 8        G811 E415 LAINATE    MI
MATTEO   VERDI    VIA DEL MARE, 8        G811 F839 NAPOLI     NA
MATTEO   VERDI    VIA DEL MARE, 8        G811 G964 POZZUOLI   NA
MATTEO   VERDI    VIA DEL MARE, 8        G811 G902 PORTICI    NA
LUCA     NERI     VIA TORINO, 30         F205 H501 ROMA       RM
LUCA     NERI     VIA TORINO, 30         F205 G811 POMEZIA    RM
LUCA     NERI     VIA TORINO, 30         F205 M297 FIUMICINO  RM
LUCA     NERI     VIA TORINO, 30         F205 F205 MILANO     MI
LUCA     NERI     VIA TORINO, 30         F205 G686 PIOLTELLO  MI
LUCA     NERI     VIA TORINO, 30         F205 E415 LAINATE    MI
LUCA     NERI     VIA TORINO, 30         F205 F839 NAPOLI     NA
LUCA     NERI     VIA TORINO, 30         F205 G964 POZZUOLI   NA
LUCA     NERI     VIA TORINO, 30         F205 G902 PORTICI    NA
AMBROGIO COLOMBO  PIAZZA DUOMO, 1        F205 H501 ROMA       RM
AMBROGIO COLOMBO  PIAZZA DUOMO, 1        F205 G811 POMEZIA    RM
AMBROGIO COLOMBO  PIAZZA DUOMO, 1        F205 M297 FIUMICINO  RM
AMBROGIO COLOMBO  PIAZZA DUOMO, 1        F205 F205 MILANO     MI
AMBROGIO COLOMBO  PIAZZA DUOMO, 1        F205 G686 PIOLTELLO  MI
AMBROGIO COLOMBO  PIAZZA DUOMO, 1        F205 E415 LAINATE    MI
AMBROGIO COLOMBO  PIAZZA DUOMO, 1        F205 F839 NAPOLI     NA
AMBROGIO COLOMBO  PIAZZA DUOMO, 1        F205 G964 POZZUOLI   NA
AMBROGIO COLOMBO  PIAZZA DUOMO, 1        F205 G902 PORTICI    NA
GENNARO  ESPOSITO VIA CARACCIOLO, 100    F839 H501 ROMA       RM
GENNARO  ESPOSITO VIA CARACCIOLO, 100    F839 G811 POMEZIA    RM
GENNARO  ESPOSITO VIA CARACCIOLO, 100    F839 M297 FIUMICINO  RM
GENNARO  ESPOSITO VIA CARACCIOLO, 100    F839 F205 MILANO     MI
GENNARO  ESPOSITO VIA CARACCIOLO, 100    F839 G686 PIOLTELLO  MI
GENNARO  ESPOSITO VIA CARACCIOLO, 100    F839 E415 LAINATE    MI
GENNARO  ESPOSITO VIA CARACCIOLO, 100    F839 F839 NAPOLI     NA
GENNARO  ESPOSITO VIA CARACCIOLO, 100    F839 G964 POZZUOLI   NA
GENNARO  ESPOSITO VIA CARACCIOLO, 100    F839 G902 PORTICI    NA
PASQUALE RUSSO    VIA GIULIO CESARE, 119 F839 H501 ROMA       RM
PASQUALE RUSSO    VIA GIULIO CESARE, 119 F839 G811 POMEZIA    RM
PASQUALE RUSSO    VIA GIULIO CESARE, 119 F839 M297 FIUMICINO  RM
PASQUALE RUSSO    VIA GIULIO CESARE, 119 F839 F205 MILANO     MI
PASQUALE RUSSO    VIA GIULIO CESARE, 119 F839 G686 PIOLTELLO  MI
PASQUALE RUSSO    VIA GIULIO CESARE, 119 F839 E415 LAINATE    MI
PASQUALE RUSSO    VIA GIULIO CESARE, 119 F839 F839 NAPOLI     NA
PASQUALE RUSSO    VIA GIULIO CESARE, 119 F839 G964 POZZUOLI   NA
PASQUALE RUSSO    VIA GIULIO CESARE, 119 F839 G902 PORTICI    NA
VINCENZO AMATO    VIA NAPOLI, 234        G964 H501 ROMA       RM
VINCENZO AMATO    VIA NAPOLI, 234        G964 G811 POMEZIA    RM
VINCENZO AMATO    VIA NAPOLI, 234        G964 M297 FIUMICINO  RM
VINCENZO AMATO    VIA NAPOLI, 234        G964 F205 MILANO     MI
VINCENZO AMATO    VIA NAPOLI, 234        G964 G686 PIOLTELLO  MI
VINCENZO AMATO    VIA NAPOLI, 234        G964 E415 LAINATE    MI
VINCENZO AMATO    VIA NAPOLI, 234        G964 F839 NAPOLI     NA
VINCENZO AMATO    VIA NAPOLI, 234        G964 G964 POZZUOLI   NA
VINCENZO AMATO    VIA NAPOLI, 234        G964 G902 PORTICI    NA

Selezionate 72 righe.
```

Come si vede ognuno degli otto clienti è stato associato ad ognuno dei 9 comuni, generando dunque un insieme di 72 record.

7.6.3 Inner join

Raramente l'obiettivo del programmatore è realizzare un prodotto cartesiano. Normalmente l'esigenza è abbinare tra loro i dati estratti dalle due tabelle secondo un criterio preciso.

Nell'esempio precedente è evidente che, per ogni cliente, non interessano i dati relativi a tutti i comuni, ma solo i dati del comune in cui quel cliente risiede. Alla query appena presentata, dunque, è necessario aggiungere una condizione di abbinamento, detta anche condizione di JOIN. In questo caso la condizione di JOIN è che il codice del comune presente in COMUNI (COD_COMUNE) sia lo stesso presente nella tabella CLIENTI (COMUNE). Si scriverà dunque:

```
WTO >select nome, cognome, indirizzo, comune, cod_comune,
  2  des_comune, provincia
  3  from clienti, comuni
  4  where comune=cod_comune;

NOME     COGNOME  INDIRIZZO              COMU COD_ DES_COMUNE PR
-------- -------- ---------------------- ---- ---- ---------- --
LUCA     NERI     VIA TORINO, 30         F205 F205 MILANO     MI
AMBROGIO COLOMBO  PIAZZA DUOMO, 1        F205 F205 MILANO     MI
PASQUALE RUSSO    VIA GIULIO CESARE, 119 F839 F839 NAPOLI     NA
GENNARO  ESPOSITO VIA CARACCIOLO, 100    F839 F839 NAPOLI     NA
MATTEO   VERDI    VIA DEL MARE, 8        G811 G811 POMEZIA    RM
VINCENZO AMATO    VIA NAPOLI, 234        G964 G964 POZZUOLI   NA
GIOVANNI BIANCHI  VIA OSTIENSE, 850      H501 H501 ROMA       RM
MARCO    ROSSI    VIA LAURENTINA, 700    H501 H501 ROMA       RM

Selezionate 8 righe.
```

Il risultato è molto più significativo del precedente presentando in effetti una riga per ogni cliente con associati i dati del comune relativo.

Giusto come esempio, la stessa query nella sintassi standard ISO sarebbe stata.

```
WTO >select nome, cognome, indirizzo, comune, cod_comune,
  2  des_comune, provincia
  3  from clienti inner join comuni on comune=cod_comune;

NOME     COGNOME  INDIRIZZO              COMU COD_ DES_COMUNE PR
-------- -------- ---------------------- ---- ---- ---------- --
LUCA     NERI     VIA TORINO, 30         F205 F205 MILANO     MI
AMBROGIO COLOMBO  PIAZZA DUOMO, 1        F205 F205 MILANO     MI
PASQUALE RUSSO    VIA GIULIO CESARE, 119 F839 F839 NAPOLI     NA
GENNARO  ESPOSITO VIA CARACCIOLO, 100    F839 F839 NAPOLI     NA
MATTEO   VERDI    VIA DEL MARE, 8        G811 G811 POMEZIA    RM
VINCENZO AMATO    VIA NAPOLI, 234        G964 G964 POZZUOLI   NA
GIOVANNI BIANCHI  VIA OSTIENSE, 850      H501 H501 ROMA       RM
MARCO    ROSSI    VIA LAURENTINA, 700    H501 H501 ROMA       RM

Selezionate 8 righe.
```

La complessità di questa sintassi aumenta significativamente quando aumenta il numero delle tabelle in JOIN.

La caratteristica essenziale di una INNER JOIN è che i record presenti in una tabella che non hanno corrispondenza nell'altra tabella vengono

ignorati. Nell'esempio precedente Tutti i comuni in cui non risiede alcun cliente sono stati ignorati. Allo stesso modo se un cliente non avesse in tabella il riferimento al comune di residenza sarebbe escluso dalla query. Ciò è impedito, nel nostro database d'esempio, dal fatto che il codice comune è obbligatorio nella tabella CLIENTI.

Ovviamente l'esistenza di una JOIN non esclude la possibilità di utilizzare nella clausola WHERE altre condizioni per filtrare i dati.

La query precedente, ad esempio, può essere ristretta ai soli clienti il cui nome comincia per 'M'

```
WTO >select nome, cognome, indirizzo, comune, cod_comune,
  2  des_comune, provincia
  3  from clienti, comuni
  4  where comune=cod_comune
  5  and nome like 'M%';

NOME     COGNOME  INDIRIZZO              COMU COD_ DES_COMUNE PR
-------- -------- ---------------------- ---- ---- ---------- --
MARCO    ROSSI    VIA LAURENTINA, 700    H501 H501 ROMA       RM
MATTEO   VERDI    VIA DEL MARE, 8        G811 G811 POMEZIA    RM
```

Una JOIN può interessare anche più tabelle. Ovviamente sarà necessario specificare un numero di condizioni di JOIN sufficiente ad abbinare ogni tabella con le altre o direttamente, o attraverso altre condizioni di JOIN.

Si vogliano ad esempio estrarre, per ogni cliente, gli ordini e le fatture. I campi da estrarre sono NOME e COGNOME da CLIENTI; NUM_ORDINE, DATA_ORDINE ed IMPORTO da ORDINI; NUM_FATTURA, DATA_FATTURA ed IMPORTO da FATTURE.

Le condizioni di JOIN: CLIENTI è legata ad ORDINI mediante il campo COD_CLIENTE presente in entrambe le tabelle; ORDINI è legata a FATTURE mediante il campo NUM_ORDINE presente in entrambe le tabelle; CLIENTI E FATTURE non sono legate direttamente ma sono legate indirettamente mediante i due legami precedenti.

Un problema da risolvere è quello dei campi omonimi. Sia tra i campi da estrarre che nelle condizioni di JOIN ci sono campi che hanno lo stesso nome presenti in diverse tabelle. Per evitare che queste omonimie generino errori è necessario qualificare ciascun campo con il nome oppure con un alias della tabella a cui appartiene.

La query sarà dunque scritta come segue:

```
WTO >select c.nome, c.cognome, o.num_ordine, o.data_ordine,
  2  o.importo, f.num_fattura, f.importo
  3  from clienti c, ordini o, fatture f
  4  where c.cod_cliente=o.cod_cliente
  5  and f.num_ordine=o.num_ordine;

NOME     COGNOME  NUM_ORDINE DATA_ORDI  IMPORTO NUM_FATTURA  IMPORTO
-------- -------- ---------- --------- -------- ----------- --------
MARCO    ROSSI             1 01-OTT-10      800           2      500
MARCO    ROSSI             1 01-OTT-10      800           1      300
GIOVANNI BIANCHI           2 20-OTT-10      700           3      700
LUCA     NERI              3 01-NOV-10     1000           4     1000
PASQUALE RUSSO             5 01-DIC-10     1700           5      500
```

Ovviamente se un cliente ha più di un ordine oppure se un ordine ha più fatture I record di clienti o ordini sono moltiplicati. Viceversa, come già osservato, i clienti senza ordini o con ordini ma senza fatture non compaiono affatto nella lista.

7.6.4 Outer join

Come già detto la INNER JOIN causa la perdita dei record di una tabella quando questi non hanno corrispondenza nell'altra tabella.

Un ulteriore esempio può essere la selezione congiunta di ordini e fatture.

```
WTO >select * from ordini;

NUM_ORDINE DATA_ORDI COD  IMPORTO NOTE
---------- --------- --- -------- ------------------------------
         1 01-OTT-10   1      800 Sconto 60 euro per promozione.
         2 20-OTT-10   2      700
         3 01-NOV-10   4     1000 Fattura differita 90gg
         4 01-NOV-10   5     1200
         5 01-DIC-10   7     1700

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT  IMPORTO P
----------- ---------- --------- -------- -
          1          1 01-OTT-10      300 S
          2          1 01-DIC-10      500 N
          3          2 20-OTT-10      700 S
          4          3 01-FEB-11     1000 N
          5          5 01-DIC-10      500 S

WTO >select o.num_ordine, f.num_fattura
  2  from ordini o, fatture f
  3  where o.num_ordine=f.num_ordine;

NUM_ORDINE NUM_FATTURA
---------- -----------
         1           1
         1           2
         2           3
         3           4
         5           5
```

Ci sono cinque ordini e cinque fatture, l'ordine numero uno ha due fatture mentre l'ordine numero quattro non ne ha nessuna.

Nel risultato della INNER JOIN, dunque, l'ordine numero uno compare due volte mentre l'ordine numero quattro è assente.

Potrebbe succedere di avere necessità di estrarre tutti gli ordini, anche qualora non ci fossero fatture collegate.

Per realizzare questo nuovo tipo di JOIN, detto OUTER JOIN, è sufficiente una piccola modifica alla condizione di JOIN. In particolare, nella condizione di JOIN, bisogna aggiungere i tre caratteri '(+)' dopo la colonna in cui potrebbero mancare alcune corrispondenze.

Nel nostro esempio le colonne coinvolte nella condizione di JOIN sono il numero ordine di ORDINI (O.NUM_ORDINE) ed il numero ordine di fatture (F.NUM_ORDINE). Abbiamo detto che vogliamo tutti gli ordini anche quelli che non hanno fatture corrispondenti. Quindi la colonna in cui vogliamo ignorare le mancanze di corrispondenze è F.NUM_ORDINE. La condizione di JOIN diventa dunque

```
O.NUM_ORDINE=F.NUM_ORDINE(+)
```

```
WTO >select o.num_ordine, f.num_fattura
  2  from ordini o, fatture f
  3  where o.num_ordine=f.num_ordine(+);

NUM_ORDINE NUM_FATTURA
---------- -----------
         1           1
         1           2
         2           3
         3           4
         5           5
         4

Selezionate 6 righe.
```

In effetti è come se la presenza dei caratteri '(+)' vicino alla colonna F.NUM_ORDINE assumesse il seguente significato: si aggiungano durante la query dei valori fittizi nella colonna F.NUM_ORDINE in modo che tutti i record degli ordini trovino una corrispondenza.

A differenza della INNER JOIN, la OUTER JOIN ha un verso. Nella query precedente si poteva ribaltare la prospettiva chiedendo tutte le fatture, anche quelle eventualmente senza ordine. Nel nostro caso specifico tali record non ci possono essere grazie all'esistenza della FOREIGN KEY obbligatoria tra FATTURE ed ORDINI, la query comunque si scrive così:

```
WTO >select o.num_ordine, f.num_fattura
  2  from ordini o, fatture f
  3  where o.num_ordine(+)=f.num_ordine;

NUM_ORDINE NUM_FATTURA
---------- -----------
         1           1
         1           2
         2           3
         3           4
         5           5
```

Ed è perfettamente, nel nostro caso, identica alla INNER JOIN.

Questa caratteristica può essere meglio mostrata utilizzando le tabelle EMP e DEPT dello schema SCOTT.

```
WTO> desc emp
 Name                 Null?    Type
 ------------------ --------- ------------
 EMPNO              NOT NULL NUMBER(4)
 ENAME                        VARCHAR2(10)
 JOB                          VARCHAR2(9)
 MGR                          NUMBER(4)
 HIREDATE                     DATE
 SAL                          NUMBER(7,2)
 COMM                         NUMBER(7,2)
 DEPTNO                       NUMBER(2)

WTO> desc dept
 Name                 Null?    Type
 ------------------ --------- ------------
 DEPTNO             NOT NULL NUMBER(2)
 DNAME                        VARCHAR2(14)
 LOC                          VARCHAR2(13)

WTO> select ename, deptno from emp;

ENAME          DEPTNO
---------- ----------
SMITH              20
ALLEN              30
WARD               30
JONES              20
MARTIN             30
BLAKE              30
CLARK              10
SCOTT              20
KING               10
TURNER             30
ADAMS              20
JAMES              30
FORD               20
MILLER             10

14 rows selected.

WTO> select deptno, dname from dept;

    DEPTNO DNAME
---------- --------------
        10 ACCOUNTING
        20 RESEARCH
        30 SALES
        40 OPERATIONS
```

Le due tabelle sono legate dal campo DEPTNO comune ad entrambe. La INNER JOIN si scrive dunque.

```
WTO> select ename, e.deptno, d.deptno, dname
  2  from emp e, dept d
  3  where e.deptno=d.deptno;

ENAME          DEPTNO     DEPTNO DNAME
---------- ---------- ---------- --------------
CLARK              10         10 ACCOUNTING
KING               10         10 ACCOUNTING
MILLER             10         10 ACCOUNTING
JONES              20         20 RESEARCH
```

```
FORD                 20          20 RESEARCH
ADAMS                20          20 RESEARCH
SMITH                20          20 RESEARCH
SCOTT                20          20 RESEARCH
WARD                 30          30 SALES
TURNER               30          30 SALES
ALLEN                30          30 SALES
JAMES                30          30 SALES
BLAKE                30          30 SALES
MARTIN               30          30 SALES

14 rows selected.
```

Se adesso apportiamo una piccola modifica alla tabella EMP

```
WTO> update emp set deptno=null where ename='FORD';

1 row updated.
```

La INNER JOIN fornisce un risultato differente.

```
WTO> select ename, e.deptno, d.deptno, dname
  2  from emp e, dept d
  3  where e.deptno=d.deptno;

ENAME          DEPTNO     DEPTNO DNAME
---------- ---------- ---------- --------------
CLARK              10         10 ACCOUNTING
KING               10         10 ACCOUNTING
MILLER             10         10 ACCOUNTING
JONES              20         20 RESEARCH
SCOTT              20         20 RESEARCH
ADAMS              20         20 RESEARCH
SMITH              20         20 RESEARCH
ALLEN              30         30 SALES
WARD               30         30 SALES
TURNER             30         30 SALES
JAMES              30         30 SALES
BLAKE              30         30 SALES
MARTIN             30         30 SALES

13 rows selected.
```

Mancano sia il dipendente FORD, che non ha un dipartimento corrispondente, sia il dipartimento 40, che non ha dipendenti corrispondenti.

Per ottenere tutti i dipendenti, anche quelli senza dipartimento, utilizzeremo la seguente OUTER JOIN.

```
WTO> select ename, e.deptno, d.deptno, dname
  2  from emp e, dept d
  3  where e.deptno=d.deptno(+);

ENAME          DEPTNO     DEPTNO DNAME
---------- ---------- ---------- --------------
MILLER             10         10 ACCOUNTING
KING               10         10 ACCOUNTING
CLARK              10         10 ACCOUNTING
ADAMS              20         20 RESEARCH
SCOTT              20         20 RESEARCH
JONES              20         20 RESEARCH
SMITH              20         20 RESEARCH
JAMES              30         30 SALES
```

```
TURNER              30         30 SALES
BLAKE               30         30 SALES
MARTIN              30         30 SALES
WARD                30         30 SALES
ALLEN               30         30 SALES
FORD

14 rows selected.
```

Mentre per estrarre tutti i dipartimenti, anche senza dipendenti, dovremo usare la OUTER JOIN opposta.

```
WTO> select ename, e.deptno, d.deptno, dname
  2  from emp e, dept d
  3  where e.deptno(+)=d.deptno;

ENAME          DEPTNO     DEPTNO DNAME
---------- ---------- ---------- --------------
CLARK              10         10 ACCOUNTING
KING               10         10 ACCOUNTING
MILLER             10         10 ACCOUNTING
JONES              20         20 RESEARCH
SCOTT              20         20 RESEARCH
ADAMS              20         20 RESEARCH
SMITH              20         20 RESEARCH
ALLEN              30         30 SALES
WARD               30         30 SALES
TURNER             30         30 SALES
JAMES              30         30 SALES
BLAKE              30         30 SALES
MARTIN             30         30 SALES
                              40 OPERATIONS

14 rows selected.
```

Queste due OUTER JOIN vengono dette anche DESTRA e SINISTRA, ma ciò non è molto importante.

È invece significativo notare che, fino ad Oracle 11g, le OUTER JOIN appena viste non posso estrarre tutti i record di entrambe le tabelle. Non è insomma possibile scrivere la sequenza '(+)' da entrambi i dati della condizione di JOIN.

```
WTO> select ename, e.deptno, d.deptno, dname
  2  from emp e, dept d
  3  where e.deptno(+)=d.deptno(+);
where e.deptno(+)=d.deptno(+)
                 *
ERROR at line 3:
ORA-01468: a predicate may reference only one outer-joined table
```

L'estrazione completa di entrambe le tabelle ha comunque senso e prende il nome di FULL OUTER JOIN.

Come visto questo tipo di JOIN non può essere realizzato mediante la sintassi Oracle. È invece possibile realizzarla usando la sintassi standard ANSI.

```
WTO> select ename, e.deptno, d.deptno, dname
  2  from emp e full outer join dept d on e.deptno=d.deptno;

ENAME          DEPTNO     DEPTNO DNAME
```

```
---------- ---------- ---------- --------------
MILLER             10         10 ACCOUNTING
KING               10         10 ACCOUNTING
CLARK              10         10 ACCOUNTING
ADAMS              20         20 RESEARCH
SCOTT              20         20 RESEARCH
JONES              20         20 RESEARCH
SMITH              20         20 RESEARCH
JAMES              30         30 SALES
TURNER             30         30 SALES
BLAKE              30         30 SALES
MARTIN             30         30 SALES
WARD               30         30 SALES
ALLEN              30         30 SALES
FORD
                              40 OPERATIONS

15 rows selected.
```

Ovviamente, visto che ci sono sempre molte strade per giungere alla medesima destinazione, c'è modo per ottenere lo stesso risultato in Oracle senza utilizzare la FULL OUTER JOIN. Ce ne occuperemo nel paragrafo successivo.

Nelle precedenti versioni di Oracle non era possibile mettere in outer join più tabelle con la stessa utilizzando la sintassi nativa. Era necessario utilizzare la sintassi standard ANSI.

Questo limite è stato superato in Oracle 12c. Per comprendere bene la questione dobbiamo fare un esempio. Ipotizziamo di avere tre tabelle: CLIENTI, PIANI e COMUNI create e popolate come segue:

```
create table clienti (
codice  number       not null,
nome    varchar2(20) not null,
piano   varchar2(3)  not null,
comune  varchar2(4)  not null);

create table piani (
codice varchar2(3) not null,
descr  varchar2(20));

create table comuni (
codice varchar2(4) not null,
nome   varchar2(20));

insert into piani values ('P1G','1Gb mese flat');
insert into piani values ('P2G','2Gb mese flat');
insert into piani values ('P4G','4Gb mese flat');
insert into piani values ('P8G','8Gb mese flat');
insert into comuni values ('H501','ROMA');
insert into comuni values ('F839','NAPOLI');
insert into comuni values ('F205','MILANO');
insert into comuni values ('L219','TORINO');
insert into clienti values (1, 'Gianni Bianchi', 'P2G', 'L219');
insert into clienti values (2, 'Stefano Neri', 'P8G', 'L219');
insert into clienti values (3, 'Luca Rossi', 'P2G', 'H501');
insert into clienti values (4, 'Davide Verdi', 'P8G', 'L219');
```

```
insert into clienti values (5, 'Marco Gialli', 'P2G', 'F839');
insert into clienti values (6, 'Vittorio Grigi', 'P1G', 'H501');
```

Di conseguenza il contenuto delle tabelle sarà il seguente:

```
SQL >select * from clienti;

    CODICE NOME                 PIA COMU
---------- -------------------- --- ----
         1 Gianni Bianchi       P2G L219
         2 Stefano Neri         P8G L219
         3 Luca Rossi           P2G H501
         4 Davide Verdi         P8G L219
         5 Marco Gialli         P2G F839
         6 Vittorio Grigi       P1G H501

6 rows selected.

SQL >select * from piani;

COD DESCR
--- --------------------
P1G 1Gb mese flat
P2G 2Gb mese flat
P4G 4Gb mese flat
P8G 8Gb mese flat

SQL >select * from comuni;

CODI NOME
---- --------------------
H501 ROMA
F839 NAPOLI
F205 MILANO
L219 TORINO
```

Se leggiamo i dati mettendo le tre tabelle in join:

```
SQL >select c.nome cliente, descr piano, co.nome comune
  2  from clienti c, piani p, comuni co
  3  where c.piano=p.codice
  4  and c.comune=co.codice;

CLIENTE              PIANO                COMUNE
-------------------- -------------------- ------------------
Vittorio Grigi       1Gb mese flat        ROMA
Marco Gialli         2Gb mese flat        NAPOLI
Gianni Bianchi       2Gb mese flat        TORINO
Luca Rossi           2Gb mese flat        ROMA
Davide Verdi         8Gb mese flat        TORINO
Stefano Neri         8Gb mese flat        TORINO

6 rows selected.
```

Se ci chiediamo quanti clienti hanno attivato ogni piano, volendo visualizzare tutti i piani disponibili, dobbiamo utilizzare una outer join:

```
SQL >select descr piano, count(c.nome) clienti
  2  from clienti c, piani p
  3  where c.piano(+)=p.codice
  4  group by descr;

PIANO                   CLIENTI
```

```
-------------------- ----------
1Gb mese flat                 1
2Gb mese flat                 3
4Gb mese flat                 0
8Gb mese flat                 2
```

Lo stesso dicasi se vogliamo sapere quanti clienti abbiamo per ogni comune, visualizzando tutti i comuni:

```
SQL >select co.nome comune, count(c.nome) clienti
  2  from clienti c, comuni co
  3  where c.comune(+)=co.codice
  4  group by co.nome;

COMUNE                  CLIENTI
-------------------- ----------
MILANO                        0
NAPOLI                        1
ROMA                          2
TORINO                        3
```

Se, infine, vogliamo sapere quanti clienti abbiamo per ogni combinazione di comune e piano abbiamo bisogno di una doppia outer join:

```
SQL >select descr piano, co.nome comune, count(c.nome) clienti
  2  from clienti c, piani p, comuni co
  3  where c.piano(+)=p.codice
  4  and c.comune(+)=co.codice
  5  group by co.nome, descr
  6  order by 1, 2;
where c.piano(+)=p.codice
                *
ERROR at line 3:
ORA-01417: a table may be outer joined to at most one other table
```

Che in Oracle 11g non è ammessa.

Nelle vecchie versioni del db era necessario riscrivere la query nella sintassi ANSI:

```
SQL >select descr piano, co.nome comune, count(c.nome) clienti
  2  from comuni co cross join piani p
  3  left outer join clienti c
  4  on c.piano=p.codice
  5  and c.comune=co.codice
  6  group by descr, co.nome
  7  order by 1, 2;

PIANO                COMUNE                  CLIENTI
-------------------- -------------------- ----------
1Gb mese flat        MILANO                        0
1Gb mese flat        NAPOLI                        0
1Gb mese flat        ROMA                          1
1Gb mese flat        TORINO                        0
2Gb mese flat        MILANO                        0
2Gb mese flat        NAPOLI                        1
2Gb mese flat        ROMA                          1
2Gb mese flat        TORINO                        1
4Gb mese flat        MILANO                        0
4Gb mese flat        NAPOLI                        0
```

```
4Gb mese flat        ROMA                          0
4Gb mese flat        TORINO                        0
8Gb mese flat        MILANO                        0
8Gb mese flat        NAPOLI                        0
8Gb mese flat        ROMA                          0
8Gb mese flat        TORINO                        2

16 rows selected.
```

Oppure, per utilizzare la sintassi nativa, usare una view inline:

```
SQL >select x.descr piano, x.nome , count(c.nome) clienticomune
  2  from (select descr, nome, co.codice comune, p.codice piano
  3          from comuni co, piani p) x,
  4       clienti c
  5  where x.comune=c.comune(+)
  6  and x.piano=c.piano(+)
  7  group by x.descr, x.nome;

PIANO                NOME                 CLIENTICOMUNE
-------------------- -------------------- -------------
1Gb mese flat        ROMA                             1
1Gb mese flat        MILANO                           0
1Gb mese flat        NAPOLI                           0
1Gb mese flat        TORINO                           0
2Gb mese flat        ROMA                             1
2Gb mese flat        MILANO                           0
2Gb mese flat        NAPOLI                           1
2Gb mese flat        TORINO                           1
4Gb mese flat        ROMA                             0
4Gb mese flat        MILANO                           0
4Gb mese flat        NAPOLI                           0
4Gb mese flat        TORINO                           0
8Gb mese flat        ROMA                             0
8Gb mese flat        MILANO                           0
8Gb mese flat        NAPOLI                           0
8Gb mese flat        TORINO                           2

16 rows selected.
```

In Oracle 12c la doppia outer join può essere eseguita direttamente nella sintassi nativa:

```
O12c>select descr piano, co.nome comune, count(c.nome) clienti
  2  from clienti c, piani p, comuni co
  3  where c.piano(+)=p.codice
  4  and c.comune(+)=co.codice
  5  group by co.nome, descr
  6  order by 1, 2;

PIANO                COMUNE                  CLIENTI
-------------------- -------------------- ----------
1Gb mese flat        MILANO                        0
1Gb mese flat        NAPOLI                        0
1Gb mese flat        ROMA                          1
1Gb mese flat        TORINO                        0
2Gb mese flat        MILANO                        0
2Gb mese flat        NAPOLI                        1
2Gb mese flat        ROMA                          1
```

```
2Gb mese flat        TORINO                          1
4Gb mese flat        MILANO                          0
4Gb mese flat        NAPOLI                          0
4Gb mese flat        ROMA                            0
4Gb mese flat        TORINO                          0
8Gb mese flat        MILANO                          0
8Gb mese flat        NAPOLI                          0
8Gb mese flat        ROMA                            0
8Gb mese flat        TORINO                          2

16 righe selezionate.
```

Questa maggiore flessibilità della sintassi nativa facilita la vita agli sviluppatori, ma non solo. L'ottimizzatore ha un'opzione in più quando riscrive le query per determinare il miglior percorso d'esecuzione quindi, in alcuni casi, si possono anche ottenere performance migliori.

7.6.5 Le clausole APPLY e LATERAL

Per comprendere quest'argomento è necessaria una conoscenza del PL/SQL, questo linguaggio sarà trattato nel capitolo successivo.

In Oracle 12c è stata introdotta la clausola APPLY che consente di eseguire una join tra una tabella ed una query che faccia riferimento alla tabella stessa. Cerchiamo di chiarire con un esempio:

In Oracle 11g questa query va in errore perché la select che si trova nella FROM fa riferimento ad un'altra tabella che si trova nella stessa FROM.

```
O11g>Select p.descr, v.nome
  2  from piani p,
  3       (select * from clienti c where c.piano=p.codice) v;
     (select * from clienti c where c.piano=p.codice) v
                                            *
ERRORE alla riga 3:
ORA-00904: "P"."CODICE": identificativo non valido
```

In pratica, nella stessa FROM la seconda tabella (una select) è correlata alla prima e ciò non è ammesso.

Anche in Oracle 12c la stessa istruzione va in errore:

```
O12c>Select p.descr, v.nome
  2  from piani p, (select * from clienti c where c.piano=p.codice) v;
from piani p, (select * from clienti c where c.piano=p.codice) v
                                                   *
ERRORE alla riga 2:
ORA-00904: "P"."CODICE": identificativo non valido
```

Possiamo però comunque eseguirla sostituendo la join con la clausola CROSS APPLY

```
O12c>Select p.descr, v.nome
  2  from piani p CROSS APPLY
  3  (select * from clienti c where c.piano=p.codice) v;
```

```
DESCR                NOME
-------------------- --------------------
2Gb mese flat        Gianni Bianchi
8Gb mese flat        Stefano Neri
2Gb mese flat        Luca Rossi
8Gb mese flat        Davide Verdi
2Gb mese flat        Marco Gialli
1Gb mese flat        Vittorio Grigi

6 righe selezionate.
```

Se ci serve un outer join, possiamo utilizzare la OUTER APPLY

```
O12c>Select p.descr, v.nome
  2  from piani p OUTER APPLY
  3  (select * from clienti c where c.piano=p.codice) v;

DESCR                NOME
-------------------- --------------------
2Gb mese flat        Gianni Bianchi
8Gb mese flat        Stefano Neri
2Gb mese flat        Luca Rossi
8Gb mese flat        Davide Verdi
2Gb mese flat        Marco Gialli
1Gb mese flat        Vittorio Grigi
4Gb mese flat

7 righe selezionate.
```

Nello standard ANSI non è prevista la clausola APPLY, viceversa è prevista la clausola LATERAL che racchiude una subquery correlata ad un'altra tabella all'interno della FROM.

```
O12c>Select p.descr, v.nome
  2  from piani p,
  3  LATERAL(select * from clienti c where c.piano=p.codice) v;

DESCR                NOME
-------------------- --------------------
2Gb mese flat        Gianni Bianchi
8Gb mese flat        Stefano Neri
2Gb mese flat        Luca Rossi
8Gb mese flat        Davide Verdi
2Gb mese flat        Marco Gialli
1Gb mese flat        Vittorio Grigi

6 righe selezionate.
```

La clausola LATERAL è supportata a partire da Oracle 12c ed in questo scenario è equivalente alla CROSS APPLY.

La differenza è che la LATERAL può essere utilizzata solo con una subquery mentre la CROSS APPLY anche con altro, una table function, cioè una funzione PL/SQL che restituisce una tabella di dati come risultato, ad esempio.

Creiamo un tipo che ricalca il record della tabella CLIENTI:

```
O12c>create or replace type
  2  t_cliente is object (
  3  nome varchar2(30),
  4  piano varchar2(3),
```

```
  5  comune varchar2(4));
  6  /

Tipo creato.
```

E poi un tipo che ricalca l'intera tabella CLIENTI:

```
O12c>create or replace type
  2  t_clienti is table of t_cliente;
  3  /
Tipo creato.
```

Definiamo una TABLE FUNCTION che, ricevuto in input un codice di piano, restituisca i clienti che lo hanno sottoscritto:

```
O12c>create or replace function tab_cli(p_piano in varchar2)
  2  return t_clienti is
  3  cli t_clienti;
  4  begin
  5    select t_cliente(nome,piano,comune)
  6      bulk collect into cli
  7      from clienti
  8     where piano=p_piano;
  9
 10    return cli;
 11  end;
 12  /

Funzione creata.

O12c>select * from table(tab_cli('P2G'));

NOME                           PIA COMU
------------------------------ --- ----
Gianni Bianchi                 P2G L219
Luca Rossi                     P2G H501
Marco Gialli                   P2G F839
```

A questo punto, la CROSS APPLY si può utilizzare pure su questa TABLE FUNCTION:

```
O12c>Select p.descr, v.nome
  2  from piani p CROSS APPLY table(tab_cli(p.codice)) v;

DESCR                NOME
-------------------- ------------------------------
1Gb mese flat        Vittorio Grigi
2Gb mese flat        Gianni Bianchi
2Gb mese flat        Luca Rossi
2Gb mese flat        Marco Gialli
8Gb mese flat        Stefano Neri
8Gb mese flat        Davide Verdi

6 righe selezionate.
```

Mentre la LATERAL vuole per forza una SELECT:

```
O12c>Select p.descr, v.nome
  2  from piani p, LATERAL(table(tab_cli(p.codice))) v;
from piani p, LATERAL(table(tab_cli(p.codice))) v
```

```
                       *
ERRORE alla riga 2:
ORA-00928: parola chiave SELECT mancante
```

7.7 Gli operatori insiemistici

Abbiamo finora associato molte volte il risultato di una query ad un insieme di record. I principali operatori insiemistici sono implementati anche in SQL.

7.7.1 UNION

L'operatore UNION consente di specificare due query distinte ed ottenere come risultato l'unione insiemistica dei risultati. Le due query devono ovviamente estrarre lo stesso numero di colonne e le colonne che si trovano nella stessa posizione nelle diverse query devono essere dello stesso tipo di dato.

```
WTO >select nome, cognome from clienti
  2  where cod_fisc is null
  3  union
  4  select nome, cognome from clienti
  5  where comune='H501';

NOME                           COGNOME
------------------------------ ------------------------------
AMBROGIO                       COLOMBO
GIOVANNI                       BIANCHI
MARCO                          ROSSI
VINCENZO                       AMATO
```

Questa query è corretta perché entrambe le query che la compongono estraggono due stringhe. Il risultato di questa UNION è equivalente a quello ottenuto con la query

```
WTO >select nome, cognome from clienti
  2  where cod_fisc is null or comune='H501';

NOME                           COGNOME
------------------------------ --------------------
MARCO                          ROSSI
GIOVANNI                       BIANCHI
AMBROGIO                       COLOMBO
VINCENZO                       AMATO
```

Le seguenti due query invece sono errate.

```
WTO >select nome, cognome, indirizzo from clienti
  2  where cod_fisc is null
  3  union
  4  select nome, cognome from clienti
  5  where comune='H501';
select nome, cognome, indirizzo from clienti
*
ERRORE alla riga 1:
ORA-01789: query block has incorrect number of result columns
```

```
WTO >select nome, cognome from clienti
  2  where cod_fisc is null
  3  union
  4  select nome, cod_cliente from clienti
  5  where comune='H501';
select nome, cognome from clienti
             *
ERRORE alla riga 1:
ORA-01790: expression must have same datatype as corresponding
expression
```

La prima è errata perché le due query che la compongono estraggono un numero differente di colonne, la seconda è errata perché, la seconda colonna nella prima query è una stringa mentre nella seconda query è un numero.

La query successiva è formalmente corretta sebbene le colonne estratte contengono dati che non ha molto senso mescolare insieme.

```
WTO >select nome, cognome from clienti
  2  where cod_fisc is null
  3  union
  4  select indirizzo, comune
  5  from clienti
  6  where comune='H501';

NOME                           COGNOME
------------------------------ ------------------------------
AMBROGIO                       COLOMBO
GIOVANNI                       BIANCHI
VIA LAURENTINA, 700            H501
VIA OSTIENSE, 850              H501
VINCENZO                       AMATO
```

È possibile inserire una sola clausola di ordinamento alla fine della UNION, cioè in coda alla seconda query.

```
WTO >select nome, cognome from clienti
  2  where cod_fisc is null
  3  union
  4  select indirizzo, comune
  5  from clienti
  6  where comune='H501'
  7  order by nome;

NOME                           COGNOME
------------------------------ ---------------------
AMBROGIO                       COLOMBO
GIOVANNI                       BIANCHI
VIA LAURENTINA, 700            H501
VIA OSTIENSE, 850              H501
VINCENZO                       AMATO

WTO >select nome, cognome from clienti
  2  where cod_fisc is null
  3  union
  4  select indirizzo, comune
  5  from clienti
  6  where comune='H501'
  7  order by 1;

NOME                           COGNOME
```

```
------------------------------ ---------------------
AMBROGIO                       COLOMBO
GIOVANNI                       BIANCHI
VIA LAURENTINA, 700            H501
VIA OSTIENSE, 850              H501
VINCENZO                       AMATO
```

L'operatore UNION elimina i record duplicati.

```
WTO >select nome from clienti
  2  union
  3  select nome from clienti;

NOME
------------------------------
AMBROGIO
GENNARO
GIOVANNI
LUCA
MARCO
MATTEO
PASQUALE
VINCENZO
```

Tornando all'esempio della FULL OUTER JOIN visto nel paragrafo precedente, il medesimo risultato può essere ottenuto con una UNION come segue:

```
WTO> select ename, e.deptno, d.deptno, dname
  2  from emp e, dept d
  3  where e.deptno=d.deptno(+)
  4  union
  5  select ename, e.deptno, d.deptno, dname
  6  from emp e, dept d
  7  where e.deptno(+)=d.deptno;

ENAME          DEPTNO     DEPTNO DNAME
---------- ---------- ---------- --------------
ADAMS              20         20 RESEARCH
ALLEN              30         30 SALES
BLAKE              30         30 SALES
CLARK              10         10 ACCOUNTING
FORD
JAMES              30         30 SALES
JONES              20         20 RESEARCH
KING               10         10 ACCOUNTING
MARTIN             30         30 SALES
MILLER             10         10 ACCOUNTING
SCOTT              20         20 RESEARCH
SMITH              20         20 RESEARCH
TURNER             30         30 SALES
WARD               30         30 SALES
                              40 OPERATIONS

15 rows selected.
```

In pratica si uniscono i risultati delle due OUTER JOIN destra e sinistra.

7.7.2 UNION ALL

L'operatore UNION ALL si comporta come la UNION ma non scarta i record duplicati.

```
WTO >select nome from clienti
  2  union all
  3  select nome from clienti;

NOME
------------------------------
MARCO
GIOVANNI
MATTEO
LUCA
AMBROGIO
GENNARO
PASQUALE
VINCENZO
MARCO
GIOVANNI
MATTEO
LUCA
AMBROGIO
GENNARO
PASQUALE
VINCENZO

Selezionate 16 righe.
```

Utilizzando la UNION ALL è possibile ottenere nuovamente il risultato della FULL OUTER JOIN di sopra, ma con una query leggermente più performante.

```
WTO> select ename, e.deptno, d.deptno, dname
  2  from emp e, dept d
  3  where e.deptno=d.deptno(+)
  4  union all
  5  select ename, e.deptno, d.deptno, dname
  6  from emp e, dept d
  7  where e.deptno(+)=d.deptno and e.deptno is null;

ENAME          DEPTNO     DEPTNO DNAME
---------- ---------- ---------- --------------
MILLER             10         10 ACCOUNTING
KING               10         10 ACCOUNTING
CLARK              10         10 ACCOUNTING
ADAMS              20         20 RESEARCH
SCOTT              20         20 RESEARCH
JONES              20         20 RESEARCH
SMITH              20         20 RESEARCH
JAMES              30         30 SALES
TURNER             30         30 SALES
BLAKE              30         30 SALES
MARTIN             30         30 SALES
WARD               30         30 SALES
ALLEN              30         30 SALES
FORD
                              40 OPERATIONS
15 rows selected.
```

La UNION ALL è in generale più performante della UNION perché, non dovendo scartare i duplicati, Oracle non è costretto ad ordinare i record delle due query per trovare i record uguali.

7.7.3 INTERSECT

L'operatore INTERSECT restituisce l'intersezione insiemistica dei due insiemi di record estratti dalla due singole query.

```
WTO >select nome, cognome from clienti
  2  where cod_fisc is not null;

NOME                           COGNOME
------------------------------ --------------------
LUCA                           NERI
MARCO                          ROSSI
PASQUALE                       RUSSO
GENNARO                        ESPOSITO
MATTEO                         VERDI

WTO >select nome, cognome from clienti
  2  where comune='H501';

NOME                           COGNOME
------------------------------ --------------------
MARCO                          ROSSI
GIOVANNI                       BIANCHI

WTO >select nome, cognome from clienti
  2  where cod_fisc is not null
  3  intersect
  4  select nome, cognome from clienti
  5  where comune='H501';

NOME                           COGNOME
------------------------------ --------------------
MARCO                          ROSSI
```

7.7.4 MINUS

L'operatore MINUS implementa il complemento insiemistico. Quando due query vengono messe in relazione con questo operatore Oracle estrae i record presenti nel risultato della prima query e non presenti nel risultato della seconda.

```
WTO >select nome, cognome from clienti
  2  where cod_fisc is not null;

NOME                           COGNOME
------------------------------ ------------------
LUCA                           NERI
MARCO                          ROSSI
PASQUALE                       RUSSO
GENNARO                        ESPOSITO
MATTEO                         VERDI

WTO >select nome, cognome from clienti
  2  where comune='H501';

NOME                           COGNOME
```

```
------------------------------ ------------------
MARCO                          ROSSI
GIOVANNI                       BIANCHI

WTO >select nome, cognome from clienti where cod_fisc is not null
  2  minus
  3  select nome, cognome from clienti where comune='H501';

NOME                           COGNOME
------------------------------ ------------------
GENNARO                        ESPOSITO
LUCA                           NERI
MATTEO                         VERDI
PASQUALE                       RUSSO
```

Tra gli operatori insiemistici MINUS è l'unico non commutativo, scambiando le due query si ottengono risultati differenti.

```
WTO >select nome, cognome from clienti where comune='H501'
  2  minus
  3  select nome, cognome from clienti where cod_fisc is not null
  4  ;

NOME                           COGNOME
------------------------------ ------------------
GIOVANNI                       BIANCHI
```

7.8 Ricerche innestate

All'interno di una query possono essere incluse altre espressioni di query. Queste prendono il nome di *subquery*.

Una subquery può essere presente praticamente dappertutto, facciamo qualche esempio. L'importo medio delle fatture si calcola con la semplice query.

```
WTO >select avg(importo) from fatture;

AVG(IMPORTO)
------------
         600
```

Ipotizziamo di voler estrarre tutte le fatture e per ognuna mostrare numero ed importo di quella fattura e l'importo medio di tutte le fatture.

Per realizzare questa estrazione è possibile utilizzare la semplice query appena vista come se fosse una colonna da estrarre.

```
WTO >select num_fattura, importo,
  2  (select avg(importo) from fatture) IMPORTO_MEDIO
  3  from fatture;

NUM_FATTURA    IMPORTO IMPORTO_MEDIO
----------- ---------- -------------
          1        300           600
          2        500           600
          3        700           600
          4       1000           600
          5        500           600
```

Un altro modo per ottenere lo stesso risultato è mettere in JOIN la tabella fatture con una subquery che ritorna solo l'importo medio delle fatture. Senza condizione di JOIN visto che la subquery restituisce una sola riga.

```
WTO >select num_fattura, importo, IMPORTO_MEDIO
  2  from fatture, (select avg(importo) IMPORTO_MEDIO
  3                  from fatture);

NUM_FATTURA    IMPORTO IMPORTO_MEDIO
----------- ---------- -------------
          1        300           600
          2        500           600
          3        700           600
          4       1000           600
          5        500           600
```

Se invece vogliamo estrarre solo le fatture che hanno un importo maggiore o uguale all'importo medio la subquery andrà nella clausola WHERE.

```
WTO >select num_fattura, importo
  2  from fatture
  3  where importo>=(select avg(importo) IMPORTO_MEDIO
  4                   from fatture);

NUM_FATTURA    IMPORTO
----------- ----------
          3        700
          4       1000
```

La stessa subquery può essere utilizzata pure nella clausola di ordinamento, sebbene in questo caso non abbia molto senso.

```
WTO >select num_fattura, importo
  2  from fatture
  3  order by (select avg(importo) from fatture);

NUM_FATTURA    IMPORTO
----------- ----------
          1        300
          2        500
          5        500
          4       1000
          3        700
```

Se la subquery viene utilizzata nella lista delle colonne, nella GROUP BY o nella ORDER BY essa deve ritornare al massimo una riga. Se ne torna più di una Oracle solleva un errore.

7.8.1 Operatori di confronto

Nel caso in cui la subquery sia utilizzata nelle clausole WHERE o HAVING essa dovrà essere sicuramente confrontata con altre espressioni.

Nell'esempio del paragrafo precedente l'operatore di confronto utilizzato è "maggiore o uguale". Di conseguenza se la subquery avesse estratto più di una riga Oracle avrebbe dovuto sollevare un errore poiché

l'operatore >= è in grado di gestire il confronto di un valore (IMPORTO in quel caso) con un solo valore (il risultato della subquery).

```
WTO >select num_fattura, importo
  2  from fatture
  3  where importo>=(select importo from fatture);
where importo>=(select importo from fatture)
                *
ERRORE alla riga 3:
ORA-01427: single-row subquery returns more than one row
```

L'esempio precedente mostra proprio l'errore appena descritto. La subquery restituisce più di un record ed Oracle solleva errore.

Oracle incorpora alcuni operatori appositamente creati per gestire il confronto di un singolo valore con i molti record estratti da una subquery.

- ANY

L'operatore di confronto ANY ritorna VERO se almeno un record estratto dalla subquery verifica la condizione. Ipotizziamo ad esempio di volere estrarre tutte le fatture il cui importo è maggiore dell'importo di almeno uno degli ordini.

```
WTO >select num_ordine, importo from ordini;

NUM_ORDINE    IMPORTO
---------- ----------
         1        800
         2        700
         3       1000
         4       1200
         5       1700

WTO >select num_fattura, importo from fatture;

NUM_FATTURA    IMPORTO
----------- ----------
          1        300
          2        500
          3        700
          4       1000
          5        500

WTO >select num_fattura, importo from fatture
  2  where importo > ANY (select importo
  3                          from ordini);

NUM_FATTURA    IMPORTO
----------- ----------
          4       1000
```

Mentre se come condizione avessimo usato "maggiore o uguale" sarebbero stati estratti due record:

```
WTO >select num_fattura, importo from fatture
  2  where importo >= ANY (select importo
  3                          from ordini);

NUM_FATTURA    IMPORTO
----------- ----------
          4       1000
```

```
          3         700
```

- ALL

L'operatore ALL ritorna VERO se la condizione a cui è applicato è vera per tutti i record estratti dalla subquery.

Ipotizziamo ad esempio di volere estrarre gli ordini il cui importo è maggiore dell'importo di tutte le fatture esistenti.

```
WTO >select num_ordine, importo from ordini
  2  where importo > ALL (select importo
  3                        from fatture);

NUM_ORDINE    IMPORTO
---------- ----------
         4       1200
         5       1700
```

- IN (subquery)

L'operatore IN è del tutto analogo a quello già incontrato in precedenza. La differenza consiste nel fatto che la lista dei valori è ottenuta mediante una subquery.

Elenco di tutti i clienti residenti in provincia di Roma.

```
WTO >select nome, cognome, indirizzo
  2  from clienti
  3  where comune in (select cod_comune
  4                     from comuni
  5                    where provincia='RM');

NOME     COGNOME  INDIRIZZO
-------- -------- ----------------------
MATTEO   VERDI    VIA DEL MARE, 8
GIOVANNI BIANCHI  VIA OSTIENSE, 850
MARCO    ROSSI    VIA LAURENTINA, 700
```

È evidente che lo stesso risultato poteva essere ottenuto anche con una JOIN.

```
WTO >select nome, cognome, indirizzo
  2  from clienti, comuni
  3  where comune=cod_comune
  4  and provincia='RM';

NOME     COGNOME  INDIRIZZO
-------- -------- ----------------------
MATTEO   VERDI    VIA DEL MARE, 8
GIOVANNI BIANCHI  VIA OSTIENSE, 850
MARCO    ROSSI    VIA LAURENTINA, 700
```

- EXISTS

L'operatore EXISTS ritorna VERO se la subquery estrae almeno un record.

Si ipotizzi ad esempio di volere visualizzare nome e cognome dei clienti che hanno fatto almeno un ordine.

```
WTO >select nome, cognome
  2  from clienti c
  3  where exists (select * from ordini o
  4                     where o.cod_cliente = c.cod_cliente);

NOME     COGNOME
-------- --------
MARCO    ROSSI
GIOVANNI BIANCHI
LUCA     NERI
AMBROGIO COLOMBO
PASQUALE RUSSO
```

Questa subquery ha una grande differenza rispetto a tutte quelle viste finora: al suo interno c'è un riferimento ad un dato della query esterna (C.COD_CLIENTE). Questo tipo di subquery, dette *correlate*, saranno descritte nel prossimo paragrafo.

A prima vista si potrebbe pensare che la query appena eseguita sia equivalente alla JOIN seguente.

```
WTO >select nome, cognome
  2  from clienti c, ordini o
  3  where o.cod_cliente = c.cod_cliente;

NOME     COGNOME
-------- --------
MARCO    ROSSI
GIOVANNI BIANCHI
LUCA     NERI
AMBROGIO COLOMBO
PASQUALE RUSSO
```

Sebbene esse estraggano lo stesso insieme di record, però, le due query non sono equivalenti. Ce ne accorgiamo aggiungendo agli ordini la riga seguente:

```
WTO >insert into wto_esempio.ordini
  2  (NUM_ORDINE,DATA_ORDINE,COD_CLIENTE,IMPORTO,NOTE)
  3  values (wto_esempio.seq_ordini.nextval,date'2010-12-01',
  4          1,500.00,null);

Creata 1 riga.

WTO >select nome, cognome
  2  from clienti c, ordini o
  3  where o.cod_cliente = c.cod_cliente;

NOME     COGNOME
-------- --------
MARCO    ROSSI
MARCO    ROSSI
GIOVANNI BIANCHI
LUCA     NERI
AMBROGIO COLOMBO
PASQUALE RUSSO

Selezionate 6 righe.

WTO >select nome, cognome
  2  from clienti c
```

```
  3  where exists (select * from ordini o
  4                  where o.cod_cliente = c.cod_cliente);

NOME     COGNOME
-------- --------
MARCO    ROSSI
GIOVANNI BIANCHI
LUCA     NERI
AMBROGIO COLOMBO
PASQUALE RUSSO
```

Il cliente ROSSI adesso ha due ordini, la JOIN dunque duplica i record mentre la query che fa uso della subquery non li duplica.

Per scrivere una query effettivamente equivalente con la JOIN bisogna fare ricorso alla clausola DISTINCT.

```
WTO >select distinct nome, cognome
  2  from clienti c, ordini o
  3  where o.cod_cliente = c.cod_cliente;

NOME     COGNOME
-------- --------
GIOVANNI BIANCHI
LUCA     NERI
MARCO    ROSSI
AMBROGIO COLOMBO
PASQUALE RUSSO
```

7.8.2 Subquery correlate e scorrelate

Come già accennato, tutte le subquery viste finora (tranne l'ultima) sono "scorrelate" nel senso che la subquery non ha alcun riferimento a dati della query esterna. In tal caso Oracle può eseguire dapprima la subquery e successivamente eseguire la query esterna sostituendo la subquery con il valore appena ottenuto. In pratica nel primo esempio che abbiamo visto

```
WTO >select num_fattura, importo,
  2  (select avg(importo) from fatture) IMPORTO_MEDIO
  3  from fatture;

NUM_FATTURA    IMPORTO IMPORTO_MEDIO
----------- ---------- -------------
          1        300           600
          2        500           600
          3        700           600
          4       1000           600
          5        500           600
```

Oracle può eseguire dapprima la subquery

```
WTO >select avg(importo) from fatture;

AVG(IMPORTO)
------------
         600
```

E poi "semplificare" la query esterna sostituendo la subquery con il suo valore.

```
WTO >select num_fattura, importo,
  2  600 IMPORTO_MEDIO
  3  from fatture;

NUM_FATTURA    IMPORTO IMPORTO_MEDIO
----------- ---------- -------------
          1        300           600
          2        500           600
          3        700           600
          4       1000           600
          5        500           600
```

Le subquery correlate, invece, fanno riferimento ad almeno un dato della query esterna e dunque non possono essere eseguite prima della query esterna. Ipotizziamo ad esempio di volere estrarre, per ogni cliente, il nome del cliente ed il codice dell'ultimo ordine a lui relativo.

Una JOIN può aiutare.

```
WTO >select nome, max(num_ordine)
  2  from clienti c, ordini o
  3  where c.cod_cliente=o.cod_cliente
  4  group by nome
  5  ;

NOME     MAX(NUM_ORDINE)
-------- ---------------
AMBROGIO               4
PASQUALE               5
LUCA                   3
MARCO                  1
GIOVANNI               2
```

Ma lo stesso risultato può essere ottenuto anche utilizzando una subquery (correlata).

```
WTO >select nome,
  2 (select max(num_ordine) from ordini o
  3   where o.cod_cliente=c.cod_cliente) max_num_ord
  4  from clienti c;

NOME     MAX_NUM_ORD
-------- -----------
MARCO              1
GIOVANNI           2
MATTEO
LUCA               3
AMBROGIO           4
GENNARO
PASQUALE           5
VINCENZO

Selezionate 8 righe.
```

La differenza è dovuta al fatto che, come sappiamo, la JOIN elimina i clienti che non hanno ordini, la query con subquery è dunque effettivamente equivalente all'OUTER JOIN seguente.

```
WTO >select nome, max(num_ordine)
  2  from clienti c, ordini o
  3  where c.cod_cliente=o.cod_cliente(+)
  4  group by nome
```

```
  5  ;

NOME     MAX(NUM_ORDINE)
-------- ---------------
MATTEO
AMBROGIO               4
PASQUALE               5
LUCA                   3
VINCENZO
MARCO                  1
GIOVANNI               2
GENNARO

Selezionate 8 righe.
```

Chiariamo la differenza di esecuzione di una subquery scorrelata da una correlata.

In una subquery scorrelata, come detto, Oracle può eseguire dapprima la subquery e poi la query esterna utilizzando il valore, o l'insieme di record, ottenuto dalla subquery. È come eseguire due query distinte una dietro l'altra.

In una subquery correlata, invece, Oracle deve partire dalla query esterna. Per ogni record ottenuto dall'esecuzione della query esterna Oracle esegue la subquery. Dopo la subquery torna al prossimo record della query esterna. Ovviamente in Oracle ci sono delle forti ottimizzazioni che consentono di non rieseguire la subquery più volte se non è strettamente necessario, ma comunque il costo di una subquery correlata è molto maggiore del costo di una subquery scorrelata di pari complessità.

7.8.3 La clausola WITH

La clausola WITH consente di associare, subito prima dell'istruzione di query, un nome ad una subquery. In pratica la query definita nella clausola WITH è del tutto analoga ad una vista. La clausola WITH è particolarmente utile per semplificare query che utilizzano complesse subquery nella clausola FROM. Si riprenda ad esempio il secondo esempio relativo alle subquery scorrelate.

```
WTO >select num_fattura, importo, IMPORTO_MEDIO
  2  from fatture, (select avg(importo) IMPORTO_MEDIO
  3                  from fatture);

NUM_FATTURA    IMPORTO IMPORTO_MEDIO
----------- ---------- -------------
          1        300           600
          2        500           600
          3        700           600
          4       1000           600
          5        500           600
```

Utilizzando la clausola WITH è possibile estrarre dalla query la subquery e definirla a priori assegnandole un nome.

```
WTO >with t as
  2  (select avg(importo) IMPORTO_MEDIO from fatture)
  3  select num_fattura, importo, IMPORTO_MEDIO
```

```
  4  from fatture, t;

NUM_FATTURA    IMPORTO IMPORTO_MEDIO
----------- ---------- -------------
          1        300           600
          2        500           600
          3        700           600
          4       1000           600
          5        500           600
```

Nella clausola WITH possono essere definite più viste una dietro l'altra, basta separale con una virgola.

Complichiamo, ad esempio, la query precedente aggiungendo anche l'importo minimo degli ordini.

```
WTO >with media as
  2  (select avg(importo) IMPORTO_MEDIO from fatture),
  3  minimo_ordine as
  4  (select min(importo) MIN_ORDINE from ordini)
  5  select num_fattura, importo, IMPORTO_MEDIO, MIN_ORDINE
  6  from fatture, media, minimo_ordine;

NUM_FATTURA    IMPORTO IMPORTO_MEDIO MIN_ORDINE
----------- ---------- ------------- ----------
          1        300           600        500
          2        500           600        500
          3        700           600        500
          4       1000           600        500
          5        500           600        500
```

7.9 Comandi TCL

Quando un utente si collega al database ed esegue un qualunque comando SQL viene avviata una transazione. Una transazione è un'unita di lavoro atomica che si completa quando l'utente decide di confermare o annullare tutte le modifiche in essa effettuate. I comandi TCL consentono di gestire correttamente le transazioni.

7.9.1 Gestione delle transazioni

Come detto Oracle avvia automaticamente una transazione quando un utente collegato al database esegue un qualunque comando SQL. Se il comando eseguito è un DDL la transazione finisce immediatamente dopo il comando poiché questi comandi vengono automaticamente confermati da Oracle, non sono annullabili.

7.9.2 COMMIT

Per default ogni transazione legge solo i dati esplicitamente confermati da altre transazioni ed i dati modificati, anche se ancora non confermati, nella propria transazione.

Facciamo un esempio. Apriamo due sessioni di SQL*plus utilizzando lo stesso utente. Negli esempi seguenti ho utilizzato il prompt "WT1" per indicare la prima sessione ed il prompt "WT2" per indicare la seconda sessione.

La sessione 1 crea una tabella.

```
WT1 >create table test_tr (a number, b varchar2(10));

Tabella creata.
```

Poiché il comando CREATE TABLE è un DDL esso viene esplicitamente confermato da Oracle ed anche la sessione 2 vede la tabella subito.

```
WT2 >desc test_tr
 Nome                  Nullo?    Tipo
 ----------------- -------- ------------
 A                              NUMBER
 B                              VARCHAR2(10)
```

Se la sessione 1 inserisce un record in tabella avvia una transazione.

```
WT1 >insert into test_tr values (1,'ins wt1');

Creata 1 riga.

WT1 >select * from test_tr
  2  ;

         A B
---------- ----------
         1 ins wt1
```

La sessione 1 vede il proprio inserimento anche se non lo ha ancora confermato, viceversa la sessione 2 non lo vede.

```
WT2 >select * from test_tr;

Nessuna riga selezionata
```

Se un utente vuole confermare esplicitamente una modifica apportata e renderla visibile a tutte le altre sessioni deve utilizzare il comando COMMIT.

```
WT1 >commit;

Commit completato.
```

Successivamente anche la sessione 2 vede il record inserito.

```
WT2 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
```

Ovviamente la conferma può essere eseguita implicitamente da Oracle se l'utente esegue un comando DDL. Ipotizziamo ad esempio che la sessione 1 inserisca una seconda riga e poi crei un'ulteriore tabella.

```
WT1 >insert into test_tr values (2,'ins wt1');

Creata 1 riga.

WT1 >create table test_2 (x number);

Tabella creata.
```

L'utente non ha esplicitamente confermato l'insert ma Oracle ha implicitamente confermato e chiuso l'intera transazione al comando DDL. L'altra sessione, infatti, vede entrambe le modifiche.

```
WT2 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
         2 ins wt1

WT2 >desc test_2;
 Nome                 Nullo?   Tipo
 ------------------ --------- ------------
 X                             NUMBER
```

7.9.3 ROLLBACK

Il comando ROLLBACK consente di annullare tutte le modifiche apportate durante la transazione.

```
WT1 >insert into test_tr values (3,'ins wt1');

Creata 1 riga.

WT1 >insert into test_tr values (4,'ins wt1');

Creata 1 riga.

WT1 >insert into test_2 values (100);

Creata 1 riga.

WT1 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
         2 ins wt1
         3 ins wt1
         4 ins wt1

WT1 >select * from test_2;

         X
----------
       100

WT1 >rollback;

Rollback completato.

WT1 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
         2 ins wt1

WT1 >select * from test_2;
```

```
Nessuna riga selezionata
```

7.9.4 SAVEPOINT

Il comando SAVEPOINT consente di inserire dei segnaposto all'interno della transazione in modo che sia possibile, mediante una opzione del comando ROLLBACK, annullare parzialmente le modifiche fatte durante la transazione.

Nell'esempio seguente vengono definiti alcuni savepoint all'interno della transazione e poi si annullano alcuni comandi eseguiti.

Innanzitutto si inserisce una riga in TEST_TR e si crea un savepoint.

```
WT1 >insert into test_tr values (3,'ins wt1');

Creata 1 riga.

WT1 >savepoint rec_3;

Creato savepoint.
```

Poi si inserisce un record in TEST_2 e si definisce un nuovo savepoint.

```
WT1 >insert into test_2 values (200);

Creata 1 riga.

WT1 >savepoint rec_200;

Creato savepoint.
```

Successivamente si inserisce un nuovo record in TEST_TR.

```
WT1 >insert into test_tr values (4,'ins wt1');

Creata 1 riga.

WT1 >select * from test_tr;

        A B
---------- ----------
        1 ins wt1
        2 ins wt1
        3 ins wt1
        4 ins wt1

WT1 >select * from test_2;

        X
----------
      200
```

Adesso è possibile annullare solo l'ultimo inserimento utilizzando il comando ROLLBACK con l'opzione TO rec_2000.

```
WT1 >rollback to rec_200;

Rollback completato.
```

Il comando annulla tutto ciò che era stato fatto successivamente al savepoint rec_200.

```
WT1 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
         2 ins wt1
         3 ins wt1

WT1 >select * from test_2;

         X
----------
       200
```

Allo stesso modo, un ROLLBACK al savepoint rec_3 annulla l'inserimento in TEST_2.

```
WT1 >rollback to rec_3;

Rollback completato.

WT1 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
         2 ins wt1
         3 ins wt1

WT1 >select * from test_2;

Nessuna riga selezionata
```

Adesso confermiamo l'unica modifica rimasta, la prima che era stata eseguita.

```
WT1 >commit;

Commit completato.

WT1 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
         2 ins wt1
         3 ins wt1
```

Un ulteriore ROLLBACK non produrrà risultato, ormai la sessione è stata confermata.

```
WT1 >rollback;
```

```
Rollback completato.

WT1 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
         2 ins wt1
         3 ins wt1
```

7.9.5 SET TRANSACTION

Finora abbiamo visto esempi di transazioni avviate implicitamente da Oracle quando l'utente eseguiva il primo comando SQL.

L'utente può esplicitamente avviare una transazione utilizzando il comando SET TRANSACTION.

Il comando SET TRANSACTION può essere completato con varie opzioni particolarmente utili.

Il comando SET TRANSACTION READ ONLY apre una transazione di sola lettura. Tutti i comandi di select eseguiti durante la transazione leggeranno i dati come erano all'istante in cui la transazione è cominciata. Questo vale anche se, durante la transazione, un altro utente modifica le tabelle e conferma le modifiche.

La sessione 1 apre la transazione in modalità READ ONLY. Nel momento in cui la apre in TEST_TR ci sono tre record.

```
WT1 >set transaction read only;

Impostata transazione.

WT1 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
         2 ins wt1
         3 ins wt1
```

La sessione 2 inserisce un record in TEST_TR e fa COMMIT. In una situazione normale anche la sessione 1 dovrebbe vedere la modifica.

```
WT2 >insert into test_tr values (4,'ins wt2');

Creata 1 riga.

WT2 >commit;

Commit completato.
```

Invece la sessione 2 non se ne accorge.

```
WT1 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
```

```
         2 ins wt1
         3 ins wt1
```

Almeno fino a fine transazione.

```
WT1 >rollback;

Rollback completato.

WT1 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
         2 ins wt1
         3 ins wt1
         4 ins wt2
```

Ovviamente durante una transazione READ ONLY non si possono fare modifiche al DB.

```
WT1 >set transaction read only;

Impostata transazione.

WT1 >update test_tr set a=1;
update test_tr set a=1
       *
ERRORE alla riga 1:
ORA-01456: may not perform insert/delete/update operation inside a READ
ONLY transaction
```

L'alternativa ad una transazione READ ONLY è una transazione READ WRITE, cioè una transazione standard in cui si possono modificare i dati e si leggono le modifiche appena confermate dalle altre transazioni.

Un'altra opzione importante del comando SET TRANSACTION è quella che consente di impostare il livello di isolamento, cioè di regolare come le diverse transazioni influiscono l'una sull'altra.

Per default il livello di isolamento utilizzato dalle transazioni Oracle è READ COMMITTED. Ciò significa che una transazione legge solo dati che sono stati confermati dalle altre. In fase di modifica se una transazione cerca di aggiornare un record già aggiornato, ma non confermato, da un'altra transazione, resta in attesa fino a quando l'altra transazione non conferma o annulla.

Per comprendere meglio facciamo un esempio. La sessione 1 aggiorna il primo record di TEST_TR.

```
WT1 >select * from test_tr;

         A B
---------- ----------
         1 ins wt1
         2 ins wt1
         3 ins wt1
         4 ins wt2

WT1 >update test_tr set b='agg'
  2  where a=1;
```

```
Aggiornata 1 riga.

WT1 >select * from test_tr;

         A B
---------- ----------
         1 agg
         2 ins wt1
         3 ins wt1
         4 ins wt2
```

La sessione 2 non vede la modifica, se cerca di eseguire un aggiornamento sulla medesima riga resta in attesa.

```
WT2 >update test_tr
  2  set a=10 where a=1;
```

E l'attesa dura indefinitamente, fino a quando la sessione 1 non conferma o annulla. Ipotizziamo che confermi.

```
WT1 >commit;

Commit completato.
```

Allora la sessione 2 porta a completamente il suo aggiornamento. Ed è visibile l'effetto di entrambe le modifiche.

```
WT2 >update test_tr
  2  set a=10 where a=1;

Aggiornata 1 riga.

WT2 >select * from test_tr;

         A B
---------- ----------
        10 agg
         2 ins wt1
         3 ins wt1
         4 ins wt2

WT2 >commit;

Commit completato.
```

L'altro livello di isolamento che si può utilizzare è SERIALIZABLE. Questo livello di isolamento prevede che una transazione non possa vedere e modificare record modificati e committati in un'altra transazione se tali modifiche erano state apportate ma non confermate all'istante dell'apertura della transazione serializzabile. Ci vuole un esempio.

Ipotizziamo che la sessione 2 aggiorni un record in TEST_TR.

```
WT2 >select * from test_tr;

         A B
---------- ----------
        10 agg
         2 ins wt1
         3 ins wt1
```

```
         4 ins wt2

WT2 >update test_tr
  2  set b='agg2'
  3  where a=2;

Aggiornata 1 riga.

WT2 >select * from test_tr;

         A B
---------- ----------
        10 agg
         2 agg2
         3 ins wt1
         4 ins wt2
```

A questo punto la sessione 1 avvia una transazione con livello di isolamento SERIALIZABLE ed ovviamente non vede le modifiche fatte dalla sessione 2.

```
WT1 >set transaction isolation level serializable;

Impostata transazione.

WT1 >select * from test_tr;

         A B
---------- ----------
        10 agg
         2 ins wt1
         3 ins wt1
         4 ins wt2
```

Questo sarebbe successo con qualunque tipo di transazione. La differenza si manifesta quando la sessione 2 conferma la sua modifica.

```
WT2 >commit;

Commit completato.
```

A questo punto in una normale transazione READ COMMITTED la sessione 1 vedrebbe le modifiche apportate dalla sessione 2 e potrebbe a sua volta aggiornare il medesimo record.

In una transazione SERIALIZABLE, invece, ciò non è possibile.

```
WT1 >update test_tr set a=20 where a=2;
update test_tr set a=20 where a=2
       *
ERRORE alla riga 1:
ORA-08177: can't serialize access for this transaction

WT1 >select * from test_tr;

         A B
---------- ----------
        10 agg
         2 ins wt1
         3 ins wt1
         4 ins wt2
```

La transazione serializzabile è partita quando la modifica della sessione 2 non era stata ancora committata e dunque questa transazione non può utilizzare quei dati anche se essi sono stati successivamente confermati.

7.9.6 Lock di dati ed oggetti

Come fa Oracle a garantire che i livelli di isolamento vengano rispettati? Vengono utilizzati i cosiddetti LOCK di risorse. Gli identificativi delle risorse che sono riservate ad una transazione e non possono essere utilizzate dalle altre vengono inseriti in un'opportuna tabella di sistema e rimossi solo quando la transazione le rilascia. La gestione dei lock e delle transazioni che li detengono è tema di amministrazione e dunque non sarà approfondita in questo corso. In questo paragrafo mi limito a mostrare il comando LOCK TABLE che consente di mettere in lock esclusivo una tabella e riservarla alla transazione corrente.

La sessione 1 pone un lock sulla tabella TEST_TR.

```
WT1 >lock table test_tr in exclusive mode nowait;

Tabella/e bloccata/e.
```

A questo punto la sessione 2 non può eseguire su di essa comandi di aggiornamento, se ci prova resta in attesa che la transazione della sessione 1 finisca.

```
WT2 >update test_tr
  2  set b='agg3'
  3  where a=3;
```

Quando la sessione 1 chiude la transazione.

```
WT1 >roll;
Completato rollback.
```

La sessione 2 va avanti.

```
WT2 >update test_tr
  2  set b='agg3'
  3  where a=3;

Aggiornata 1 riga.

WT2 >select * from test_tr;

         A B
---------- ----------
        10 agg
         2 agg2
         3 agg3
         4 ins wt2

WT2 >commit;

Commit completato.
```

Il comando LOCK TABLE fallisce quando nella tabella c'è già almeno un record riservato.

Ipotizziamo che la sessione 2 esegua un comando di update.

```
WT2 >update test_tr
  2  set b='agg4'
  3  where a=4;

Aggiornata 1 riga.
```

La sessione 1 non riuscirà ad eseguire il comando LOCK TABLE.

```
WT1 >lock table test_tr in exclusive mode nowait;
lock table test_tr in exclusive mode nowait
           *
ERRORE alla riga 1:
ORA-00054: resource busy and acquire with NOWAIT specified
```

7.9.7 Una situazione particolare, il deadlock

Abbiamo visto che quando una transazione acquisisce un lock esclusivo su una risorsa nessun’altra transazione può modificare la stessa risorsa. Se ci prova resta in attesa indefinitamente finché la prima transazione non libera la risorsa.

A causa di questo comportamento si può verificare che due transazioni restino in attesa entrambe che si liberi una risorsa su cui l’altra transazione ha acquisito un lock. Questa situazione di stallo prende il nome di DEADLOCK e può essere risolta solo mediante un intervento arbitrario di Oracle che annulla una delle ultime operazioni che hanno causato il blocco.

Facciamo un esempio.

La sessione 1 acquisisce un lock sulla riga numero tre della tabella TEST_TR.

```
WT1 >update test_tr set b='x' where a=3;

Aggiornata 1 riga.
```

La sessione 2 acquisisce un lock sulla riga numero quattro della stessa tabella.

```
WT2 >update test_tr set b='x' where a=4;

Aggiornata 1 riga.
```

La sessione 1 cerca di aggiornare la riga numero quattro. Poiché quella riga è riservata alla sessione 2, la sessione 1 resta in attesa.

```
WT1 >update test_tr set b='x' where a=4;
```

La sessione 2 tenta di aggiornare la riga numero tre. Poiché essa è riservata alla sessione 1 anche la sessione 2 deve restare in attesa.

```
WT2 >update test_tr set b='x' where a=3;
```

Ma è evidente che questa situazione non ha futuro, nessuna delle due sessioni ha il controllo per poter sbloccare. Di conseguenza ci pensa Oracle ad eliminare l’ultimo comando della sessione 1.

```
WT1 >update test_tr set b='x' where a=4;
update test_tr set b='x' where a=4
       *
ERRORE alla riga 1:
```

```
ORA-00060: deadlock detected while waiting for resource
```

La sessione 2 è ancora in attesa, Oracle ha annullato solo l'ultimo commando della sessione 1, non il primo update quindi la sessione 1 detiene ancora il lock sulla riga tre e la sessione 2 deve attendere.

Quando la sessione 1 rilascia la risorsa chiudendo la transazione, la sessione 2 si libera a sua volta.

```
WT1 >rollback;

Rollback completato.
```

A questo punto la sessione 2 effettua l'aggiornamento e poi può chiudere la transazione.

```
WT2 >update test_tr set b='x' where a=3;

Aggiornata 1 riga.

WT2 >commit;

Commit completato.
```

7.10 Comandi DCL

Ogni utente ha normalmente diritto di accedere esclusivamente agli oggetti definiti nel proprio schema. Utenti particolarmente privilegiati, poi, hanno diritto di accesso a tutti gli oggetti presenti nel database. In questo paragrafo, senza entrare nel merito di attività di amministrazione, descriveremo i due comandi SQL che consentono di concedere e rimuovere un privilegio di accesso ad un utente.

Per gli esempi lavoreremo questa volta con due utenti distinti: SCOTT e WTO_ESEMPIO. Per distinguere le due sessioni WTO_ESEMPIO avrà come prompt la stringa "WTO" mentre SCOTT avrà "SCO".

7.10.1 GRANT

Il comando GRANT consente di concedere ad un utente il privilegio di eseguire una determinata azione su un oggetto.

La sintassi del comando è

```
GRANT <privilegio> ON <oggetto> TO <utente>
```

Ipotizziamo ad esempio che l'utente SCOTT voglia concedere all'utente WTO_ESEMPIO il privilegio di effettuare SELECT sulla tabella EMP.

```
SCO >grant select on emp to wto_esempio;

Concessione riuscita.
```

A questo punto l'utente WTO_ESEMPIO potrà eseguire il seguente comando.

```
WTO >select ename, job from scott.emp;

ENAME      JOB
```

```
---------- ---------
SMITH      CLERK
ALLEN      SALESMAN
WARD       SALESMAN
JONES      MANAGER
MARTIN     SALESMAN
BLAKE      MANAGER
CLARK      MANAGER
SCOTT      ANALYST
KING       PRESIDENT
TURNER     SALESMAN
ADAMS      CLERK
JAMES      CLERK
FORD       ANALYST
MILLER     CLERK

Selezionate 14 righe.
```

Ricordandosi che il nome della tabella deve essere qualificata con il nome dello schema perché non si trova nello schema di WTO_ESEMPIO.

WTO_ESEMPIO può eseguire solo la SELECT sulla tabella. Se prova ad eseguire una diversa azione per cui non ha ricevuto il privilegio otterrà un errore.

```
WTO >update scott.emp set ename='FORD2' where ename='FORD';
update scott.emp set ename='FORD2' where ename='FORD'
             *
ERRORE alla riga 1:
ORA-01031: insufficient privileges
```

La lista dei privilegi che possono essere concessi su un oggetto è lunga e dipende dal tipo di oggetto. Qui nominiamo solo SELECT, INSERT, UPDATE, DELETE e ALTER. Il significato dei privilegi si intuisce facilmente dal nome.

Per concedere più privilegi sullo stesso oggetto allo stesso utente questi possono essere indicati nella stessa istruzione GRANT separati da virgola.

```
SCO >grant select, insert, update on emp to wto_esempio;

Concessione riuscita.
```

Oltre a questi privilegi a livello di oggetto il comando GRANT consente anche di concedere privilegi di sistema che danno all'utente una serie più ampia di facoltà. Tali privilegi sono però in genere concessi nell'ambito di attività di amministrazione e dunque sono al di là degli obbiettivi di questo manuale.

7.10.2 REVOKE

La revoca di uno o più privilegi precedentemente concessi ad un utente si realizza con il comando REVOKE. La sintassi è

```
REVOKE <privilegio> ON <oggetto> FROM <utente>
```

Per privare dunque WTO_ESEMPIO del diritto di modificare i dati di EMP, SCOTT dovrà eseguire il seguente comando.

```
SCO >revoke update on emp from wto_esempio;

Revoca riuscita.
```

7.10.3 RUOLI

Può essere utile accorpare in un unico contenitore un insieme di privilegi in modo da poterli concedere o revocare tutti insieme. Un contenitore di privilegi si chiama RUOLO.

La creazione del ruolo si realizza col comando CREATE ROLE. L'associazione al ruolo dei privilegi si esegue mediante il comando GRANT.

Ad esempio ipotizziamo che l'utente SCOTT voglia creare un ruolo RUOLO_TEST che includa i privilegi di SELECT ed UPDATE su EMP ed il privilegio di SELECT su DEPT.

```
SCO >create role ruolo_test;

Ruolo creato.

SCO >grant insert, update on emp to ruolo_test;

Concessione riuscita.

SCO >grant select on dept to ruolo_test;

Concessione riuscita.
```

A questo punto SCOTT dispone di un contenitore di privilegi che può assegnare o revocare ad altri utenti.

```
SCO >grant ruolo_test to wto_esempio;

Concessione riuscita.

SCO >revoke ruolo_test from wto_esempio;

Revoca riuscita.
```

8 PL/SQL

Nel 1966 gli informatici italiani Böhm e Jacopini enunciarono un importante teorema che porta il loro nome. Un qualunque linguaggio di programmazione che disponga delle strutture di controllo sequenziale, condizionale ed iterativa può implementare qualunque algoritmo.

Non disponendo delle strutture sequenziali citate, l'SQL non soddisfa le condizioni del teorema di Böhm-Jacopini. Oracle sopperisce a tale mancanza grazie ad un linguaggio di programmazione che estende il linguaggio SQL aggiungendo una serie di strutture di controllo che consentano di implementare qualunque algoritmo. Tale linguaggio ha preso il nome di PL/SQL (*Procedural Language based on SQL*, Linguaggio procedurale basato su SQL).

In questo capitolo vengono introdotti i fondamenti del linguaggio PL/SQL. Insieme al linguaggio Oracle fornisce una vasta gamma di librerie PL/SQL che estendono enormemente le capacità del linguaggio. Alcune di queste librerie sono introdotte nel capitolo "Argomenti avanzati di PL/SQL". Le librerie presentate in questo capitolo devono essere considerate solo un piccolo esempio delle enormi potenzialità che il PL/SQL ha raggiunto.

Per una trattazione completa del linguaggio si rimanda al manuale Oracle "PL/SQL Language Reference", per il dettaglio sulle librerie predefinite al manuale "PL/SQL Packages and types Reference". Entrambi sono liberamente scaricabili dal sito http://www.oracle.com/.

Questo capitolo non è un'introduzione alla programmazione. Si assume che il lettore abbia già dimestichezza con i principali concetti di base.

8.1 Blocchi PL/SQL anonimi

Un programma PL/SQL può essere eseguito *una tantum* lanciandolo da SQL*Plus o SQL Developer oppure memorizzato nel database per essere eseguito più volte. Nel primo caso si parla di blocco PL/SQL anonimo.

8.2 Struttura di un blocco PL/SQL anonimo

Un blocco PL/SQL anonimo è suddiviso in tre aree delimitate dalle quattro parole chiave DECLARE, BEGIN, EXCEPTION; END.

```
DECLARE

BEGIN

EXCEPTION

END;
```

L'area che segue la parola chiave DECLARE, fino a BEGIN, è riservata alla dichiarazione delle variabili e costanti. E' un'area facoltativa, può essere omessa.

L'area che segue la parola chiave BEGIN e continua fino ad EXCEPTION oppure, in assenza della parola chiave EXCEPTION, fino ad END, è dedicata alle istruzioni operative del programma. L'area delle istruzioni operative è obbligatoria.

L'area che segue la parola chiave EXCEPTION e continua fino alla parola chiave END è riservata alla gestione degli errori. Si tratta di un'area facoltativa.

Sia nell'area delle istruzioni operative che nell'area di gestione delle eccezioni possono essere inclusi (innestati) altri blocchi PL/SQL.

Ogni singola istruzione, o costrutto, del linguaggio deve essere conclusa con un punto e virgola. L'intero blocco, essendo esso stesso un costrutto, deve essere concluso con un punto e virgola dopo la parola chiave END.

In tutti (quasi) i manuali di programmazione il primo esempio è un programma che stampa a video la stringa "Hello world!". Non possiamo essere da meno.

```
WTO >set serverout on
WTO >begin
  2  dbms_output.put_line('Hello World!');
  3  end;
  4  /
Hello World!

Procedura PL/SQL completata correttamente.
```

L'esempio, per quanto semplice, richiede alcuni commenti.

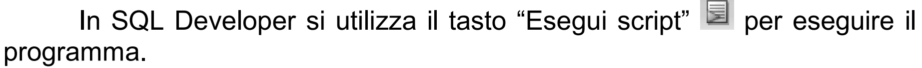
Il comando SQL*Plus `set serverout on` serve ad abilitare la visualizzazione dei messaggi stampati dai programmi PL/SQL. Non è un comando SQL ed è necessario solo quando si lavora da SQL*Plus. Se l'avessimo omesso il programma sarebbe stato eseguito correttamente ma non avrebbe stampato il messaggio.

Il comando

```
dbms_output.put_line('Hello World!');
```

indica ad Oracle di stampare in output il messaggio contenuto tra parentesi.

Lo slash posto sotto la parola chiave END indica a SQL*Plus che deve eseguire il programma. Per i comandi SQL avevamo visto che si potevano indifferentemente utilizzare lo slash o il punto e virgola. In PL/SQL il punto e virgola ha il significato di “fine singola istruzione” quindi per dire a SQL*Plus di eseguire il programma si può utilizzare solo lo slash. Lo slash ovviamente non fa parte del programma PL/SQL.

In SQL Developer si utilizza il tasto “Esegui script”  per eseguire il programma.

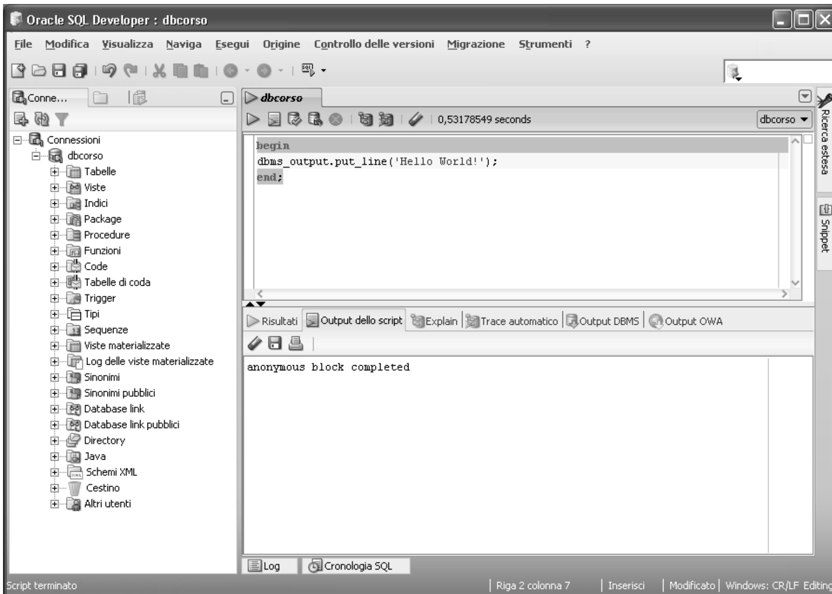


Figura 8-1 Esecuzione del programma PL/SQL in SQL Developer.

L’output del programma può essere consultato nella finestra “Output DBMS” in basso.

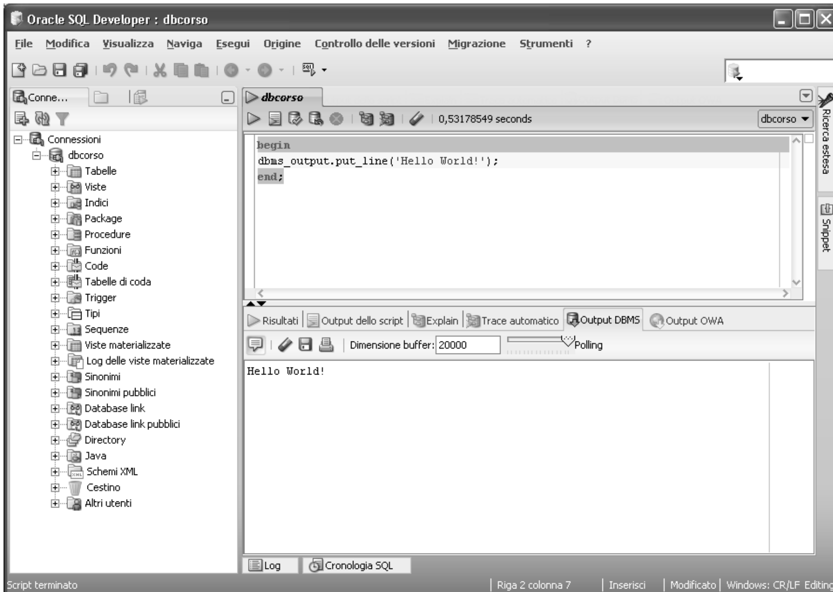


Figura 8-2 Visualizzazione dell'output in SQL Developer.

In qualunque punto di un programma PL/SQL è possibile inserire un commento che sarà ignorato da Oracle in due modi:

- Includendo il commento tra i caratteri /* e */, in tal caso il commento può proseguire anche su più righe.
- Inserendo il commento dopo un doppio trattino --, in tal caso il commento continua fino a fine riga.

Ad esempio:

```
DECLARE
  A VARCHAR2(3); --Commento che continua fino a fine linea
BEGIN
  /*commento che continua anche
    Sulle righe successive
    A quella in cui è iniziato*/
  dbms_output.put_line('Hello World!');
END;
```

8.3 Dichiarazione di costanti e variabili

8.3.1 Tipi di dato

I tipi di dato elementari utilizzabili in PL/SQL sono grossomodo gli stessi che si utilizzano in SQL. Le differenze degne di nota sono le seguenti:

- il tipo BOOLEAN, esistente solo in PL/SQL. Questo tipo consente di definire variabili booleane che possono assumere i valori TRUE o FALSE;
- i tipi VARCHAR2, CHAR, RAW possono avere lunghezza fino a 32676 byte;

- i tipi CLOB e BLOB possono contenere fino a 128TB di dati.

In aggiunta ai tipi di dato elementari, il programmatore può definire tipi di dato personalizzati. Questi saranno trattati più avanti in questo capitolo.

8.3.2 Dichiarazione di variabili

Una variabile è un'area di memoria il cui contenuto può cambiare nel corso dell'esecuzione del programma e che può essere utilizzata nelle istruzioni del programma con un nome simbolico.

Il nome delle variabili deve rispettare le seguenti regole:

- Deve cominciare con una lettera.
- Può includere lettere, numeri ed i caratteri dollaro ($), cancelletto (#) e trattino basso (_).
- Può essere lungo al massimo 30 caratteri.
- Non può essere uguale ad una delle parole riservate (per un elenco delle parole riservate si veda il manuale PL/SQL Language Reference, Appendice D).

La dichiarazione di una variabile avviene nell'apposita area del blocco PL/SQL indicandone il nome, il tipo ed eventualmente il valore iniziale.

```
DECLARE
  <nome variabile> <tipo> := <valore iniziale>;
BEGIN
```

Nell'esempio seguente vengono dichiarate quattro variabili: una stringa, un numero una variabile booleana ed una data. Le prime tre hanno un valore iniziale predefinito:

```
DECLARE
  Variabile_stringa  VARCHAR2(100) := 'Esempio';
  Variabile_numero   NUMBER(10,4)  := 1234.9872;
  Variabile_booleana BOOLEAN       := TRUE;
  Variabile_data     DATE;
BEGIN
```

Le variabili non inizializzate esplicitamente vengono comunque inizializzate da Oracle al valore NULL.

Una variabile può essere definita dello stesso tipo di una colonna di database. In questo modo se la colonna di database dovesse cambiare tipo in futuro non sarebbe necessario aggiornare il programma. Tale dichiarazione utilizza la parola chiave %TYPE in coda al nome della colonna.

Nell'esempio seguente la variabile VAR_CLIENTE è definita dello stesso tipo della colonna COD_CLIENTE presente nella tabella CLIENTI.

```
DECLARE
  VAR_CLIENTE  CLIENTI.COD_CLIENTE%TYPE;
BEGIN
```

8.3.3 Dichiarazione di costanti

Una costante è un'area di memoria il cui contenuto non cambia nel corso dell'esecuzione del programma e che può essere utilizzata nelle istruzioni del programma con un nome simbolico.

Una costante è in tutto identica ad una variabile a parte il fatto che nel corso del programma non è possibile modificarne il valore.

Nella dichiarazione si utilizza la parola chiave CONSTANT come segue:

```
DECLARE
  PI_GRECO  CONSTANT NUMBER(3,2) := 3.14;
BEGIN
```

8.4 Costrutti di base

Si è già accennato al fatto che il PL/SQL nasce come estensione procedurale dell'SQL. Per realizzare tale estensione sono state introdotte le strutture di controllo che vengono descritte nei prossimi paragrafi.

Tutti i costrutti descritti possono essere utilizzati nell'area delle istruzioni operative e nell'area di gestione degli errori, non nell'area di dichiarazione delle variabili.

8.4.1 Assegnazione di un valore.

Per assegnare un valore ad una variabile si utilizza l'operatore di assegnazione :=.

Ad esempio il programma seguente dichiara la variabile A con valore iniziale tre, ne stampa il contenuto, le assegna il valore sette e poi la stampa di nuovo.

```
WTO >declare
  2  a number:= 3;
  3  begin
  4    dbms_output.put_line(a);
  5    A := 7;
  6    dbms_output.put_line(a);
  7  end;
  8  /
3
7

Procedura PL/SQL completata correttamente.
```

8.4.2 Strutture Condizionali

La struttura condizionale IF THEN ELSE consente al programmatore di eseguire istruzioni differenti in funzione del fatto che una determinata condizione sia vera o falsa. La struttura assume la seguente forma:

```
BEGIN
  IF <condizione1> THEN
     <istruzioni1>
  ELSIF <condizione2> THEN
     <istruzioni2>
...ALTRI BLOCCHI ELSIF...
```

```
  ELSE
     <istruzioniN>
  END IF;
END;
```

Il significato è molto intuitivo: Se la condizione1 è vera esegui le istruzioni1. Se invece è vera la condizione2 (e la condizione1 è falsa) esegui le istruzioni2. Altrimenti (se tutte le condizioni indicate sopra sono false) esegui le istruzioniN.

Come esempio nel seguente programma sono dichiarate due variabili numeriche A e B. Alle variabili vengono assegnati due valori predefiniti. Nel corpo del programma se A è maggiore di B si stampa “A maggiore di B”, se A è minore di B si stampa “A minore di B” altrimenti si stampa “A e B sono uguali”.

```
WTO >declare
  2  a number := 3;
  3  b number := 2;
  4  begin
  5    if a>b then
  6      dbms_output.put_line('A maggiore di B');
  7    elsif a<b then
  8      dbms_output.put_line('A minore di B');
  9    else
 10      dbms_output.put_line('A e B sono uguali');
 11    end if;
 12  end;
 13  /
A maggiore di B

Procedura PL/SQL completata correttamente.
```

Se si utilizzano diversi valori di A e B si ottengono tutti i possibili output:

```
WTO >declare
  2  a number := 3;
  3  b number := 4;
  4  begin
  5    if a>b then
  6      dbms_output.put_line('A maggiore di B');
  7    elsif a<b then
  8      dbms_output.put_line('A minore di B');
  9    else
 10      dbms_output.put_line('A e B sono uguali');
 11    end if;
 12  end;
 13  /
A minore di B

Procedura PL/SQL completata correttamente.

WTO >declare
  2  a number := 3;
  3  b number := 3;
  4  begin
  5    if a>b then
  6      dbms_output.put_line('A maggiore di B');
  7    elsif a<b then
  8      dbms_output.put_line('A minore di B');
  9    else
 10      dbms_output.put_line('A e B sono uguali');
```

```
 11    end if;
 12  end;
 13  /
A e B sono uguali

Procedura PL/SQL completata correttamente.
```

In questi esempi è stato ignorato il caso in cui almeno una tra A e B sia NULL. In tal caso valgono tutte le considerazioni fatte per l'SQL: si possono utilizzare le stesse funzioni per la gestione dei valori nulli e gli operatori di confronto IS NULL ed IS NOT NULL.

La struttura condizionale CASE serve a decodificare un valore. È simile alla CASE di primo tipo vista in SQL:

```
DECLARE
  <nome variabile> <tipo>;
BEGIN
  <nome variabile> :=
  CASE <dato da controllare>
   WHEN <valore1> THEN
     <valore di ritorno1>
   WHEN <valore2> THEN
     <valore di ritorno2>
...ALTRI BLOCCHI WHEN...
  ELSE
     <valore di ritornoN>
  END;
END;
```

Ipotizziamo ad esempio di avere una variabile numerica VOTO e di volere ottenere in output un giudizio sintetico in funzione del voto.

```
WTO >declare
  2    voto number := 8;
  3    giudizio varchar2(20);
  4  begin
  5    giudizio := case voto
  6     when 10 then 'Eccellente'
  7     when 9 then 'Ottimo'
  8     when 8 then 'Distinto'
  9     when 7 then 'Buono'
 10     when 6 then 'Sufficiente'
 11     when 5 then 'Mediocre'
 12     else 'Insufficiente'
 13    end;
 14    dbms_output.put_line('Giudizio:'||giudizio);
 15  end;
 16  /
Giudizio:Distinto

Procedura PL/SQL completata correttamente.
```

Il programma prima di tutto dichiara le due variabili VOTO e GIUDIZIO. Nell'area delle istruzioni operative determina il valore di GIUDIZIO sulla base del VOTO utilizzando una CASE e poi stampa il giudizio ottenuto. Nella stampa è stato utilizzato l'operatore di concatenazione per unire la stringa fissa "Giudizio:" al giudizio sintetico contenuto nella variabile GIUDIZIO.

Cambiando il voto di partenza si ottengono i diversi giudizi sintetici.

```
WTO >declare
  2    voto number := 6;
  3    giudizio varchar2(20);
  4  begin
  5    giudizio := case voto
  6     when 10 then 'Eccellente'
  7     when 9 then 'Ottimo'
  8     when 8 then 'Distinto'
  9     when 7 then 'Buono'
 10     when 6 then 'Sufficiente'
 11     when 5 then 'Mediocre'
 12     else 'Insufficiente'
 13    end;
 14    dbms_output.put_line('Giudizio:'||giudizio);
 15  end;
 16  /
Giudizio:Sufficiente

Procedura PL/SQL completata correttamente.

WTO >declare
  2    voto number := 3;
  3    giudizio varchar2(20);
  4  begin
  5    giudizio := case voto
  6     when 10 then 'Eccellente'
  7     when 9 then 'Ottimo'
  8     when 8 then 'Distinto'
  9     when 7 then 'Buono'
 10     when 6 then 'Sufficiente'
 11     when 5 then 'Mediocre'
 12     else 'Insufficiente'
 13    end;
 14    dbms_output.put_line('Giudizio:'||giudizio);
 15  end;
 16  /
Giudizio:Insufficiente

Procedura PL/SQL completata correttamente.
```

8.4.3 Strutture Iterative

Una struttura iterativa consente di ripetere una determinata operazione un numero di volte che, al momento della scrittura del programma, è imprecisato e, al momento dell’esecuzione del programma, è determinato dal fatto che una data condizione sia vera o falsa. Ad esempio una struttura iterativa consente di incrementare di uno il valore di una variabile numerica finché raggiunge il valore di un’altra variabile. Al momento della scrittura del programma i valori delle due variabili potrebbero essere ignoti, e dunque ignoto è il numero di volte che l’incremento dovrà avvenire.

La struttura iterativa di base il PL/SQL assume la forma

```
LOOP
  <istruzioni1>
  EXIT WHEN <condizione>
  <istruzioni2>
END LOOP;
```

Quando Oracle incontra in un programma questa struttura esegue le istruzioni1, poi testa la condizione, se questa è vera salta alla fine del LOOP e prosegue, se la condizione è falsa esegue le istruzioni2 e ricomincia eseguendo le istruzioni1. A questo punto verifica di nuovo la condizione e si comporta come al giro precedente. Ovviamente la condizione ad un certo punto deve diventare falsa, altrimenti si genera un ciclo infinito ed il programma non finisce più (almeno finché non consuma tutta la memoria o determina qualche altro errore…).

Come esempio scriviamo un programma che stampi tutti i numeri da uno al giorno corrente del mese. Al momento della scrittura del programma non si sa quanti numeri bisogna stampare, perché il programma potrebbe girare un giorno qualsiasi.

Prima di tutto si dichiara una variabile numerica che è inizializzata ad uno. Nel corpo del programma c'è il LOOP, ad ogni iterazione si stampa il valore della variabile e poi si controlla se tale valore ha raggiunto il giorno corrente. Se la condizione è vera si esce dal LOOP, altrimenti si incrementa la variabile.

```
WTO >declare
  2   g number:=1;
  3  begin
  4   loop
  5    dbms_output.put_line('giorno '||g);
  6    exit when g=extract(day from systimestamp);
  7    g := g+1;
  8   end loop;
  9  end;
 10  /
giorno 1
giorno 2
giorno 3
giorno 4
giorno 5
giorno 6
giorno 7
giorno 8
giorno 9
giorno 10

Procedura PL/SQL completata correttamente.

WTO >select sysdate from dual;

SYSDATE
---------
10-MAR-11
```

Se il programma non stampa nulla bisogna ricordarsi di attivare la visualizzazione dell'output.

```
WTO >SET SEVEROUT ON
```

Il LOOP generico appena mostrato può soddisfare qualunque esigenza di programmazione, la possibilità di posizionare la condizione d'uscita (EXIT WHEN) prima, dopo oppure in mezzo alle istruzioni operative rende questa struttura estremamente flessibile. Per agevolare i

programmatori che hanno familiarità con altri linguaggi, Oracle ha introdotto in PL/SQL altre due strutture iterative che sono casi particolari del LOOP appena presentato. Si tratta del FOR e del WHILE.

Il ciclo FOR si applica quando la condizione di uscita è di un tipo particolare: una variabile si incrementa fino a raggiungere un determinato valore. È proprio il caso visto nell'esempio precedente. La struttura FOR assume la sintassi seguente:

```
FOR <variabile> IN <valore1>..<valore2> LOOP
  <istruzioni>
END LOOP;
```

La variabile non deve essere dichiarata, è dichiarata automaticamente da Oracle, è anche automaticamente inizializzata al valore1 ed incrementata di uno ad ogni iterazione. Il ciclo termina quando la variabile raggiunge il valore2.

L'esempio precedente si può riscrivere.

```
WTO >begin
  2    for g in 1..extract(day from systimestamp) loop
  3      dbms_output.put_line('giorno '||g);
  4    end loop;
  5  end;
  6  /
giorno 1
giorno 2
giorno 3
giorno 4
giorno 5
giorno 6
giorno 7
giorno 8
giorno 9
giorno 10

Procedura PL/SQL completata correttamente.
```

Se anziché incrementare è necessario decrementare la variabile partendo da valore2 ed arrivando a valore1 si può utilizzare la parola chiave REVERSE prima dei valori:

```
WTO >begin
  2    for g in REVERSE 1..extract(day from systimestamp) loop
  3      dbms_output.put_line('giorno '||g);
  4    end loop;
  5  end;
  6  /
giorno 10
giorno 9
giorno 8
giorno 7
giorno 6
giorno 5
giorno 4
giorno 3
giorno 2
giorno 1

Procedura PL/SQL completata correttamente.
```

La struttura FOR è molto utilizzata perché semplifica la programmazione non richiedendo la dichiarazione, l'inizializzazione e l'incremento espliciti della variabile di ciclo.

Il ciclo WHILE si applica a qualunque condizione ma tale condizione è controllata ad inizio ciclo, prima dell'esecuzione delle istruzioni operative. La sintassi della struttura è:

```
WHILE <condizione> LOOP
  <istruzioni>
END LOOP;
```

Le istruzioni sono eseguite quando la condizione è vera, se la condizione diventa false Oracle esce dal LOOP. La condizione viene dunque testata al contrario rispetto al LOOP base.

```
WTO >declare
  2    g number:=1;
  3   begin
  4     while g<=extract(day from systimestamp) loop
  5        dbms_output.put_line('giorno '||g);
  6        g := g+1;
  7     end loop;
  8   end;
  9   /
giorno 1
giorno 2
giorno 3
giorno 4
giorno 5
giorno 6
giorno 7
giorno 8
giorno 9
giorno 10

Procedura PL/SQL completata correttamente.
```

8.4.4 Strutture Sequenziali

La struttura GOTO consente di far saltare il flusso del programma ad un'istruzione precedentemente contrassegnata con un'etichetta.

La struttura assume la sintassi

```
GOTO <etichetta>
  <istruzioni1>
<<<etichetta>>>
  <istruzioni2>
```

Quando Oracle incontra la GOTO non esegue le istruzioni1 e salta direttamente alle istruzioni2. Nell'esempio seguente se il giorno corrente è maggiore di 10 il programma salta alcune istruzioni e stampa il giorno corrente.

```
WTO >declare
  2     oggi number:= extract(day from systimestamp);
  3   begin
  4     dbms_output.put_line('giorno '||oggi);
  5     if oggi>10 then
  6        goto salta_qui;
```

```
  7    end if;
  8    oggi := 0;
  9  <<salta_qui>>
 10    dbms_output.put_line('giorno dopo il salto '||oggi);
 11  end;
 12  /
giorno 17
giorno dopo il salto 17

Procedura PL/SQL completata correttamente.
```

Poiché oggi è 17 il programma entra nell'IF ed esegue il salto all'etichetta SALTA_QUI. Di conseguenza l'istruzione `OGGI:=0`

Non è eseguita e dopo il salto si ottiene di nuovo 17.

Se il programma si fa girare tra il giorno uno ed il giorno nove del mese si ottiene:

```
WTO >declare
  2    oggi number:= extract(day from systimestamp);
  3  begin
  4    dbms_output.put_line('giorno '||oggi);
  5    if oggi>10 then
  6       goto salta_qui;
  7    end if;
  8    oggi := 0;
  9  <<salta_qui>>
 10    dbms_output.put_line('giorno dopo il salto '||oggi);
 11  end;
 12  /
giorno 3
giorno dopo il salto 0

Procedura PL/SQL completata correttamente.
```

Quindi Oracle non entra nell'IF e non effettua il salto. Viene eseguito l'azzeramento della variabile OGGI che, dunque, alla seconda stampa vale zero.

La struttura GOTO è diffusamente considerata il "nemico pubblico numero uno" della buona programmazione. In effetti, questa struttura invita il programmatore a fare salti incontrollati e se ne può tranquillamente fare a meno. È però importante sottolineare che si può facilmente scrivere un buon programma usando le GOTO e si può altrettanto facilmente scrivere un pessimo programma senza utilizzare questa struttura.

L'istruzione NULL non esegue nessuna operazione. In alcuni casi Oracle richiede necessariamente che in una struttura ci sia un'istruzione. Ad esempio non è possibile scrivere un'IF senza istruzioni al proprio interno:

```
WTO >begin
  2   if 1>2 then
  3   end if;
  4  end;
  5  /
 end if;
 *
ERRORE alla riga 3:
ORA-06550: line 3, column 2:
```

```
PLS-00103: Encountered the symbol "END" when expecting one of the
following:
begin case declare exit for goto if loop mod null pragma
raise return select update while with <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
close current delete fetch lock insert open rollback
savepoint set sql execute commit forall merge pipe
```

L'istruzione NULL può essere utilizzata per riempire tali spazi obbligatori senza eseguire, in effetti, nessuna operazione.

```
WTO >begin
  2   if 1>2 then
  3     null;
  4   end if;
  5  end;
  6  /

Procedura PL/SQL completata correttamente.
```

Inutile dire che se si è costretti a mettere l'istruzione NULL in una struttura senza istruzioni c'è puzza di bruciato, conviene rivedere la logica del programma.

L'istruzione NULL è anche molto (e spesso erroneamente) utilizzata nella gestione degli errori quando si vuole che il verificarsi di un determinato errore non abbia effetti sul programma. Se ne vedranno esempi più avanti.

8.5 Utilizzo dell'SQL

Il PL/SQL nasce come estensione procedurale dell'SQL. Fin qui ne sono stati velocemente descritti gli aspetti procedurali, adesso ne vengono introdotte le caratteristiche tipicamente SQL.

8.5.1 SQL Statico

In PL/SQL possono essere eseguiti direttamente comandi DML (INSERT, UPDATE, DELETE, SELECT) e TCL (COMMIT, ROLLBACK, SAVEPOINT). L'unica differenza notevole rispetto all'SQL introdotto nei capitoli precedenti è sul comando di query. In PL/SQL, infatti, nel comando SELECT è necessario specificare in quali variabili del programma Oracle deve conservare i dati estratti. Tale dettaglio operativo deve essere specificato mediante la clausola aggiuntiva INTO, che non esiste nell'SQL standard.

Facciamo un esempio. Scriviamo un programma PL/SQL che stampa a video il numero dei clienti presenti in tabella CLIENTI.

Da SQL scriveremmo

```
WTO >select count(*) from clienti;

  COUNT(*)
----------
         8
```

In PL/SQL bisogna avere a disposizione una variabile in cui scaricare il valore letto dalla query e poi stampare la variabile.

```
WTO >declare
  2   num_clienti number;
  3  begin
  4   select count(*)
  5     into num_clienti
  6     from clienti;
  7   dbms_output.put_line('Ci sono '||num_clienti||' clienti.');
  8  end;
  9  /
Ci sono 8 clienti.

Procedura PL/SQL completata correttamente.
```

Se la query restituisce più colonne bisogna avere più variabili. Ipotizziamo ad esempio di voler leggere il numero, il massimo importo e l'importo medio delle fatture:

```
WTO >declare
  2    num_f number;
  3    max_f number;
  4    avg_f number;
  5  begin
  6    select count(*), max(importo), avg(importo)
  7      into num_f, max_f, avg_f
  8      from fatture;
  9
 10    dbms_output.put_line('Ci sono '||num_f||' fatture.');
 11    dbms_output.put_line('La più alta è di '||max_f||' euro');
 12    dbms_output.put_line('La media è di '||avg_f||' euro');
 13
 14  end;
 15  /
Ci sono 5 fatture.
La più alta è di 1000 euro
La media è di 600 euro

Procedura PL/SQL completata correttamente.
```

8.5.2 SQL Dinamico

Per tutte le istruzioni SQL che non rientrano tra quelle per cui si può utilizzare l'SQL statico è possibile costruire il comando SQL come stringa e poi eseguirlo.

L'esecuzione dinamica di SQL si esegue grazie all'istruzione EXECUTE IMMEDIATE a cui si passa la stringa contenente il comando da eseguire ed eventualmente altri parametri. Per introdurre l'istruzione ad esempio si può utilizzare il caso in cui un programma PL/SQL deve creare una tabella.

Con un'istruzione SQL statica ciò è vietato.

```
WTO >begin
  2   create table abcd (a number);
  3  end;
  4  /
 create table abcd (a number);
 *
ERRORE alla riga 2:
ORA-06550: line 2, column 2:
```

```
PLS-00103: Encountered the symbol "CREATE" when expecting one of the
following:
begin case declare exit for goto if loop mod null pragma
raise return select update while with <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
close current delete fetch lock insert open rollback
savepoint set sql execute commit forall merge pipe
```

Invece lo stesso commando può essere inserito in una stringa ed eseguito in maniera dinamica.

```
WTO >declare
  2   s varchar2(100):='create table abcd (a number)';
  3  begin
  4   execute immediate s;
  5  end;
  6  /

Procedura PL/SQL completata correttamente.

WTO >desc abcd
 Nome                       Nullo?   Tipo
 -------------------------- -------- ---------
 A                                   NUMBER
```

Il comando EXECUTE IMMEDIATE può essere utilizzato anche per eseguire delle query. In tal caso bisogna specificare la clausola INTO, non nell'SQL da eseguire ma nell'EXECUTE IMMEDIATE.

```
WTO >declare
  2    num_f number;
  3    max_f number;
  4    avg_f number;
  5    s varchar2(100):=
  6   'select count(*), max(importo), avg(importo) from fatture';
  7  begin
  8    execute immediate s into num_f, max_f, avg_f;
  9    dbms_output.put_line('Ci sono '||num_f||' fatture.');
 10    dbms_output.put_line('La più alta è di '||max_f||' euro');
 11    dbms_output.put_line('La media è di '||avg_f||' euro');
 12
 13  end;
 14  /
Ci sono 5 fatture.
La più alta è di 1000 euro
La media è di 600 euro

Procedura PL/SQL completata correttamente.
```

Talvolta può essere necessario utilizzare più volte la stessa istruzione SQL dinamica cambiando solo alcuni valori. In tal caso è possibile utilizzare, nella stringa che rappresenta l'istruzione, delle variabili di sostituzione precedute dal carattere due punti e poi chiamare la EXECUTE IMMEDIATE passando il valore effettivo mediante la clausola USING. Nell'esempio seguente l'istruzione SQL dinamica conta il numero di clienti residenti in un generico comune :c.

Nel corpo del programma la EXECUTE IMMEDIATE è eseguita due volte, una volta per Roma (H501) e la seconda volta per Napoli (F839).

```
WTO >declare
  2    num_r number;
  3    s varchar2(100):=
  4  'select count(*) from clienti where comune=:c';
  5  begin
  6    execute immediate s into num_r using 'H501';
  7    dbms_output.put_line('Ho '||num_r||' clienti a ROMA.');
  8    execute immediate s into num_r using 'F839';
  9    dbms_output.put_line('Ho '||num_r||' clienti a NAPOLI.');
 10  end;
 11  /
Ho 2 clienti a ROMA.
Ho 2 clienti a NAPOLI.

Procedura PL/SQL completata correttamente.
```

Il vantaggio dell'SQL dinamico è evidente: consente di eseguire qualunque istruzione SQL valida. Lo svantaggio è che ci si accorge di eventuali errori nell'istruzione SQL dinamica solo se il programma tenta effettivamente di eseguire l'istruzione SQL. Viceversa, se in un programma si utilizza SQL statico Oracle controlla immediatamente se l'istruzione SQL è corretta, anche se poi all'esecuzione il programma in effetti non deve utilizzare quell'istruzione.

Facciamo un esempio. Se il giorno corrente è maggiore di dieci stampiamo il numero totale dei clienti, altrimenti il numero totale delle fatture. Utilizzando l'SQL statico.

```
WTO >declare
  2    r number;
  3  begin
  4    if extract(day from systimestamp)>10 then
  5      select count(*) into r from clienti;
  6    else
  7      select count(*) into r from fatture;
  8    end if;
  9    dbms_output.put_line('Ci sono '||r||' righe in tabella.');
 10  end;
 11  /
Ci sono 8 righe in tabella.

Procedura PL/SQL completata correttamente.
```

Utilizzando l'SQL dinamico

```
WTO >declare
  2    r number;
  3    s_cli varchar2(100):= 'select count(*) from clienti';
  4    s_fat varchar2(100):= 'select count(*) from fatture';
  5  begin
  6    if extract(day from systimestamp)>10 then
  7       execute immediate s_cli into r;
  8    else
  9       execute immediate s_fat into r;
 10    end if;
 11    dbms_output.put_line('Ci sono '||r||' righe in tabella.');
 12  end;
 13  /
Ci sono 8 righe in tabella.

Procedura PL/SQL completata correttamente.
```

I due programmi sono equivalenti. Proviamo solo per un momento a rinominare la tabella FATTURE in FATTURE_NEW.

```
WTO >rename fatture to fatture_new;

Tabella rinominata.
```

La tabella FATTURE non esiste più.

```
WTO >desc fatture
ERROR:
ORA-04043: object fatture does not exist
```

Dunque il programma che utilizza l'SQL statico va in errore, anche se effettivamente essendo oggi 17 non deve eseguire la query da FATTURE, ma da CLIENTI.

```
WTO >declare
  2    r number;
  3  begin
  4    if extract(day from systimestamp)>10 then
  5      select count(*) into r from clienti;
  6    else
  7      select count(*) into r from fatture;
  8    end if;
  9    dbms_output.put_line('Ci sono '||r||' righe in tabella.');
 10  end;
 11  /
    select count(*) into r from fatture;
                                *
ERRORE alla riga 7:
ORA-06550: line 7, column 37:
PL/SQL: ORA-00942: table or view does not exist
ORA-06550: line 7, column 5:
PL/SQL: SQL Statement ignored
```

Viceversa il programma che utilizza SQL dinamico funziona, si accorgerebbe dell'errore solo se provasse effettivamente a fare la query da FATTURE.

```
WTO >declare
  2    r number;
  3    s_cli varchar2(100):= 'select count(*) from clienti';
  4    s_fat varchar2(100):= 'select count(*) from fatture';
  5  begin
  6    if extract(day from systimestamp)>10 then
  7       execute immediate s_cli into r;
  8    else
  9       execute immediate s_fat into r;
 10    end if;
 11    dbms_output.put_line('Ci sono '||r||' righe in tabella.');
 12  end;
 13  /
Ci sono 8 righe in tabella.

Procedura PL/SQL completata correttamente.
```

Il fatto che in questo caso il programma continui a funzionare potrebbe sembrare un vantaggio, in realtà è uno svantaggio perché l'esempio evidenzia come l'SQL dinamico nasconde gli errori di programmazione.

8.5.3 CURSORI

Gli esempi di query SQL visti finora, sia nella modalità statica che nella modalità dinamica, estraggono un singolo record dal database. Nel caso in cui la query estragga un insieme di record la clausola INTO genera un errore. Chiaramente è spesso necessario leggere molti record dal DB, per gestire quest'esigenza il PL/SQL mette a disposizione i cursori. Un cursore non è altro che un nome simbolico assegnato ad una query. Il cursore si dichiara nella sezione DECLARE e poi si utilizza (eseguendo effettivamente la query e scorrendo le righe estratte) nella sezione delle istruzioni operative.

Lo scorrimento di un cursore richiede un LOOP, non essendo noto a priori quanti record ci saranno nel cursore.

Nell'esempio seguente viene dichiarato un cursore che estrae nome, cognome e comune dei clienti, il cursore viene poi aperto (cioè viene eseguita la query), scorso ed infine chiuso. Solo i clienti residenti a Napoli o Roma vengono stampati a video.

```
WTO >declare
  2    v_nome clienti.nome%type;
  3    v_cogn clienti.cognome%type;
  4    v_com  clienti.comune%type;
  5    cursor c is
  6     select nome, cognome, comune
  7       from clienti;
  8  begin
  9   open c;
 10   loop
 11    fetch c into v_nome, v_cogn, v_com;
 12    exit when c%notfound;
 13    if v_com in ('H501','F839') then
 14      dbms_output.put_line('Cliente '||v_nome||' '||v_cogn);
 15    end if;
 16   end loop;
 17   close c;
 18  end;
 19  /
Cliente MARCO ROSSI
Cliente GIOVANNI BIANCHI
Cliente GENNARO ESPOSITO
Cliente PASQUALE RUSSO

Procedura PL/SQL completata correttamente.
```

Il comando OPEN "apre" il cursore, cioè esegue la query sul DB.

Il comando FETCH punta il prossimo record del cursore (il primo record la prima volta) e scarica il valore delle singole colonne nelle variabili indicate nella clausola INTO.

L'attributo %NOTFOUND del cursore può essere invocato subito dopo una FETCH e restituisce TRUE se un nuovo record è stato trovato oppure FALSE se non ci sono più record nel cursore. È quindi perfetto per essere utilizzato nella clausola EXIT WHEN del LOOP e determinare la condizione d'uscita (non ci sono più record nel cursore).

Il comando CLOSE chiude il cursore liberando la memoria ad esso associata.

Oltre all'attributo %NOTFOUND ce ne sono un paio particolarmente utili. L'attributo %ISOPEN restituisce TRUE se il cursore è stato già aperto e FALSE se non è stato ancora aperto. Serve a prevenire il rischio di chiudere un cursore che non è mai stato aperto evitando così un errore.

Ipotizziamo, infatti, di scrivere un semplice programma in cui si chiude un cursore mai aperto.

```
WTO >declare
  2   cursor c is select sysdate from dual;
  3  begin
  4   if 1>2 then
  5     open c;
  6   end if;
  7   close c;
  8  end;
  9  /
declare
*
ERRORE alla riga 1:
ORA-01001: invalid cursor
ORA-06512: at line 7
```

Poiché la OPEN è in un'IF in cui il programma non entrerà mai la CLOSE non ha senso e genera un errore.

Se dunque c'è il rischio che si tenti di chiudere un cursore mai aperto si può mettere la CLOSE in un'IF come quella che segue.

```
WTO >declare
  2   cursor c is select sysdate from dual;
  3  begin
  4   if 1>2 then
  5     open c;
  6   end if;
  7   if c%isopen then
  8      close c;
  9   end if;
 10  end;
 11  /

Procedura PL/SQL completata correttamente.
```

L'altro attributo interessante è %ROWCOUNT. Esso restituisce le righe del cursore lette con una FETCH fino a quel momento.

A fine ciclo (prima della CLOSE ovviamente) rappresenta quindi il numero totale dei record presenti nel cursore.

```
WTO >declare
  2    v_nome clienti.nome%type;
  3    v_cogn clienti.cognome%type;
  4    v_com  clienti.comune%type;
  5    cursor c is
  6     select nome, cognome, comune
  7       from clienti;
  8  begin
  9    open c;
 10    loop
 11      fetch c into v_nome, v_cogn, v_com;
 12      exit when c%notfound;
 13      dbms_output.put_line('Letta la riga '||c%rowcount);
 14      if v_com in ('H501','F839') then
 15        dbms_output.put_line('Cliente '||v_nome||' '||v_cogn);
 16      end if;
 17    end loop;
 18    close c;
 19  end;
 20  /
Letta la riga 1
Cliente MARCO ROSSI
Letta la riga 2
Cliente GIOVANNI BIANCHI
Letta la riga 3
Letta la riga 4
Letta la riga 5
Letta la riga 6
Cliente GENNARO ESPOSITO
Letta la riga 7
Cliente PASQUALE RUSSO
Letta la riga 8

Procedura PL/SQL completata correttamente.
```

Quando bisogna leggere tutte le righe di un cursore c'è un LOOP semplificato che si può utilizzare, si tratta del CURSOR FOR LOOP.

Questa struttura rispetta la seguente sintassi.

```
FOR <record> IN <cursore> LOOP
   <istruzioni>
END LOOP;
```

Oracle definisce implicitamente una struttura complessa di tipo record (le approfondiremo più avanti) per ospitare tutte le colonne estratte dal cursore, apre il cursore, effettua la fetch per tutti i record, gestisce automaticamente l'uscita dal LOOP quando i record si esauriscono ed alla fine chiama implicitamente anche la CLOSE.

L'esempio precedente diventa.

```
WTO >declare
  2    cursor c is
  3     select nome, cognome, comune
  4       from clienti;
  5  begin
  6   for r in c loop
  7    dbms_output.put_line('Letta la riga '||c%rowcount);
  8    if r.comune in ('H501','F839') then
  9     dbms_output.put_line('Cliente '||r.nome||' '||r.cognome);
 10    end if;
```

```
 11   end loop;
 12  end;
 13  /
Letta la riga 1
Cliente MARCO ROSSI
Letta la riga 2
Cliente GIOVANNI BIANCHI
Letta la riga 3
Letta la riga 4
Letta la riga 5
Letta la riga 6
Cliente GENNARO ESPOSITO
Letta la riga 7
Cliente PASQUALE RUSSO
Letta la riga 8

Procedura PL/SQL completata correttamente.
```

Dove il record r ha un campo per ogni colonna estratta dal cursore, questo campo ha lo stesso nome della colonna.

Il CURSOR FOR LOOP è di gran lunga più utilizzato del LOOP standard visto che di solito è necessario scorrere tutte le righe presenti in un cursore e la sintassi è enormemente facilitata.

8.6 Gestione delle eccezioni

8.6.1 Blocco Exception

Se nel corso del programma si verifica un errore Oracle passa il controllo alla parte di codice presente dopo la parola chiave EXCEPTION. Poiché tale parte del blocco è facoltativa, in sua assenza Oracle passa il controllo al programma che ha chiamato il blocco PL/SQL segnalando l'errore. Facciamo un esempio banale.

```
WTO >set serverout on
WTO >declare
  2  a number;
  3  begin
  4    a := 1/0;
  5    dbms_output.put_line('A='||a);
  6  end;
  7  /
declare
*
ERRORE alla riga 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 4
```

Il programma esegue una divisione per zero, ciò causa un errore e dunque la stampa prevista dopo l'assegnazione non viene eseguita.

Poiché non c'è gestione degli errori (manca la parola chiave EXCEPTION) il controllo viene ritornato a SQL*Plus con la segnalazione d'errore.

Modifichiamo il programma aggiungendo la gestione dell'errore.

```
WTO >declare
  2  a number;
  3  begin
  4    a := 1/0;
  5    dbms_output.put_line('A='||a);
  6  exception
  7    when zero_divide then
  8      dbms_output.put_line('Divisione per zero non ammessa.');
  9  end;
 10  /
Divisione per zero non ammessa.

Procedura PL/SQL completata correttamente.
```

La gestione dell'errore si realizza mediante la clausola

```
WHEN <eccezione> THEN
  <istruzioni>
```

In questo coso il programma, in caso di eccezione ZERO_DIVIDE, stampa un messaggio. Dopo la gestione dell'errore il controllo non torna al punto dove l'errore si era verificato, ma continua con il resto del blocco (in questo caso non c'è nient'altro dopo l'errore).

L'eccezione ZERO_DIVIDE è un'eccezione predefinita, ne esistono altre che saranno descritte nel prossimo paragrafo.

In uno stesso blocco possono essere gestite più eccezioni.

```
WTO >declare
  2  a number;
  3  begin
  4    a := 1/0;
  5    dbms_output.put_line('A='||a);
  6  exception
  7    when zero_divide then
  8      dbms_output.put_line('Divisione per zero non ammessa.');
  9    when no_data_found then
 10      dbms_output.put_line('Dati non trovati.');
 11  end;
 12  /
Divisione per zero non ammessa.

Procedura PL/SQL completata correttamente.
```

8.6.2 Eccezioni predefinite

Oracle fornisce alcune eccezioni predefinite che consentono di gestire molti errori frequenti. Di seguito sono introdotte le più utilizzate, per una lista completa si può fare riferimento al manuale "PL/SQL Language Reference".

DUP_VAL_ON_INDEX

Si verifica nel caso in cui si cerca, con un UPDATE o con un INSERT, di inserire un valore duplicato in una colonna su cui è definito un indice univoco (ad esempio una primary key). Ad esempio nella tabella CLIENTI è prevista la PRIMARY KEY sulla colonna COD_CLIENTE, che dunque non può essere duplicata. In tabella sono presenti per tale campo i valori da uno a otto.

Il programma seguente cerca di aggiornare il codice cliente otto a sette, causando una duplicazione di valore e dunque un errore.

```
WTO >begin
  2    update clienti
  3       set cod_cliente=7
  4     where cod_cliente=8;
  5  end;
  6  /
begin
*
ERRORE alla riga 1:
ORA-00001: unique constraint (WTO_ESEMPIO.CLI_PK) violated
ORA-06512: at line 2
```

Gestendo l'eccezione possiamo evitare che il programma vada in errore e dare un messaggio all'utente.

```
WTO >begin
  2    update clienti
  3       set cod_cliente=7
  4     where cod_cliente=8;
  5  exception
  6    when dup_val_on_index then
  7      dbms_output.put_line('Valore duplicato, non aggiorno.');
  8  end;
  9  /
Valore duplicato, non aggiorno.

Procedura PL/SQL completata correttamente.
```

INVALID_NUMBER

Si verifica nel caso in cui viene effettuata una conversione di una stringa a numero e la stringa non contiene un numero valido.

Nell'esempio seguente si cerca di convertire il codice fiscale dei clienti in numero.

```
WTO >declare
  2  a number;
  3  begin
  4    select to_number(cod_fisc)
  5      into a
  6      from clienti;
  7  end;
  8  /
declare
*
ERRORE alla riga 1:
ORA-01722: invalid number
ORA-06512: at line 4


WTO >declare
  2  a number;
  3  begin
  4    select to_number(cod_fisc)
  5      into a
  6      from clienti;
  7  exception
  8    when invalid_number then
  9      dbms_output.put_line('Numero non valido.');
```

```
 10  end;
 11  /
Numero non valido.

Procedura PL/SQL completata correttamente.
```

NO_DATA_FOUND

Si verifica quando una SELECT..INTO non estrae nessun record.

```
WTO >declare
  2  a number;
  3  begin
  4    select cod_cliente
  5      into a
  6      from clienti
  7     where nome='ASTOLFO';
  8  end;
  9  /
declare
*
ERRORE alla riga 1:
ORA-01403: no data found
ORA-06512: at line 4


WTO >declare
  2  a number;
  3  begin
  4    select cod_cliente
  5      into a
  6      from clienti
  7     where nome='ASTOLFO';
  8  exception
  9    when no_data_found then
 10     dbms_output.put_line('Cliente non trovato.');
 11  end;
 12  /
Cliente non trovato.

Procedura PL/SQL completata correttamente.
```

TOO_MANY_ROWS

Si verifica quando una SELECT..INTO estrae più di un record.

```
WTO >declare
  2  a number;
  3  begin
  4    select cod_cliente
  5      into a
  6      from clienti
  7     where comune='H501';
  8  end;
  9  /
declare
*
ERRORE alla riga 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
```

```
WTO >declare
  2  a number;
  3  begin
  4    select cod_cliente
  5      into a
  6      from clienti
  7     where comune='H501';
  8  exception
  9    when too_many_rows then
 10     dbms_output.put_line('Più di un cliente estratto.');
 11  end;
 12  /
Più di un cliente estratto.

Procedura PL/SQL completata correttamente.
```

VALUE_ERROR

Si verifica quando si mette in una variabile un valore non valido. Nell'esempio seguente si cerca di assegnare il numero 5000 ad una variabile definita NUMBER(3).

```
WTO >declare
  2  a number(3);
  3  begin
  4    a:= 5000;
  5  end;
  6  /
declare
*
ERRORE alla riga 1:
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 4

WTO >declare
  2  a number(3);
  3  begin
  4    a:= 5000;
  5  exception
  6   when value_error then
  7     dbms_output.put_line('Valore non valido.');
  8  end;
  9  /
Valore non valido.

Procedura PL/SQL completata correttamente.
```

ZERO_DIVIDE

Si verifica quando si cerca di effettuare una divisione per zero. Alcuni esempi sono stati mostrati nel paragrafo precedente.

OTHERS

È un'eccezione generica che si utilizza con il significato "qualunque altro errore si verifichi". Nell'esempio seguente il programma gestisce gli errori NO_DATA_FOUND e TOO_MANY_ROWS ma quello che si verifica è INVALID_NUMBER che non è esplicitamente gestito. Il flusso finisce dunque nel WHEN OTHERS dove c'è un messaggio generico.

```
WTO >declare
  2  a number;
  3  begin
  4    select to_number(cod_fisc)
  5      into a
  6      from clienti
  7     where cod_cliente=1;
  8  exception
  9    when no_data_found then
 10     dbms_output.put_line('Cliente non trovato.');
 11    when too_many_rows then
 12     dbms_output.put_line('Più di un cliente estratto.');
 13    when others then
 14     dbms_output.put_line('Errore generico.');
 15  end;
 16  /
Errore generico.

Procedura PL/SQL completata correttamente.
```

8.6.3 SQLCODE e SQLERRM

Nell'ultimo esempio si è visto come avviene la gestione di un errore generico. In tal caso, ovviamente, è importante segnalare all'utente quale errore si è verificato. Per tale ragione all'interno del blocco EXCEPTION è possibile utilizzare le due funzioni SQLCODE e SQLERRM.

La funzione SQLCODE restituisce il codice Oracle dell'errore.

```
WTO >declare
  2  a number;
  3  begin
  4    select to_number(cod_fisc)
  5      into a
  6      from clienti
  7     where cod_cliente=1;
  8  exception
  9    when no_data_found then
 10     dbms_output.put_line('Cliente non trovato.');
 11    when too_many_rows then
 12     dbms_output.put_line('Più di un cliente estratto.');
 13    when others then
 14     dbms_output.put_line('Errore generico '||sqlcode);
 15  end;
 16  /
Errore generico -1722

Procedura PL/SQL completata correttamente.
```

La funzione SQLERRM restituisce una stringa comprendente il codice ed il messaggio d'errore.

```
WTO >declare
  2  a number;
  3  begin
  4    select to_number(cod_fisc)
  5      into a
  6      from clienti
  7     where cod_cliente=1;
  8  exception
  9    when no_data_found then
 10     dbms_output.put_line('Cliente non trovato.');
```

```
 11    when too_many_rows then
 12     dbms_output.put_line('Più di un cliente estratto.');
 13    when others then
 14     dbms_output.put_line('Errore generico.');
 15     dbms_output.put_line(sqlerrm);
 16  end;
 17  /
Errore generico.
ORA-01722: invalid number

Procedura PL/SQL completata correttamente.
```

8.6.4 Sollevare un'eccezione

Come abbiamo visto negli esempi precedenti Oracle solleva automaticamente le eccezioni predefinite quando si verifica l'errore corrispondente. È però possibile sollevare un'eccezione esplicitamente utilizzando il comando RAISE.

Nell'esempio seguente non si verifica nessun errore, non c'è nemmeno una query nel programma. Ciononostante viene esplicitamente sollevata l'eccezione NO_DATA_FOUND ed il controllo finisce nel blocco EXCEPTION.

```
WTO >begin
  2    raise no_data_found;
  3  exception
  4    when no_data_found then
  5     dbms_output.put_line('Cliente non trovato.');
  6  end;
  7  /
Cliente non trovato.

Procedura PL/SQL completata correttamente.
```

Sollevare esplicitamente un'eccezione significa far credere ad Oracle che si sia verificato effettivamente l'errore associato a quell'eccezione. Ciò implica che il codice di errore che giunge al chiamante sarà quello associato all'eccezione. Per sollevare un errore utente utilizzando un codice d'errore a piacere è possibile utilizzare il comando RAISE_APPLICATION_ERROR.

Tale procedura riceve in input un codice d'errore non usato da Oracle (compreso tra -20000 e -20999) ed un messaggio e solleva l'errore.

```
WTO >begin
  2  raise_application_error(-20123,'Errore utente.');
  3  end;
  4  /
begin
*
ERRORE alla riga 1:
ORA-20123: Errore utente.
ORA-06512: at line 2
```

Nell'esempio precedente l'errore definito dall'utente è sollevato e non gestito e dunque torna a SQL*Plus.

Nell'esempio seguente l'errore definito dall'utente è gestito mediante WHEN OTHERS. Le funzioni SQLCODE e SQLERRM sono state utilizzate per mostrare quale errore si verifica.

```
WTO >begin
  2  raise_application_error(-20123,'Errore utente.');
  3  exception
  4    when others then
  5      dbms_output.put_line('Errore generico.');
  6      dbms_output.put_line('Codice='||sqlcode);
  7      dbms_output.put_line(sqlerrm);
  8  end;
  9  /
Errore generico.
Codice=-20123
ORA-20123: Errore utente.

Procedura PL/SQL completata correttamente.
```

8.6.5 Eccezioni utente

Oltre a quelle predefinite è possibile utilizzare altre eccezioni definite dall'utente.

Nell'esempio seguente si dichiara un'eccezione utente DOPO_IL_15 che viene poi sollevata mediante il comando RAISE solo se il giorno corrente è maggiore di 15. L'eccezione è gestita nel blocco EXCEPTION.

```
WTO >declare
  2    dopo_il_15 exception;
  3  begin
  4    if extract(day from systimestamp)>10 then
  5       raise dopo_il_15;
  6    else
  7       dbms_output.put_line('Prima del 16');
  8    end if;
  9  exception
 10   when dopo_il_15 then
 11    dbms_output.put_line('Errore: dopo il 15 del mese!');
 12  end;
 13  /
Errore: dopo il 15 del mese!

Procedura PL/SQL completata correttamente.
```

Se il programma gira prima del 16 del mese il risultato sarà

```
WTO >declare
  2    dopo_il_15 exception;
  3  begin
  4    if extract(day from systimestamp)>10 then
  5       raise dopo_il_15;
  6    else
  7       dbms_output.put_line('Prima del 16');
  8    end if;
  9  exception
 10   when dopo_il_15 then
 11    dbms_output.put_line('Errore: dopo il 15 del mese!');
 12  end;
 13  /
Prima del 16
```

```
Procedura PL/SQL completata correttamente.
```

Un’eccezione definita dall’utente può essere sollevata esplicitamente mediante il comando RAISE, come appena mostrato, oppure può essere associata ad un errore Oracle in modo da essere sollevata automaticamente da Oracle quando quell’errore si verifica. Tale associazione si realizza mediante l’utilizzo della PRAGMA EXCEPTION_INIT che sarà trattata più avanti, nella parte “avanzata” di questo capitolo, insieme alle altre direttive di compilazione.

8.6.6 Eccezioni mute

Frequentemente ci si imbatte in programmi PL/SQL che ammutoliscono alcune o addirittura tutte le eccezioni. Ammutolire un’eccezione significa gestirla senza dare alcun riscontro al chiamante utilizzando il comando NULL. Nell’esempio seguente, qualunque errore si verifichi, il programma terminerà sempre con successo e senza segnalazioni di alcun tipo.

```
WTO >declare
  2  a number(1);
  3  begin
  4    --divisione per zero
  5    a:=1/0;
  6    --nessun dato trovato
  7    select cod_cliente into a
  8      from clienti
  9     where nome='EUSTACCHIO';
 10    --troppi record estratti
 11    select cod_cliente into a
 12      from clienti;
 13    --valore non valido
 14    a:=1000;
 15  exception
 16    when others then
 17      null;
 18  end;
 19  /

Procedura PL/SQL completata correttamente.
```

La clausola WHEN OTHERS THEN NULL mette a tacere qualunque errore, rendendo impossibile (in casi più complessi di quello presentato) l’analisi di ciò che è successo nel programma.

8.6.7 Eccezioni in un ciclo

Se all’interno di un ciclo si verifica un’eccezione il controllo passa al gestore delle eccezioni. Se il blocco EXCEPTION si trova all’esterno del ciclo il controllo esce dal ciclo e non rientrerà mai più, saltando le elaborazioni residue.

Nell’esempio seguente Un FOR che dovrebbe stampare i numeri da uno a venti viene interrotto da un’eccezione che si verifica alla settima iterazione. I numeri residui non vengono stampati.

```
WTO >declare
  2    a number(3,2);
```

```
  3  begin
  4    for i in 1..20 loop
  5      dbms_output.put_line('I='||i);
  6      a:=1/(i-7);
  7      dbms_output.put_line('A='||a);
  8    end loop;
  9  exception
 10    when zero_divide then
 11      dbms_output.put_line('Divisione per zero!');
 12  end;
 13  /
I=1
A=-,17
I=2
A=-,2
I=3
A=-,25
I=4
A=-,33
I=5
A=-,5
I=6
A=-1
I=7
Divisione per zero!

Procedura PL/SQL completata correttamente.
```

Per ovviare a questo problema e continuare nell'iterazione dopo l'errore è necessario gestire l'eccezione all'interno del LOOP.

È possibile inserire nel ciclo un nuovo blocco BEGIN … EXCEPTION ... END.

```
WTO >declare
  2    a number(3,2);
  3  begin
  4    for i in 1..20 loop
  5      Begin
  6        dbms_output.put_line('I='||i);
  7        a:=1/(i-7);
  8        dbms_output.put_line('A='||a);
  9      exception
 10        when zero_divide then
 11          dbms_output.put_line('Divisione per zero!');
 12      end;
 13    end loop;
 14  end;
 15  /
I=1
A=-,17
I=2
A=-,2
I=3
A=-,25
I=4
A=-,33
I=5
A=-,5
I=6
A=-1
I=7
```

```
Divisione per zero!
I=8
A=1
I=9
A=,5
I=10
A=,33
I=11
A=,25
I=12
A=,2
I=13
A=,17
I=14
A=,14
I=15
A=,13
I=16
A=,11
I=17
A=,1
I=18
A=,09
I=19
A=,08
I=20
A=,08

Procedura PL/SQL completata correttamente.
```

Alla settima iterazione il programma divide per zero, il controllo passa al gestore dell'eccezione WHEN ZERO_DIVIDE e stampa "Divisione per zero!". Poi il blocco finisce ma siamo ancora dentro il ciclo che continua fino alla fine.

8.6.8 Propagazione delle eccezioni

Come evidenziato dall'esempio precedente le eccezioni vengono gestite nel blocco EXCEPTION più vicino all'errore e, per default, non si propagano ai blocchi exception più esterni.

Nell'esempio seguente si verifica un NO_DATA_FOUND. L'eccezione è gestita nel gestore delle eccezioni del blocco interno e non arriva al gestore delle eccezioni del blocco esterno.

```
WTO >declare
  2  a number;
  3  begin
  4    begin
  5      select 1 into a
  6        from dual where 1=2;
  7      dbms_output.put_line('Dopo l''errore');
  8    exception
  9      when no_data_found then
 10        dbms_output.put_line('Gestore interno');
 11    end;
 12    dbms_output.put_line('Siamo nel blocco esterno');
 13  exception
 14    when no_data_found then
 15      dbms_output.put_line('Gestore esterno');
```

```
 16  end;
 17  /
Gestore interno
Siamo nel blocco esterno

Procedura PL/SQL completata correttamente.
```

Se invece volessimo che le eccezioni siano gestite in entrambi i gestori potremmo propagarle con il comando RAISE senza specificare l'eccezione da sollevare.

```
WTO >declare
  2  a number;
  3  begin
  4    begin
  5      select 1 into a
  6        from dual where 1=2;
  7      dbms_output.put_line('Dopo l''errore');
  8    exception
  9      when no_data_found then
 10        dbms_output.put_line('Gestore interno');
 11        raise;
 12    end;
 13    dbms_output.put_line('Siamo nel blocco esterno');
 14  exception
 15    when no_data_found then
 16      dbms_output.put_line('Gestore esterno');
 17  end;
 18  /
Gestore interno
Gestore esterno

Procedura PL/SQL completata correttamente.
```

Il comando RAISE, all'interno di un gestore delle eccezioni e senza specifica di un'eccezione da gestire, risolleva l'eccezione corrente causando la propagazione al gestore esterno.

8.7 Tipi di dato complessi

8.7.1 Record

Un record è un tipo di dati complesso costituito da un numero determinato di elementi. Una variabile di tipo record è particolarmente indicata per contenere i dati delle diverse colonne di una riga estratta dal database con una SELECT.

La dichiarazione di un tipo di dati complesso record segue la seguente sintassi.

```
TYPE <nome tipo> IS RECORD (
<nome colonna1> <tipo colonna1>,
<nome colonna2> <tipo colonna2>,
...
<nome colonnaN> <tipo colonnaN>
)
```

Dopo questa definizione è possibile dichiarare una variabile di tipo <nome tipo>.

Riprendiamo un esempio utilizzato in precedenza per mostrare il funzionamento della SELECT..INTO.

```
WTO >declare
  2    num_f number;
  3    max_f number;
  4    avg_f number;
  5  begin
  6    select count(*), max(importo), avg(importo)
  7      into num_f, max_f, avg_f
  8      from fatture;
  9
 10    dbms_output.put_line('Ci sono '||num_f||' fatture.');
 11    dbms_output.put_line('La più alta è di '||max_f||' euro');
 12    dbms_output.put_line('La media è di '||avg_f||' euro');
 13
 14  end;
 15  /
Ci sono 5 fatture.
La più alta è di 1000 euro
La media è di 600 euro

Procedura PL/SQL completata correttamente.
```

Invece di dichiarare tre variabili indipendenti si sarebbe potuto dichiarare un nuovo tipo di dati complesso: un record costituito da tre dati numerici.

```
WTO >declare
  2    type t_numeri is record (
  3     num_f number,
  4     max_f number,
  5     avg_f number);
  6    r t_numeri;
  7  begin
  8    select count(*), max(importo), avg(importo)
  9      into r
 10      from fatture;
 11
 12    dbms_output.put_line('Ci sono '||r.num_f||' fatture.');
 13  dbms_output.put_line('La più alta è di '||r.max_f||' euro');
 14    dbms_output.put_line('La media è di '||r.avg_f||' euro');
 15
 16  end;
 17  /
Ci sono 5 fatture.
La più alta è di 1000 euro
La media è di 600 euro

Procedura PL/SQL completata correttamente.
```

Dopo la dichiarazione del tipo T_NUMERI si può dichiarare una variabile di quel tipo.

A questo punto la SELECT..INTO si semplifica perché invece di specificare le singole variabili è sufficiente specificare il nome del record. Il resto del programma è quasi invariato, basta aggiungere 'R.' prima dei nomi delle singole variabili che compongono il record.

La dichiarazione di un record strutturalmente identico al record di una tabella è molto più semplice. È sufficiente utilizzare come tipo il <nome

tabella>%ROWTYPE, risparmiando così di indicare tutte le singole colonne ed i relativi tipi.

Nell'esempio seguente si leggono tutte le colonne della tabella CLIENTI per una specifica riga e poi si stampano solo due campi.

```
WTO >declare
  2    r clienti%rowtype;
  3  begin
  4    select *
  5      into r
  6      from clienti
  7     where cod_cliente=3;
  8
  9    dbms_output.put_line('Nome='||r.nome);
 10    dbms_output.put_line('Cognome='||r.cognome);
 11
 12  end;
 13  /
Nome=MATTEO
Cognome=VERDI

Procedura PL/SQL completata correttamente.
```

L'attributo %ROWTYPE può essere utilizzato anche per ottenere un record identico al risultato di un cursore.

Nell'esempio seguente si dichiarano un cursore ed un record strutturalmente identico al record estratto dal cursore. Successivamente il cursore viene aperto, viene eseguita la FETCH della prima riga nella variabile di tipo record e vengono stampati i tre campi che compongono la variabile.

```
WTO >declare
  2    cursor c is
  3      select nome, cognome, cod_fisc from clienti;
  4    r c%rowtype;
  5  begin
  6    open c;
  7    fetch c into r;
  8    dbms_output.put_line('Nome='||r.nome);
  9    dbms_output.put_line('Cognome='||r.cognome);
 10    dbms_output.put_line('Cod fisc='||r.cod_fisc);
 11    close c;
 12  end;
 13  /
Nome=MARCO
Cognome=ROSSI
Cod fisc=RSSMRC70R20H501X

Procedura PL/SQL completata correttamente.
```

Ancora più semplice, come abbiamo già visto, è la gestione del CURSOR FOR LOOP, dove la variabile di tipo RECORD viene implicitamente dichiarata da Oracle.

```
WTO >declare
  2    cursor c is
  3     select nome, cognome, comune from clienti;
  4  begin
  5   for r in c loop
  6    dbms_output.put_line('Letta la riga '||c%rowcount);
```

```
  7    if r.comune in ('H501','F839') then
  8     dbms_output.put_line('Cliente '||r.nome||' '||r.cognome);
  9    end if;
 10   end loop;
 11  end;
 12  /
Letta la riga 1
Cliente MARCO ROSSI
Letta la riga 2
Cliente GIOVANNI BIANCHI
Letta la riga 3
Letta la riga 4
Letta la riga 5
Letta la riga 6
Cliente GENNARO ESPOSITO
Letta la riga 7
Cliente PASQUALE RUSSO
Letta la riga 8

Procedura PL/SQL completata correttamente.
```

8.7.2 Array associativi (PL/SQL Tables)

Un array associativo è un tipo di dati che consente di gestire una lista di coppie (chiave, valore). Gli array associativi erano inizialmente chiamati PL/SQL Tables e sono ancora spesso riconosciuti con questo nome.

La chiave di un array associativo può essere un numero intero oppure una stringa, Oracle conserva gli elementi nell'array associativo in ordine di chiave, non in ordine di inserimento.

Nell'esempio seguente si definisce, si popola e poi si stampa un array associativo avente come chiave il codice cliente e come valore il cognome del cliente.

```
WTO >declare
  2    type t_clienti is table of varchar2(30)
  3      index by binary_integer;
  4    cli t_clienti;
  5    i number;
  6  begin
  7    for r in (select cod_cliente, nome from clienti) loop
  8        cli(r.cod_cliente) := r.nome;
  9    end loop;
 10
 11    i := cli.first;
 12    loop
 13      dbms_output.put_line('Il cliente '||i||' è '||cli(i));
 14      exit when i=cli.last;
 15      i:=cli.next(i);
 16    end loop;
 17  end;
 18  /
Il cliente 1 è MARCO
Il cliente 2 è GIOVANNI
Il cliente 3 è MATTEO
Il cliente 4 è LUCA
Il cliente 5 è AMBROGIO
Il cliente 6 è GENNARO
Il cliente 7 è PASQUALE
```

```
Il cliente 8 è VINCENZO

Procedura PL/SQL completata correttamente.
```

La dichiarazione del tipo

```
type t_clienti is table of varchar2(30)
  index by binary_integer;
```

Indica che il tipo T_CLIENTI è una lista di coppie (chiave, valore) in cui la chiave è un intero ed il valore è un VARCHAR2(30).

Il tipo BINARY_INTEGER è un tipo predefinito di PL/SQL.

Successivamente il tipo T_CLIENTI è utilizzato per dichiarare una variabile di tipo lista, inizialmente vuota.

La sintassi utilizzata per il CURSOR FOR LOOP evita al programmatore di esplicitare la dichiarazione del cursore. Il cursore è implicitamente definito nel FOR scrivendone la SELECT tra parentesi.

Il popolamento della lista CLI è fatto utilizzando un LOOP sugli elementi estratti dalla query. Per ogni record si definisce un nuovo elemento della lista avente come indice R.COD_CLIENTE e come valore R.NOME.

La stampa di tutti i valori dell'array si effettua mediante un ciclo che fa uso di tre funzioni di base che tutti gli array associativi mettono a disposizione.

CLI.FIRST restituisce la chiave di valore più basso presente nella lista.

CLI.NEXT(I) restituisce la chiave successiva a quella che riceve in input.

CLI.LAST restituisce la chiave di valore più alto presente nella lista.

Il valore di un array associativo può essere un record. In tal caso l'array è una vera e propria tabella in memoria.

Nell'esempio seguente si ripete quanto fatto nell'ultimo esempio ma si mette nell'array l'intero contenuto della tabella clienti.

```
WTO >declare
  2     type t_clienti is table of clienti%rowtype
  3       index by binary_integer;
  4     cli t_clienti;
  5     i number;
  6   begin
  7    for r in (select * from clienti) loop
  8        cli(r.cod_cliente) := r;
  9    end loop;
 10
 11    i := cli.first;
 12    loop
 13    dbms_output.put_line('Il cliente '||i||' è '||cli(i).nome);
 14    dbms_output.put_line('il suo cf è '||cli(i).cod_fisc);
 15    dbms_output.put_line('---------------------------------');
 16    exit when i=cli.last;
 17    i:=cli.next(i);
 18    end loop;
 19   end;
```

```
Il cliente 1 è MARCO
il suo cf è RSSMRC70R20H501X
----------------------------------
Il cliente 2 è GIOVANNI
il suo cf è
----------------------------------
Il cliente 3 è MATTEO
il suo cf è VRDMTT69S02H501X
----------------------------------
Il cliente 4 è LUCA
il suo cf è NRILCU77A22F205X
----------------------------------
Il cliente 5 è AMBROGIO
il suo cf è
----------------------------------
Il cliente 6 è GENNARO
il suo cf è SPSGNN71B10F839X
----------------------------------
Il cliente 7 è PASQUALE
il suo cf è RSSPSQ70C14F839X
----------------------------------
Il cliente 8 è VINCENZO
il suo cf è
----------------------------------

Procedura PL/SQL completata correttamente.
```

Rispetto all'esempio precedente le differenze più significative sono nella valorizzazione dell'array e nella stampa.

Nella prima si utilizza l'istruzione

```
cli(r.cod_cliente) := r;
```

dove entrambi i membri dell'assegnazione sono record.

Nella stampa si utilizza l'istruzione

```
dbms_output.put_line('Il cliente '||i||' è '||cli(i).nome);
```

Visto che CLI(I) è un record di tipo CLIENTI%ROWTIPE, esso è composto da tutti i campi di cui è composta la tabella CLIENTI. Per accedere ad ognuno di questi campi si utilizza CLI(I).<nome campo>.

Un array associativo può avere come valori altri array associativi, realizzando in questo modo una matrice avente sia sulle righe che sulle colonne un numero indefinito di elementi.

Ipotizziamo di voler definire una variabile di tipo matrice in cui in una dimensione ci siano le città, nell'altra gli anni e nell'intersezione delle due dimensioni il numero di nuovi residenti in una specifica città in uno specifico anno.

I dati da gestire (inventati di sana pianta) sono quelli rappresentati di seguito.

ANNO CITTA	2000	2001	2002	2003	2004	…
Roma	13.000	11.300	12.500	14.400	16.000	
Napoli	5.400	6.500	6.200	5.800	6.100	
Milano	11.300	13.100	12.600	15.700	14.700	
Firenze	2.350	2.680	2.200	2.920	2.810	
Bologna	1.350	1.470	1.840	1.930	1.150	
…						

Nel programma si dichiara prima un array associativo avente come chiave una stringa (il nome della città) e come valore un numero (il numero di nuovi residenti).

Successivamente si dichiara un array associativo avente come chiave un intero (l'anno) e come valore l'array associativo precedente.

A questo punto utilizzando un doppio indice è possibile specificare il numero di nuovi residenti in una determinata città per un determinato anno.

```
WTO >declare
  2    type t_citta is table of number(8)
  3     index by varchar2(30);
  4    type t_anni is table of t_citta
  5     index by binary_integer;
  6    res t_anni;
  7  begin
  8    res(2000)('ROMA') := 13000;
  9    res(2001)('ROMA') := 11300;
 10    -- tutti gli altri elementi
 11    -- omessi per brevità
 12    -- stampiamo solo i nuovi residenti di Roma nel 2000
 13    dbms_output.put_line('Roma nel 2000:'||res(2000)('ROMA'));
 14  end;
 15  /
Roma nel 2000:13000

Procedura PL/SQL completata correttamente.
```

Il ragionamento appena fatto per definire una matrice bidimensionale è estendibile per aggiungere altre dimensioni. Si può, ad esempio, definire una matrice tridimensionale con un array associativo avente come valore l'array associativo T_ANNI dell'esempio precedente e così via.

8.7.3 Varray

Un varray, o array a dimensione variabile, è una lista di coppie (chiave, valore) avente un numero di elementi massimo definito ed un indice sempre numerico.

Il massimo numero di coppie presenti nell'array si definisce in fase di dichiarazione utilizzando la sintassi seguente:

```
TYPE <nome tipo> is VARRAY(<dimensione massima>) of <tipo>
```

Nell'esempio seguente si definisce un VARRAY chiamato T_GIORNI destinato a contenere il numero di giorni di ogni singolo mese dell'anno.

```
WTO >declare
  2  type t_giorni is varray(12) of number(2);
  3  g t_giorni := t_giorni(31,28,31,30,31,30,31,31,30,31,30,31);
  4  begin
  5   dbms_output.put_line('Febbraio:'||g(2));
  6   g(2) := 29;
  7   for i in g.first..g.last loop
  8    dbms_output.put_line('Mese '||i||' giorni '||g(i));
  9   end loop;
 10  end;
 11  /
Febbraio:28
Mese 1 giorni 31
Mese 2 giorni 29
Mese 3 giorni 31
Mese 4 giorni 30
Mese 5 giorni 31
Mese 6 giorni 30
Mese 7 giorni 31
Mese 8 giorni 31
Mese 9 giorni 30
Mese 10 giorni 31
Mese 11 giorni 30
Mese 12 giorni 31

Procedura PL/SQL completata correttamente.
```

La dichiarazione del tipo indica che i VARRAY di tipo T_GIORNO devono avere al massimo 12 elementi di tipo NUMBER(2).

La dichiarazione del VARRAY G include l'inizializzazione del VARRAY. Usando una funzione che ha lo stesso nome del tipo sono valorizzati tutti i dodici elementi nel VARRAY.

Nel corpo del programma prima si stampa il valore dei giorni di febbraio, poi questo valore viene modificato ed infine con un LOOP di tipo FOR si stampano tutti gli elementi del VARRAY.

Rispetto agli array associativi, i varray hanno le seguenti caratteristiche:

- Hanno un numero massimo di elementi prefissato.
- Hanno per forza indice numerico.
- Devono obbligatoriamente essere inizializzati.
- Possono essere utilizzati come colonna in una tabella di database.

L'ultima caratteristica è, di fatto, l'unica che può far decidere di utilizzare un VARRAY al posto di un array associativo.

8.7.4 Nested Table

Una NESTED TABLE (tabella innestata) è una lista di valori non ordinati di un determinato tipo.

Per definire una Nested Table si utilizza la sintassi

```
TYPE <nome tipo> is TABLE of <tipo>
```

Nell'esempio seguente si definisce una nested table destinata a contenere una lista di nomi di città.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO');
  4  begin
  5   for i in c.first..c.last loop
  6     dbms_output.put_line('Città '||i||': '||c(i));
  7   end loop;
  8   dbms_output.put_line('Aggiungo un elemento');
  9   c.extend;
 10   c(c.count) := 'FIRENZE';
 11   for i in c.first..c.last loop
 12     dbms_output.put_line('Città '||i||': '||c(i));
 13   end loop;
 14  end;
 15  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Aggiungo un elemento
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE

Procedura PL/SQL completata correttamente.
```

Dopo alla dichiarazione ed inizializzazione della nested table se ne stampano tutti gli elementi con un ciclo FOR.

Successivamente la nested table viene estesa di un elemento con il metodo EXTEND. Tale metodo aggiunge un elemento vuoto in ultima posizione. Per valorizzare l'elemento nuovo appena aggiunto si utilizza la solita sintassi di assegnazione, l'unica novità è che l'indice del nuovo elemento è determinato dal metodo COUNT che restituisce il numero totale di elementi presenti nella nested table.

Rispetto agli array associativi, le nested table hanno le seguenti caratteristiche:

- Hanno per forza indice numerico.
- Devono obbligatoriamente essere inizializzate.
- Possono essere utilizzati come colonna in una tabella di database.

Anche in questo caso, l'ultima caratteristica è, di fatto, l'unica che può far decidere di utilizzare una nested table al posto di un array associativo.

8.7.5 Bulk collect

Array associativi, Varray e Nested tables prendono il nome generico di COLLECTION.

C'è un modo per semplificare il popolamento di una COLLECTION sulla base dei dati estratti con una query. Anziché utilizzare un ciclo, come fatto negli esempi precedenti, è possibile usare la clausola BULK COLLECT della SELECT..INTO.

Riprendiamo uno dei primi esempi e modifichiamolo con la clausola BULK COLLECT

```
WTO >declare
  2    type t_clienti is table of clienti%rowtype
  3      index by binary_integer;
  4    cli t_clienti;
  5    i number;
  6  begin
  7   select *
  8   bulk collect into cli
  9   from clienti;
 10
 11   i := cli.first;
 12   loop
 13   dbms_output.put_line('Il cliente '||i||' è '||cli(i).nome);
 14   dbms_output.put_line('il suo cf è '||cli(i).cod_fisc);
 15   dbms_output.put_line('----------------------------------');
 16     exit when i=cli.last;
 17     i:=cli.next(i);
 18   end loop;
 19  end;
 20  /
Il cliente 1 è MARCO
il suo cf è RSSMRC70R20H501X
----------------------------------
Il cliente 2 è GIOVANNI
il suo cf è
----------------------------------
Il cliente 3 è MATTEO
il suo cf è VRDMTT69S02H501X
----------------------------------
Il cliente 4 è LUCA
il suo cf è NRILCU77A22F205X
----------------------------------
Il cliente 5 è AMBROGIO
il suo cf è
----------------------------------
Il cliente 6 è GENNARO
il suo cf è SPSGNN71B10F839X
----------------------------------
Il cliente 7 è PASQUALE
il suo cf è RSSPSQ70C14F839X
----------------------------------
Il cliente 8 è VINCENZO
il suo cf è
----------------------------------

Procedura PL/SQL completata correttamente.
```

La sintassi è semplicissima, si tratta di aggiungere la clausola BULK COLLECT prima della INTO e fornire, ovviamente, una o più collection in cui inserire i dati.

8.7.6 Metodi delle collezioni

Tutte le collection condividono alcuni metodi utili alla loro manipolazione. Alcuni sono già stati utilizzati negli esempi, Nel seguito vengono elencati tutti per completezza.

DELETE

Riceve in input un indice oppure una coppia di indici (valore minimo, valore massimo) e cancella l'elemento della collection avente quell'indice (o tutti gli elementi nell'intervallo). A causa della cancellazione la collection potrebbe avere degli indici mancanti. Nel seguente esempio si dichiara una nested table contenente quattro elementi. La collection viene letta mediante un ciclo FOR. Successivamente viene eliminato il terzo elemento. A questo punto nella collection c'è un buco che causa un errore quando si tenta di leggere l'elemento mancante.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4  begin
  5   for i in c.first..c.last loop
  6     dbms_output.put_line('Città '||i||': '||c(i));
  7   end loop;
  8   dbms_output.put_line('Elimino il terzo elemento');
  9   c.delete(3);
 10   for i in c.first..c.last loop
 11     dbms_output.put_line('Città '||i||': '||c(i));
 12   end loop;
 13  end;
 14  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Elimino il terzo elemento
Città 1: ROMA
Città 2: NAPOLI
declare
*
ERRORE alla riga 1:
ORA-01403: no data found
ORA-06512: at line 11
```

Per evitare il problema la collection deve essere letta con un LOOP standard utilizzando il metodo NEXT.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
```

```
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Elimino il terzo elemento');
 13   c.delete(3);
 14   i := c.first;
 15   loop
 16     dbms_output.put_line('Città '||i||': '||c(i));
 17     exit when i = c.last;
 18     i := c.next(i);
 19   end loop;
 20  end;
 21  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Elimino il terzo elemento
Città 1: ROMA
Città 2: NAPOLI
Città 4: FIRENZE

Procedura PL/SQL completata correttamente.
```

Nell’esempio seguente vengono eliminati gli elementi da uno a tre.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Elimino da uno a tre.');
 13   c.delete(1,3);
 14   i := c.first;
 15   loop
 16     dbms_output.put_line('Città '||i||': '||c(i));
 17     exit when i = c.last;
 18     i := c.next(i);
 19   end loop;
 20  end;
 21  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Elimino da uno a tre.
Città 4: FIRENZE

Procedura PL/SQL completata correttamente.
```

TRIM

Riceve in input un numero intero ed elimina dalla coda della collection un numero di elementi pari al numero ricevuto in input.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
```

```
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Elimino gli ultimi due elementi.');
 13   c.trim(2);
 14   i := c.first;
 15   loop
 16     dbms_output.put_line('Città '||i||': '||c(i));
 17     exit when i = c.last;
 18     i := c.next(i);
 19   end loop;
 20  end;
 21  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Elimino gli ultimi due elementi.
Città 1: ROMA
Città 2: NAPOLI

Procedura PL/SQL completata correttamente.
```

Se il metodo TRIM è chiamato senza parametri elimina solo l'ultimo elemento.

EXTEND

Il metodo EXTEND aggiunge un elemento in coda alla collection.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Aggiungo un elemento vuoto');
 13   c.extend;
 14   i := c.first;
 15   loop
 16     dbms_output.put_line('Città '||i||': '||c(i));
 17     exit when i = c.last;
 18     i := c.next(i);
 19   end loop;
 20  end;
 21  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Aggiungo un elemento vuoto
Città 1: ROMA
```

```
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Città 5:

Procedura PL/SQL completata correttamente.
```

Passando un parametro numerico alla funzione si aggiungono più elementi.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Aggiungo tre elementi vuoti.');
 13   c.extend(3);
 14   i := c.first;
 15   loop
 16     dbms_output.put_line('Città '||i||': '||c(i));
 17     exit when i = c.last;
 18     i := c.next(i);
 19   end loop;
 20  end;
 21  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Aggiungo tre elementi vuoti.
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Città 5:
Città 6:
Città 7:

Procedura PL/SQL completata correttamente.
```

EXISTS

Ritorna TRUE o FALSE in funzione del fatto che un elemento con un determinato indice esiste o meno nella collezione.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Esiste l''elemento 3?');
```

```
 13   if c.exists(3) then
 14      dbms_output.put_line('Sì, esiste.');
 15   else
 16      dbms_output.put_line('No, non esiste.');
 17   end if;
 18  dbms_output.put_line('Esiste l''elemento 5?');
 19   if c.exists(5) then
 20      dbms_output.put_line('Sì, esiste.');
 21   else
 22      dbms_output.put_line('No, non esiste.');
 23   end if;
 24  end;
 25  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Esiste l'elemento 3?
Sì, esiste.
Esiste l'elemento 5?
No, non esiste.

Procedura PL/SQL completata correttamente.
```

FIRST

Il metodo FIRST ritorna il primo indice della collezione.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Primo indice='||c.first);
 13  end;
 14  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Primo indice=1

Procedura PL/SQL completata correttamente.
```

LAST

Il metodo LAST ritorna l’ultimo indice della collezione.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
```

```
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Ultimo indice='||c.last);
 13  end;
 14  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Ultimo indice=4

Procedura PL/SQL completata correttamente.
```

COUNT

Il metodo COUNT ritorna il numero totale di elementi presenti nella collezione.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Numero di elementi='||c.count);
 13  end;
 14  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Numero di elementi=4

Procedura PL/SQL completata correttamente.
```

LIMIT

Il metodo LIMIT restituisce il numero massimo di elementi che la collezione può contenere. Restituisce NULL se la collezione non ha un numero massimo di elementi. In un array associativo ed in una nested table sarà sempre NULL.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Numero massimo di elementi='||c.limit);
 13  end;
 14  /
Città 1: ROMA
```

```
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Numero massimo di elementi=

Procedura PL/SQL completata correttamente.
```

In un varray sarà valorizzato.

```
WTO >declare
  2  type t_giorni is varray(12) of number(2);
  3  g t_giorni := t_giorni(31,28,31,30,31,30,31,31,30,31,30,31);
  4  begin
  5   dbms_output.put_line('Numero di elementi='||g.count);
  6   dbms_output.put_line('Numero max di elementi='||g.limit);
  7   g.trim(5);
  8   dbms_output.put_line('Numero di elementi='||g.count);
  9   dbms_output.put_line('Numero max di elementi='||g.limit);
 10  end;
 11  /
Numero di elementi=12
Numero max di elementi=12
Numero di elementi=7
Numero max di elementi=12

Procedura PL/SQL completata correttamente.
```

PRIOR

Il metodo PRIOR restituisce l'indice precedente rispetto a quello che riceve in input.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Precedente di 3='||c.prior(3));
 13   c.delete(2);
 14   dbms_output.put_line('Precedente di 3='||c.prior(3));
 15  end;
 16  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Precedente di 3=2
Precedente di 3=1

Procedura PL/SQL completata correttamente.
```

NEXT

Il metodo NEXT restituisce l'indice successivo rispetto a quello ricevuto in input.

```
WTO >declare
  2   type t_citta is table of varchar2(30);
```

```
  3   c t_citta := t_citta('ROMA','NAPOLI','MILANO', 'FIRENZE');
  4   i number;
  5  begin
  6   i := c.first;
  7   loop
  8     dbms_output.put_line('Città '||i||': '||c(i));
  9     exit when i = c.last;
 10     i := c.next(i);
 11   end loop;
 12   dbms_output.put_line('Successivo di 2='||c.next(2));
 13  end;
 14  /
Città 1: ROMA
Città 2: NAPOLI
Città 3: MILANO
Città 4: FIRENZE
Successivo di 2=3

Procedura PL/SQL completata correttamente.
```

8.8 Oggetti PL/SQL nel DB

Fino a questo punto si sono visti solo programmi PL/SQL anonimi, scritti direttamente a linea di comando ed eseguiti. Il database in questo caso non memorizza in alcun modo lo script eseguito che deve essere salvato nel file system, se serve, a cura dell'utente. In questo capitolo, invece, vengono trattati i programmi PL/SQL che si salvano nel database con un nome e possono essere eseguiti a richiesta di un utente oppure automaticamente quando si verifica un determinato evento.

8.8.1 Procedure

Una procedura PL/SQL (in inglese *Stored Procedure*) è un programma PL/SQL memorizzato nel database che accetta un determinato numero di parametri di input/output ed esegue un'attività nel database senza ritornare un valore al programma che l'ha chiamato.

La sintassi per la definizione di una procedura è la seguente:

```
CREATE OR REPLACE PROCEDURE <nome procedura>
  (<nome parametro1> <modo parametro1> <tipo parametro1>,
   <nome parametro2> <modo parametro2> <tipo parametro2>,
...
   <nome parametroN> <modo parametroN> <tipo parametroN>) IS
<dichiarazione delle variabili>
BEGIN
  <corpo PL/SQL della procedura>
END;
```

Una procedura non deve obbligatoriamente accettare parametri, nel caso in cui non ci siano parametri la sintassi si riduce a

```
CREATE OR REPLACE PROCEDURE <nome procedura> IS
<dichiarazione delle variabili>
BEGIN
  <corpo PL/SQL della procedura>
END;
```

Per ogni parametro bisogna indicare

- Il nome del parametro
- La modalità di utilizzo del parametro che può essere scelta tra le seguenti:
    - **IN** per i parametri di input alla procedura, questi parametri non sono modificabili all'interno della procedura, sono valorizzati dal programma chiamante e trattati come costanti.
    - **OUT** per i parametri di output, questi non possono essere valorizzati dal programma chiamante e nella procedura possono essere modificati per essere restituiti valorizzati al chiamante.
    - **IN OUT** per i parametri di input/output, questi possono essere sia valorizzati dal chiamante che modificati all'interno della procedura per essere restituiti al chiamante.
- Il tipo di dato del parametro, sono validi tutti i tipi PL/SQL (anche quelli definiti dall'utente). Si specifica solo il tipo di dato e non la lunghezza, per una stringa, ad esempio, si indica VARCHAR2, non VARCHAR(30).

La definizione della procedura fino alla parola chiave IS è detta FIRMA della procedura.

Come primo esempio si mostra una procedura che non accetta alcun parametro e compie, come unica operazione, l'aumento del 10% di tutti gli importi delle fatture.

```
WTO >create or replace procedure aumenta is
  2  begin
  3    update fatture
  4       set importo = importo*1.1;
  5  end;
  6  /

Procedura creata.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S
```

La procedura contiene soltanto un comando SQL di UPDATE che aumenta del 10% l'importo delle fatture. Nell'esempio la procedura è stata creata ma non eseguita. Gli importi sono quelli originali.

Al momento della creazione Oracle esegue un ampio insieme di controlli formali. Innanzitutto verifica che la procedura sia sintatticamente

corretta. Poi verifica che tutti gli oggetti di database utilizzati esistano, siano accessibili e siano fatti esattamente come serve perché la procedura possa essere eseguita correttamente.

Se la procedura contiene un errore di sintassi oppure utilizza in maniera impropria oggetti del database Oracle solleva un errore.

Nell'esempio seguente la stessa procedura appena presentata è creata con errori perché è stato utilizzato un nome di tabella inesistente.

```
WTO >create or replace procedure aumenta is
  2  begin
  3    update fatturA
  4       set importo = importo*1.1;
  5  end;
  6  /

Avvertimento: procedura creata con errori di compilazione.
```

In tal caso il comando SQL*Plus SHOW ERRORS mostra gli errori.

```
WTO >sho errors
Errori in PROCEDURE AUMENTA:

LINE/COL ERROR
-------- -------------------------------------------------
3/3      PL/SQL: SQL Statement ignored
3/10     PL/SQL: ORA-00942: table or view does not exist
```

Il log degli errori letto dal basso verso l'alto mostra che a linea 3 della procedura è stato utilizzata una tabella di database inesistente e per questo motivo l'intera istruzione a linea 3 è stata ignorata.

Corretto l'errore creiamo nuovamente la procedura e cerchiamo di eseguirla.

```
WTO >create or replace procedure aumenta is
  2  begin
  3    update fatture
  4       set importo = importo*1.1;
  5  end;
  6  /

Procedura creata.
```

Una procedura non può essere richiamata all'interno di un comando SQL, ma solo da un altro PL/SQL oppure da linea di comando SQL*Plus.

Una procedura senza parametri può essere eseguita in due modi. Da SQL*Plus si può utilizzare il comando EXEC seguito dal nome della procedura.

```
WTO >exec aumenta

Procedura PL/SQL completata correttamente.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        330 S
```

```
          2          1 01-DIC-10        550 N
          3          2 20-OTT-10        770 S
          4          3 01-FEB-11       1100 N
          5          5 01-DIC-10        550 S
```

Altrimenti si può scrivere un semplice PL/SQL che come unica istruzione richiama la procedura.

```
WTO >begin
  2    aumenta;
  3  end;
  4  /

Procedura PL/SQL completata correttamente.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        363 S
          2          1 01-DIC-10        605 N
          3          2 20-OTT-10        847 S
          4          3 01-FEB-11       1210 N
          5          5 01-DIC-10        605 S
```

La procedura non include il comando di COMMIT, quindi i dati non sono salvati. L'aggiornamento si può annullare con un ROLLBACK.

```
WTO >rollback;

Rollback completato.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S
```

Nel secondo esempio si modifica la procedura precedente in modo che riceva in input la percentuale di aumento dell'importo.

```
WTO >create or replace procedure aumenta (pct in number) is
  2  begin
  3    update fatture
  4       set importo=importo*(1+pct/100);
  5  end;
  6  /

Procedura creata.
```

La procedura riceve in input il parametro PCT di tipo numerico.

Tale parametro è utilizzato nell'UPDATE per come percentuale d'incremento.

Per eseguire la procedura procediamo come prima, ma passiamo il parametro. Aumento del 20%:

```
WTO >exec aumenta(20)
```

```
Procedura PL/SQL completata correttamente.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        360 S
          2          1 01-DIC-10        600 N
          3          2 20-OTT-10        840 S
          4          3 01-FEB-11       1200 N
          5          5 01-DIC-10        600 S
```

Aumento del 10%:

```
WTO >begin
  2    aumenta(10);
  3  end;
  4  /

Procedura PL/SQL completata correttamente.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        396 S
          2          1 01-DIC-10        660 N
          3          2 20-OTT-10        924 S
          4          3 01-FEB-11       1320 N
          5          5 01-DIC-10        660 S
```

Annullo le modifiche.

```
WTO >rollback;

Rollback completato.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S
```

Dopo avere cambiato la procedura, non possiamo più chiamarla senza parametri.

```
WTO >exec aumenta
BEGIN aumenta; END;

      *
ERRORE alla riga 1:
ORA-06550: line 1, column 7:
PLS-00306: wrong number or types of arguments in call to 'AUMENTA'
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
```

C’è, però, la possibilità di assegnare ai parametri un valore di default in modo che il chiamante possa decidere se valorizzarli o no.

```
WTO >create or replace procedure aumenta
  2      (pct in number default 10) is
  3  begin
  4    update fatture
  5       set importo=importo*(1+pct/100);
  6  end;
  7  /

Procedura creata.
```

La procedura è del tutto identica alla precedente con l’eccezione dell’opzione DEFAULT 10. Quest’opzione indica che se il chiamante non passa un valore per il parametro PCT questo deve essere inizializzato al valore 10.

Chiamiamo dunque la procedura senza specificare il parametro.

```
WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S

WTO >exec aumenta

Procedura PL/SQL completata correttamente.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        330 S
          2          1 01-DIC-10        550 N
          3          2 20-OTT-10        770 S
          4          3 01-FEB-11       1100 N
          5          5 01-DIC-10        550 S
```

E dopo chiamiamola passando 20 come percentuale d’aumento

```
WTO >exec aumenta(20)

Procedura PL/SQL completata correttamente.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        396 S
          2          1 01-DIC-10        660 N
          3          2 20-OTT-10        924 S
          4          3 01-FEB-11       1320 N
          5          5 01-DIC-10        660 S

WTO >rollback;
```

```
Rollback completato.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S
```

Complichiamo ancora la procedura aggiungendo un parametro di output. Dopo l'aggiornamento degli importi la procedura deve ritornare in un parametro l'importo della fattura maggiore.

```
WTO >create or replace procedure aumenta
  2     (pct in number default 10,
  3      max_imp out number) is
  4  begin
  5    update fatture
  6       set importo=importo*(1+pct/100);
  7
  8    select max(importo)
  9      into max_imp
 10      from fatture;
 11  end;
 12  /

Procedura creata.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S
```

La procedura, rispetto alla versione precedente, presenta un comando SQL che legge il massimo importo dalle fatture e lo scrive nel parametro di output.

Per lanciare una procedura ci sono sempre due modi. Si può scrivere un semplice PL/SQL in cui si dichiara una variabile per ogni parametro di output. Nella chiamata alla procedura i parametri di input possono essere passati come costanti, mentre per i parametri di output (o di input/output) è necessario specificare una variabile in cui la procedura scriverà il risultato.

```
WTO >set serverout on
WTO >declare
  2    m number;
  3  begin
  4    aumenta(10,m);
  5    dbms_output.put_line('Massimo importo='||m);
  6  end;
  7  /
```

```
Massimo importo=1100

Procedura PL/SQL completata correttamente.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        330 S
          2          1 01-DIC-10        550 N
          3          2 20-OTT-10        770 S
          4          3 01-FEB-11       1100 N
          5          5 01-DIC-10        550 S
```

L'alternativa allo script PL/SQL è, da linea di comando SQL*Plus, dichiarare una variabile ed utilizzarla nel comando EXEC.

```
WTO >var maximp number
WTO >exec aumenta(10,:maximp)

Procedura PL/SQL completata correttamente.

WTO >print maximp

    MAXIMP
----------
      1210

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        363 S
          2          1 01-DIC-10        605 N
          3          2 20-OTT-10        847 S
          4          3 01-FEB-11       1210 N
          5          5 01-DIC-10        605 S
```

SQL*Plus ci offre, infatti, la possibilità di dichiarare delle aree di memoria che restano valide fino a quando si esce da SQL*Plus.

Il comando SQL*Plus VAR dichiara una di queste aree di memoria, nel caso specifico si chiama MAXIMP ed è un numero.

A questo punto è possibile utilizzare il comando EXEC passando come secondo parametro la variabile MAXIMP, si usano i due punti per far capire a SQL*Plus che si tratta di una variabile.

Il comando SQL*Plus PRINT stampa il contenuto della variabile, mostrando che essa è stata correttamente valorizzata con il massimo importo delle fatture.

Ripuliamo i dati prima di andare avanti.

```
WTO >rollback;

Rollback completato.

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
```

```
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        300 S
          2          1 01-DIC-10        500 N
          3          2 20-OTT-10        700 S
          4          3 01-FEB-11       1000 N
          5          5 01-DIC-10        500 S
```

8.8.2 Funzioni

Una funzione PL/SQL si comporta come una procedura con la sola differenza che deve restituire un valore al termine dell'esecuzione. La parola chiave RETURN consente, ad inizio funzione, di dichiarare che tipo di dato sarà ritornato e, nel corpo della funzione, di determinare il valore da restituire.

La sintassi di una funzione si presenta come segue:

```
CREATE OR REPLACE FUNCTION <nome funzione>
  (<nome parametro1> <modo parametro1> <tipo parametro1>,
   <nome parametro2> <modo parametro2> <tipo parametro2>,
...
   <nome parametroN> <modo parametroN> <tipo parametroN>)
  RETURN <tipo> IS
<dichiarazione delle variabili>
BEGIN
  <corpo PL/SQL della funzione>
  RETURN <valore>;
END;
```

Una funzione non deve obbligatoriamente accettare parametri, nel caso in cui non ci siano parametri la sintassi si riduce a

```
CREATE OR REPLACE FUNCTION <nome funzione>
  RETURN <tipo> IS
<dichiarazione delle variabili>
BEGIN
  <corpo PL/SQL della funzione>
  RETURN <valore>;
END;
```

La definizione della funzione fino alla parola chiave IS è detta FIRMA della funzione.

Al momento della creazione della funzione Oracle controlla che sia presente il primo RETURN, che indica quale tipo di dato sarà restituito dalla funzione. Se tale RETURN è assente Oracle solleva un errore di compilazione.

```
WTO >create or replace function test_func is
  2  begin
  3   null;
  4  end;
  5  /

Avvertimento: funzione creata con errori di compilazione.

WTO >sho err
Errori in FUNCTION TEST_FUNC:

LINE/COL ERROR
-------- -------------------------------------------------------
```

```
1/20     PLS-00103: Encountered the symbol "IS" when expecting one of
the following:
         ( return compress compiled wrapped
```

Viceversa Oracle non controlla, in fase di creazione, l'esistenza della clausola RETURN all'interno del corpo della funzione.

```
WTO >create or replace function test_func
  2  return number is
  3  begin
  4   null;
  5  end;
  6  /

Funzione creata.
```

In tal caso l'errore sarà rilevato solo al momento dell'esecuzione della funzione.

```
WTO >select test_func from dual;
select test_func from dual
       *
ERRORE alla riga 1:
ORA-06503: PL/SQL: Function returned without value
ORA-06512: at "WTO_ESEMPIO.TEST_FUNC", line 5
```

La clausola RETURN nel corpo della funzione può essere presente anche più di una volta.

```
WTO >create or replace function test_func (
  2   mypar in number) return number is
  3  begin
  4    if mypar >100 then
  5       return mypar;
  6    else
  7       return mypar*10;
  8    end if;
  9  end;
 10  /

Funzione creata.
```

La funzione TEST_FUNC ritorna il parametro che riceve in input se questo è maggiore di 100 altrimenti ritorna il parametro moltiplicato per dieci. Eseguendola se ne verifica il comportamento.

```
WTO >select test_func(200), test_func(15)
  2  from dual;

TEST_FUNC(200) TEST_FUNC(15)
-------------- -------------
           200           150
```

Sebbene non sia sintatticamente errato, è buona norma di programmazione evitare l'utilizzo di RETURN multiple all'interno di una funzione. L'esempio precedente si può scrivere come segue:

```
WTO >create or replace function test_func (
  2   mypar in number) return number is
  3    ret_val number;
  4  begin
  5    if mypar >100 then
  6       ret_val := mypar;
```

```
  7    else
  8       ret_val := mypar*10;
  9    end if;
 10
 11    return ret_val;
 12  end;
 13  /

Funzione creata.

WTO >select test_func(200), test_func(15)
  2  from dual;

TEST_FUNC(200) TEST_FUNC(15)
-------------- -------------
           200           150
```

Anche una funzione che prevede la clausola RETURN può, per errore, terminare senza valore. Nell'esempio seguente si gestisce l'eccezione VALUE_ERROR che si potrebbe verificare nelle assegnazioni:

```
WTO >create or replace function test_func (
  2   mypar in number) return number is
  3    ret_val number(3);
  4  begin
  5    if mypar >100 then
  6       ret_val := mypar;
  7    else
  8       ret_val := mypar*10;
  9    end if;
 10
 11    return ret_val;
 12  exception
 13    when value_error then
 14     dbms_output.put_line('Errore di valore.');
 15  end;
 16  /

Funzione creata.

WTO >select test_func(200), test_func(15)
  2  from dual;

TEST_FUNC(200) TEST_FUNC(15)
-------------- -------------
           200           150
```

La funzione si comporta come prima, ma se si passa in input un valore a quattro cifre:

```
WTO >set serveroutput on
WTO >select test_func(1000)
  2  from dual;
select test_func(1000)
       *
ERRORE alla riga 1:
ORA-06503: PL/SQL: Function returned without value
ORA-06512: at "WTO_ESEMPIO.TEST_FUNC", line 15


Errore di valore.
```

L'assegnazione a linea 6 ha generato un VALUE_ERROR perché cercava di mettere il valore 1000 in una variabile definita NUMBER(3), il flusso del programma è quindi saltato al blocco EXCEPTION dove era previsto un messaggio d'errore ma non c'è la clausola RETURN. Oracle quindi stampa il messaggio d'errore e l'errore dovuto al fatto che la funzione è finita senza un valore di ritorno.

Per ovviare a questo problema si può ridefinire la funzione in questo modo:

```
WTO >create or replace function test_func (
  2   mypar in number) return number is
  3    ret_val number(3);
  4  begin
  5    if mypar >100 then
  6       ret_val := mypar;
  7    else
  8       ret_val := mypar*10;
  9    end if;
 10
 11    return ret_val;
 12  exception
 13    when value_error then
 14     dbms_output.put_line('Errore di valore.');
 15     return null;
 16  end;
 17  /

Funzione creata.

WTO >select test_func(1000)
  2  from dual;

TEST_FUNC(1000)
---------------


Errore di valore.
```

Oppure, con un'unica RETURN, come segue:

```
WTO >create or replace function test_func (
  2   mypar in number) return number is
  3    ret_val number(3);
  4  begin
  5    begin
  6      if mypar >100 then
  7         ret_val := mypar;
  8      else
  9         ret_val := mypar*10;
 10      end if;
 11    exception
 12      when value_error then
 13        dbms_output.put_line('Errore di valore.');
 14        ret_val := null;
 15    end;
 16
 17    return ret_val;
 18  end;
 19  /

Funzione creata.
```

```
WTO >select test_func(200), test_func(15)
  2  from dual;

TEST_FUNC(200) TEST_FUNC(15)
-------------- -------------
           200           150

WTO >select test_func(1000)
  2  from dual;

TEST_FUNC(1000)
---------------


Errore di valore.
```

Come ampiamente visto negli esempi precedenti il modo più consueto per chiamare una funzione è all'interno di un'istruzione SQL. Per essere utilizzata in SQL, però, una funzione deve verificare alcune caratteristiche di "purezza". Le due caratteristiche principali sono: non deve prevedere parametri in modalità OUT o IN OUT e non deve modificare il database.

La funzione seguente non può essere utilizzata in SQL.

```
WTO >create or replace function test_func
  2  return number is
  3  begin
  4    update fatture
  5      set importo=0
  6     where 1=2;
  7  end;
  8  /

Funzione creata.

WTO >select test_func
  2  from dual;
select test_func
       *
ERRORE alla riga 1:
ORA-14551: cannot perform a DML operation inside a query
ORA-06512: at "WTO_ESEMPIO.TEST_FUNC", line 4
```

L'errore è causato dal fatto che la funzione tenta di eseguire un comando DML.

Se la funzione non rispetta uno dei due criteri appena esposti bisogna richiamarla come se fosse una procedura. Si può utilizzare il comando EXEC dichiarando delle variabili SQL*Plus:

```
WTO >create or replace function test_func (
  2   mypar in number) return number is
  3    ret_val number(3);
  4  begin
  5    if mypar >100 then
  6       ret_val := mypar;
  7    else
  8       ret_val := mypar*10;
  9    end if;
 10
 11    return ret_val;
```

```
 12  exception
 13    when value_error then
 14     dbms_output.put_line('Errore di valore.');
 15     return null;
 16  end;
 17  /

Funzione creata.

WTO >var ret_val number
WTO >
WTO >exec :ret_val := test_func(50)

Procedura PL/SQL completata correttamente.

WTO >print ret_val

   RET_VAL
----------
       500
```

Oppure si può utilizzare un programmino PL/SQL di test.

```
WTO >begin
  2    dbms_output.put_line(test_func(50));
  3  end;
  4  /
500

Procedura PL/SQL completata correttamente.
```

Che è, poiché la funzione non richiede variabili, equivalente al seguente, più semplice:

```
WTO >exec dbms_output.put_line(test_func(50))
500

Procedura PL/SQL completata correttamente.
```

8.8.3 Package

Un package PL/SQL è un oggetto virtuale che racchiude un insieme di procedure e funzioni. Ci sono vari buoni motivi per accorpare le procedure e funzioni in package anziché tenerle slegate tra loro:

- Modularità dell'applicazione: le procedure e funzioni logicamente collegate tra loro sono conservate nello stesso package e viste come un tutt'uno. Ciò agevola un buon disegno architetturale della soluzione consentendo di avere pochi pacchetti di programmi anziché molti programmi sciolti.
- Possibilità di nascondere le funzionalità di servizio: un package è composto di due parti, la specification ed il body. Solo le procedure, funzioni e variabili incluse nella specification possono essere accedute dagli altri programmi PL/SQL presenti nel DB; le procedure, funzioni e variabili presenti solo nel body sono "private" ed accessibili solo dagli altri programmi del package.

- Variabili di sessione: una variabile definita nel package al di fuori di tutte le procedure e funzioni è istanziata la prima volta che il package viene richiamato e resta valida fino a fine sessione. È dunque possibile utilizzare delle variabili che sopravvivono alla fine del programma e possono essere utilizzate da altri programmi che gireranno in seguito nella stessa sessione.

La sintassi per la creazione di un package, come detto, è tale da definire due oggetti separati: la specification ed il body.

Per la specification:

```
CREATE OR REPLACE PACKAGE <nome package> IS
  PROCEDURE/FUNCTION <nome procedura o funzione >
                     <parametri> <eventuale clausola return>;
  PROCEDURE/FUNCTION <nome procedura o funzione >
                     <parametri> <eventuale clausola return>;
..
  PROCEDURE/FUNCTION <nome procedura o funzione >
                     <parametri> <eventuale clausola return>;
<dichiarazione delle variabili di package>
END;
```

Per il body

```
CREATE OR REPLACE PACKAGE BODY <nome package> IS
  <dichiarazione delle variabili private di package>
PROCEDURE/FUNCTION <nome procedura o funzione >
                     <parametri> <eventuale clausola return> IS
  <variabili di procedura o funzione>
  BEGIN
    <corpo della procedura o funzione>
  END;

  PROCEDURE/FUNCTION <nome procedura o funzione >
                     <parametri> <eventuale clausola return> IS
  <variabili di procedura o funzione>
  BEGIN
    <corpo della procedura o funzione>
  END;

  PROCEDURE/FUNCTION <nome procedura o funzione >
                     <parametri> <eventuale clausola return> IS
  <variabili di procedura o funzione>
  BEGIN
    <corpo della procedura o funzione>
  END;
END;
```

Nella specification si dichiarano solo le firme delle procedure o funzioni accessibili dall’esterno del package. Nel body si dettaglia il corpo di tutte le procedure e funzioni: sia di quelle accessibili solo dagli altri programmi del package sia di quelle accessibili anche dagli altri programmi presenti nel DB.

Prima di cominciare con gli esempi eliminiamo la procedura e la funzione precedentemente create.

```
WTO >drop procedure aumenta;

Procedura eliminata.

WTO >drop function test_func;
```

```
Funzione eliminata.
```

Come primo esempio accorpiamo le procedure e funzioni viste finora in un package di nome TEST_PKG.

```
WTO >create or replace package test_pkg is
  2
  3  procedure aumenta (pct in number default 10,
  4                     max_imp out number);
  5
  6  function test_func (mypar in number) return number;
  7
  8  end;
  9  /

Package creato.

WTO >create or replace package body test_pkg is
  2
  3  procedure aumenta (pct in number default 10,
  4                     max_imp out number) is
  5  begin
  6    update fatture
  7       set importo=importo*(1+pct/100);
  8
  9    select max(importo)
 10      into max_imp
 11      from fatture;
 12  end;
 13
 14  function test_func (mypar in number) return number is
 15    ret_val number(3);
 16  begin
 17    if mypar >100 then
 18       ret_val := mypar;
 19    else
 20       ret_val := mypar*10;
 21    end if;
 22
 23    return ret_val;
 24  exception
 25    when value_error then
 26     dbms_output.put_line('Errore di valore.');
 27     return null;
 28  end;
 29
 30  end;
 31  /

Creato package body.
```

I due programmi sono entrambi nella specification, dunque possono essere entrambi richiamati dall’esterno. È tutto perfettamente analogo a quanto visto prima, bisogna solo specificare il nome del package prima del nome della procedura o funzione.

```
WTO >var m number
WTO >exec test_pkg.aumenta(20,:m)
```

```
Procedura PL/SQL completata correttamente.

WTO >print m

         M
----------
      1200

WTO >select * from fatture;

NUM_FATTURA NUM_ORDINE DATA_FATT    IMPORTO P
----------- ---------- --------- ---------- -
          1          1 01-OTT-10        360 S
          2          1 01-DIC-10        600 N
          3          2 20-OTT-10        840 S
          4          3 01-FEB-11       1200 N
          5          5 01-DIC-10        600 S

WTO >select test_pkg.test_func(15) from dual;

TEST_PKG.TEST_FUNC(15)
----------------------
                   150

WTO >rollback;

Rollback completato.
```

Proviamo adesso a rimuovere la funzione dalla specification.

```
WTO >create or replace package test_pkg is
  2
  3  procedure aumenta (pct in number default 10,
  4                     max_imp out number);
  5  end;
  6  /

Package creato.
```

La procedura può essere comunque chiamata mentre la funzione non è più richiamabile dall'esterno del package.

```
WTO >exec test_pkg.aumenta(20,:m)

Procedura PL/SQL completata correttamente.

WTO >print m

         M
----------
      1200

WTO >select test_pkg.test_func(15) from dual;
select test_pkg.test_func(15) from dual
       *
ERRORE alla riga 1:
ORA-00904: "TEST_PKG"."TEST_FUNC": invalid identifier
```

La funzione può invece essere richiamata dall'interno del package, ad esempio nella procedura aumenta.

```
WTO >create or replace package body test_pkg is
  2
```

```
  3  function test_func (mypar in number) return number is
  4    ret_val number(3);
  5  begin
  6    if mypar >100 then
  7       ret_val := mypar;
  8    else
  9       ret_val := mypar*10;
 10    end if;
 11
 12    return ret_val;
 13  exception
 14    when value_error then
 15     dbms_output.put_line('Errore di valore.');
 16     return null;
 17  end;
 18
 19  procedure aumenta (pct in number default 10,
 20                     max_imp out number) is
 21  begin
 22    update fatture
 23       set importo=importo*(1+pct/100);
 24
 25    max_imp := test_func(10);
 26
 27  end;
 28
 29
 30  end;
 31  /

Creato package body.

WTO >exec test_pkg.aumenta(20,:m)

Procedura PL/SQL completata correttamente.

WTO >print m

         M
----------
       100
```

La procedura aumenta è stata modificata in modo da richiamare la funzione TEST_FUNC invece di fare la query del massimo importo da FATTURE. Non è stato necessario specificare il nome del package davanti al nome della funzione visto che la funzione è nello stesso package della procedura che la richiama. È però necessario che la funzione sia definita prima della procedura. Se nel body invertiamo la posizione di procedura e funzione otteniamo un errore.

```
WTO >create or replace package body test_pkg is
  2
  3  procedure aumenta (pct in number default 10,
  4                     max_imp out number) is
  5  begin
  6    update fatture
  7       set importo=importo*(1+pct/100);
  8
  9    max_imp := test_func(10);
 10
 11  end;
```

```
 12
 13  function test_func (mypar in number) return number is
 14    ret_val number(3);
 15  begin
 16    if mypar >100 then
 17       ret_val := mypar;
 18    else
 19       ret_val := mypar*10;
 20    end if;
 21
 22    return ret_val;
 23  exception
 24    when value_error then
 25     dbms_output.put_line('Errore di valore.');
 26     return null;
 27  end;
 28
 29  end;
 30  /

Avvertenza: package body creato con errori di compilazione.

WTO >sho error
Errori in PACKAGE BODY TEST_PKG:

LINE/COL ERROR
-------- --------------------------------------------------------
9/3      PL/SQL: Statement ignored
9/14     PLS-00313: 'TEST_FUNC' not declared in this scope
```

La funzione TEST_FUNC non era stata ancora dichiarata nel momento in cui è stata utilizzata.

Nel prossimo esempio aggiungeremo al package due variabili, una nella specification ed una nel body.

```
WTO >create or replace package body test_pkg is
  2
  3  VAR_BODY number;
  4
  5  function test_func (mypar in number) return number is
  6    ret_val number(3);
  7  begin
  8    if mypar >100 then
  9       ret_val := mypar;
 10    else
 11       ret_val := mypar*10;
 12    end if;
 13
 14    return ret_val;
 15  exception
 16    when value_error then
 17     dbms_output.put_line('Errore di valore.');
 18     return null;
 19  end;
 20
 21  procedure aumenta (pct in number default 10,
 22                     max_imp out number) is
 23  begin
 24    update fatture
 25       set importo=importo*(1+pct/100);
 26
```

```
 27    max_imp := test_func(10);
 28
 29  end;
 30
 31  end;
 32  /

Creato package body.
```

La variabile VAR_SPEC è visibile dall'esterno del package e può essere impostata e letta. Per impostarla si può utilizzare un semplice blocco PL/SQL.

```
WTO >begin
  2    test_pkg.var_spec := 100;
  3  end;
  4  /

Procedura PL/SQL completata correttamente.
```

Per leggerla non si può usare una query ma anche qui un blocco PL/SQL oppure il comando EXEC.

```
WTO >select test_pkg.var_spec
  2  from dual;
select test_pkg.var_spec
       *
ERRORE alla riga 1:
ORA-06553: PLS-221: 'VAR_SPEC' is not a procedure or is undefined


WTO >exec :m := test_pkg.var_spec

Procedura PL/SQL completata correttamente.

WTO >print m

         M
----------
       100
```

La variabile VAR_BODY, invece, non è accessibile se non dall'interno del package.

```
WTO >begin
  2    test_pkg.var_body := 123;
  3  end;
  4  /
  test_pkg.var_body := 123;
           *
ERRORE alla riga 2:
ORA-06550: line 2, column 12:
PLS-00302: component 'VAR_BODY' must be declared
ORA-06550: line 2, column 3:
PL/SQL: Statement ignored
```

Entrambe le variabili possono essere inizializzate la prima volta che il package è chiamato nella sessione e restano vive fino a fine sessione.

Per inizializzare le variabili la prima volta che il package viene richiamato, per qualunque funzione o procedura, si può aggiungere un

blocco di codice PL/SQL prima della END del body all'esterno di tutte le procedure e funzioni.

Nell'esempio seguente si aggiunge il codice d'inizializzazione al package TEST_PKG per inizializzare le due variabili.

```
WTO >create or replace package body test_pkg is
  2
  3  VAR_BODY number;
  4
  5  function test_func (mypar in number) return number is
  6    ret_val number(3);
  7  begin
  8    if mypar >100 then
  9       ret_val := mypar;
 10    else
 11       ret_val := mypar*10;
 12    end if;
 13
 14    return ret_val;
 15  exception
 16    when value_error then
 17     dbms_output.put_line('Errore di valore.');
 18     return null;
 19  end;
 20
 21  procedure aumenta (pct in number default 10,
 22                     max_imp out number) is
 23  begin
 24    update fatture
 25       set importo=importo*(1+pct/100);
 26
 27    max_imp := test_func(10);
 28
 29  end;
 30
 31  --Quello che segue è il codice d'inizializzazione.
 32  begin
 33    var_spec := 1234;
 34    var_body := 4321;
 35  end;
 36  /

Creato package body.
```

A questo punto richiamando una qualunque funzione o procedura del package il codice d'inizializzazione viene eseguito (solo la prima volta che il package è richiamato!) e valorizza le variabili.

```
WTO >exec test_pkg.aumenta(10,:m)

Procedura PL/SQL completata correttamente.

WTO >exec :m := test_pkg.var_spec;

Procedura PL/SQL completata correttamente.

WTO >print m

         M
----------
      1234
```

8.8.4 Database Trigger

Un database trigger è un programma PL/SQL salvato nel database che scatta automaticamente quando si verifica un determinato evento. I trigger si dividono in due categorie:

I **system trigger** sono collegati ad eventi DDL (ad esempio DROP, CREATE, ALTER) oppure ad eventi amministrativi (ad esempio STARTUP, SHUTDOWN) che avvengono in un determinato schema oppure sull'intero database. Sono utilizzati prevalentemente per attività di amministrazione del database e per questo motivo non saranno approfonditi in questo manuale.

I **DML trigger** sono collegati ad eventi di tipo DML (INSERT, UPDATE o DELETE) su una specifica tabella o vista. Nel seguito si approfondiranno solo i trigger di questa categoria.

Come detto, l'evento che può far scattare un trigger è l'esecuzione di un comando DML su una tabella (o vista). Il trigger può essere configurato per scattare prima o dopo l'esecuzione del comando DML.

La sintassi minimale per la definizione di un trigger è

```
CREATE OR REPLACE TRIGGER <nome trigger>
<BEFORE/AFTER> <INSERT/UPDATE/DELETE> ON <nome tabella>
DECLARE
  <eventuali vairabili e costanti>
BEGIN
  <codice PL/SQL del trigger>
END;
```

Ad esempio il trigger seguente scatta all'aggiornamento della tabella CLIENTI e stampa un messaggio.

```
WTO >create or replace trigger TEST_UPD_CLI
  2  AFTER UPDATE ON CLIENTI
  3  begin
  4    dbms_output.put_line('Trigger eseguito!');
  5  end;
  6  /

Trigger creato.

WTO >set serverout on
WTO >update clienti set nome='pippo';
Trigger eseguito!

Aggiornate 8 righe.

WTO >rollback;
Completato rollback.
```

Un trigger di questo tipo è detto STATEMENT trigger, cioè un trigger che viene eseguito una sola volta a fronte del comando DML che lo fa scattare.

È possibile definire un EACH ROW trigger che viene eseguito una volta per ogni record coinvolto nel comando DML che lo fa scattare. Se il trigger è AFTER DELETE e scatta per una DELETE che cancella cento

record, lo statement trigger scatterebbe una sola volta mentre l'EACH ROW trigger scatta cento volte.

Per definire un EACH ROW trigger basta aggiungere la clausola FOR EACH ROW.

```
WTO >create or replace trigger TEST_UPD_CLI
  2  AFTER UPDATE ON CLIENTI
  3  FOR EACH ROW
  4  begin
  5    dbms_output.put_line('Trigger eseguito!');
  6  end;
  7  /
Trigger creato.

WTO >update clienti set nome='pippo';
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!

Aggiornate 8 righe.

WTO >rollback;
Completato rollback.
```

Adesso il trigger scatta una volta per ogni record aggiornato.

TRIGGER CONDIZIONALI

Il trigger può essere condizionale, cioè scattare solo quando si verifica una determinata condizione. Per indicare la condizione si utilizza la clausola WHEN.

Nell'esempio seguente il trigger scatta solo se il nuovo nome del cliente comincia per P.

```
WTO >create or replace trigger TEST_UPD_CLI
  2  AFTER UPDATE ON CLIENTI
  3  FOR EACH ROW
  4  WHEN (new.NOME like 'P%')
  5  begin
  6    dbms_output.put_line('Trigger eseguito!');
  7  end;
  8  /

Trigger creato.
```

```
WTO >update clienti set nome='pippo';

Aggiornate 8 righe.

WTO >update clienti set nome='Pippo';
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!
Trigger eseguito!

Aggiornate 8 righe.

WTO >rollback;

Rollback completato.
```

Nel primo UPDATE il trigger non è scattato perché il nuovo nome del cliente cominciava per p minuscola, nel secondo update invece il nuovo nome del cliente comincia per P maiuscola ed il trigger scatta.

REFERENZIARE LE COLONNE

All’interno del corpo del trigger, in caso FOR EACH ROW, è possibile referenziare i valori delle colonne del record. Per ogni colonna della tabella esistono due valori :OLD.<nome colonna> e :NEW.<nome colonna> che rappresentano il valore di quella colonna prima e dopo l’istruzione DML che fa scattare il trigger. Per capire meglio il funzionamento dei prefissi OLD e NEW si consideri l’esempio seguente.

```
WTO >create or replace trigger TEST_UPD_CLI
  2  AFTER UPDATE ON CLIENTI
  3  FOR EACH ROW
  4  WHEN (new.NOME like 'P%')
  5  begin
  6    dbms_output.put_line('Cliente '||:old.cod_cliente||
  7    ' vecchio nome '||:old.nome||' nuovo '||:new.nome);
  8  end;
  9  /

Trigger creato.

WTO >update clienti set nome='Pippo';
Cliente 1 vecchio nome MARCO nuovo Pippo
Cliente 2 vecchio nome GIOVANNI nuovo Pippo
Cliente 3 vecchio nome MATTEO nuovo Pippo
Cliente 4 vecchio nome LUCA nuovo Pippo
Cliente 5 vecchio nome AMBROGIO nuovo Pippo
Cliente 6 vecchio nome GENNARO nuovo Pippo
Cliente 7 vecchio nome PASQUALE nuovo Pippo
Cliente 8 vecchio nome VINCENZO nuovo Pippo

Aggiornate 8 righe.

WTO >rollback;

Rollback completato.
```

Da notare il fatto che prima di OLD e NEW è richiesto il carattere due punti nel corpo del trigger, ma non nella clausola WHEN.

In un trigger di UPDATE sono disponibili sia i valori OLD che i NEW, in un trigger di INSERT sono disponibili solo i NEW mentre gli OLD sono tutti NULL, viceversa in un trigger di DELETE sono disponibili solo gli OLD mentre i NEW sono tutti NULL.

CLAUSOLA REFERENCING

Nell'ipotesi che nel database esista una tabella di nome OLD oppure NEW è possibile, per evitare fraintendimenti, utilizzare la clausola REFERENCING per sostituire i prefissi OLD e NEW con prefissi a piacere scelti dal programmatore.

```
WTO >create or replace trigger TEST_UPD_CLI
  2  AFTER UPDATE ON CLIENTI
  3  REFERENCING OLD as VECCHIO NEW as NUOVO
  4  FOR EACH ROW
  5  begin
  6    dbms_output.put_line('Cliente '||:vecchio.cod_cliente||
  7    ' vecchio nome '||:vecchio.nome||' nuovo '||:nuovo.nome);
  8  end;
  9  /

Trigger creato.

WTO >update clienti set nome='pippo';
Cliente 1 vecchio nome MARCO nuovo pippo
Cliente 2 vecchio nome GIOVANNI nuovo pippo
Cliente 3 vecchio nome MATTEO nuovo pippo
Cliente 4 vecchio nome LUCA nuovo pippo
Cliente 5 vecchio nome AMBROGIO nuovo pippo
Cliente 6 vecchio nome GENNARO nuovo pippo
Cliente 7 vecchio nome PASQUALE nuovo pippo
Cliente 8 vecchio nome VINCENZO nuovo pippo

Aggiornate 8 righe.

WTO >rollback;
Rollback completato.
```

Nell'esempio precedente si è scelto di utilizzare l'alias NUOVO al posto di NEW e VECCHIO al posto di OLD.

CLAUSOLA UPDATE OF

Solo sui trigger di UPDATE è possibile specificare le colonne che devono essere sotto controllo. Il trigger scatterà solo se viene modificata una di queste colonne.

```
WTO >create or replace trigger TEST_UPD_CLI
  2  AFTER UPDATE OF NOME, COGNOME ON CLIENTI
  3  FOR EACH ROW
  4  begin
  5    dbms_output.put_line('Cliente '||:old.cod_cliente||
  6    ' vecchio nome '||:old.nome||' nuovo '||:new.nome);
  7  end;
  8  /
Trigger creato.

WTO >update clienti set cod_fisc=null;
Aggiornate 8 righe.
```

```
WTO >update clienti set nome='pippo';
Cliente 1 vecchio nome MARCO nuovo pippo
Cliente 2 vecchio nome GIOVANNI nuovo pippo
Cliente 3 vecchio nome MATTEO nuovo pippo
Cliente 4 vecchio nome LUCA nuovo pippo
Cliente 5 vecchio nome AMBROGIO nuovo pippo
Cliente 6 vecchio nome GENNARO nuovo pippo
Cliente 7 vecchio nome PASQUALE nuovo pippo
Cliente 8 vecchio nome VINCENZO nuovo pippo

Aggiornate 8 righe.

WTO >rollback;

Rollback completato.
```

Nell'esempio precedente il trigger scatta solo se si effettua un comando di UPDATE che coinvolge la colonna NOME o COGNOME. L'UPDATE che modifica il codice fiscale non fa scattare il trigger.

EVENTI COMPOSTI

Un trigger può scattare in relazione a più eventi. Il trigger seguente scatta all'inserimento oppure all'aggiornamento del nome sulla tabella CLIENTI.

```
WTO >create or replace trigger TEST_UPD_CLI
  2  AFTER INSERT OR UPDATE OF NOME ON CLIENTI
  3  FOR EACH ROW
  4  begin
  5    dbms_output.put_line('Cliente '||:old.cod_cliente||
  6    ' vecchio nome '||:old.nome||' nuovo '||:new.nome);
  7  end;
  8  /

Trigger creato.

WTO >update clienti set nome='pippo';
Cliente 1 vecchio nome MARCO nuovo pippo
Cliente 2 vecchio nome GIOVANNI nuovo pippo
Cliente 3 vecchio nome MATTEO nuovo pippo
Cliente 4 vecchio nome LUCA nuovo pippo
Cliente 5 vecchio nome AMBROGIO nuovo pippo
Cliente 6 vecchio nome GENNARO nuovo pippo
Cliente 7 vecchio nome PASQUALE nuovo pippo
Cliente 8 vecchio nome VINCENZO nuovo pippo

Aggiornate 8 righe.

WTO >insert into clienti values (
  2  9,'pippo','pluto',null,'indirizzo','00100','H501');
Cliente  vecchio nome  nuovo pippo

Creata 1 riga.

WTO >rollback;

Rollback completato.
```

Nel caso di trigger che scatta in risposta a più eventi è utile sapere nel codice del trigger qual è l'evento corrente. Oracle mette per questo motivo a disposizione i predicati INSERTING, UPDATING e DELETING che restituiscono TRUE o FALSE per indicare quale evento ha fatto scattare il trigger. L'esempio precedente si può modificare come segue:

```
WTO >create or replace trigger TEST_UPD_CLI
  2  AFTER INSERT OR UPDATE OF NOME ON CLIENTI
  3  FOR EACH ROW
  4  begin
  5    IF INSERTING THEN
  6      dbms_output.put_line('Inserimento nuovo cliente');
  7    ELSIF UPDATING THEN
  8      dbms_output.put_line('Agg. cliente '||:old.cod_cliente);
  9    END IF;
 10  end;
 11  /

Trigger creato.

WTO >update clienti
  2  set nome = 'Pippo';
Agg. cliente 1
Agg. cliente 2
Agg. cliente 3
Agg. cliente 4
Agg. cliente 5
Agg. cliente 6
Agg. cliente 7
Agg. cliente 8

Aggiornate 8 righe.

WTO >insert into clienti values (
  2  9,'pippo','pluto',null,'indirizzo','00100','H501');
Inserimento nuovo cliente

Creata 1 riga.

WTO >rollback;

Rollback completato.
```

DISABILITAZIONE ED ABILITAZIONE

Un trigger non può essere eseguito esplicitamente, va in esecuzione solo quando si verifica l'evento a cui è collegato. È però possibile disabilitare un trigger per evitare che scatti. La disabilitazione di un trigger si esegue con il comando

```
WTO >alter trigger TEST_UPD_CLI disable;

Trigger modificato.
```

Dopo la disabilitazione il trigger non scatta più.

```
WTO >update clienti set nome='pippo';

Aggiornate 8 righe.

WTO >rollback;
```

```
Rollback completato.
```

Per riabilitare il trigger si procede come segue:

```
WTO >alter trigger TEST_UPD_CLI enable;

Trigger modificato.
```

Il trigger ricomincia a funzionare.

```
WTO >update clienti set nome='pippo';
Cliente 1 vecchio nome MARCO nuovo pippo
Cliente 2 vecchio nome GIOVANNI nuovo pippo
Cliente 3 vecchio nome MATTEO nuovo pippo
Cliente 4 vecchio nome LUCA nuovo pippo
Cliente 5 vecchio nome AMBROGIO nuovo pippo
Cliente 6 vecchio nome GENNARO nuovo pippo
Cliente 7 vecchio nome PASQUALE nuovo pippo
Cliente 8 vecchio nome VINCENZO nuovo pippo

Aggiornate 8 righe.

WTO >rollback;

Rollback completato.
```

DIMENSIONE DI UN TRIGGER

La massima dimensione del codice di un trigger è 32Kb, è vivamente consigliato, però, non superare le sessanta linee di codice. Se il codice supera le sessanta linee di codice è consigliabile mettere una parte del codice in una stored procedure oppure in una funzione di database.

ORDINE DI ESECUZIONE

Su una tabella possono essere definiti più trigger. In questo caso l'ordine di esecuzione è il seguente:

- Before statement
- Before each row
- After each row
- After statement

Nell'esempio seguente si definiscono quattro trigger di UPDATE sulla tabella COMUNI e si fanno scattare tutti con un'unica istruzione per verificare l'ordine di esecuzione.

```
WTO >create trigger com_b_u_s
  2  before update on comuni
  3  begin
  4    dbms_output.put_line('Before update statement');
  5  end;
  6  /

Trigger creato.

WTO >create trigger com_b_u_e
  2  before update on comuni
  3  for each row
```

```
  4  begin
  5    dbms_output.put_line('Before update row');
  6  end;
  7  /

Trigger creato.

WTO >create trigger com_a_u_s
  2  after update on comuni
  3  begin
  4    dbms_output.put_line('After update statement');
  5  end;
  6  /

Trigger creato.

WTO >create trigger com_a_u_e
  2  after update on comuni
  3  for each row
  4  begin
  5    dbms_output.put_line('After update row');
  6  end;
  7  /

Trigger creato.

WTO >update comuni set des_comune='nuova descrizione';
Before update statement
Before update row
After update row
Before update row
After update row
Before update row
After update row
Before update row
After update row
Before update row
After update row
Before update row
After update row
Before update row
After update row
Before update row
After update row
Before update row
After update row
After update statement

Aggiornate 9 righe.

WTO >rollback;
Rollback completato.
```

UTILIZZI FREQUENTI

Gli utilizzi più frequenti dei database trigger sono:

- Valorizzazione automatica di campi
- Controlli di validità complessi
- Log delle operazioni DML

Vediamo nel seguito un esempio per ciascuna di queste casistiche.

Cominciamo dalla valorizzazione automatica di colonne. In caso di inserimento in tabella CLIENTI vogliamo che il programmatore non debba preoccuparsi di calcolare il corretto codice cliente da utilizzare.

Il trigger seguente valorizza automaticamente il codice cliente utilizzando la sequenza SEQ_CLIENTE.

```
WTO >create or replace trigger clienti_bir
  2  before insert on clienti
  3  for each row
  4  begin
  5    select seq_clienti.nextval
  6      into :new.cod_cliente
  7      from dual;
  8  end;
  9  /
Trigger creato.

WTO >insert into clienti (nome, cognome, indirizzo, cap, comune)
  2  values ('Luca','Bianchi','Via Po, 1', '00123', 'H501');
Creata 1 riga.

WTO >select * from clienti;

COD NOME     COGNOME  COD_FISC         INDIRIZZO              CAP   COMU
--- -------- -------- ---------------- ---------------------- ----- ----
  1 MARCO    ROSSI    RSSMRC70R20H501X VIA LAURENTINA, 700    00143 H501
  2 GIOVANNI BIANCHI                   VIA OSTIENSE, 850      00144 H501
  3 MATTEO   VERDI    VRDMTT69S02H501X VIA DEL MARE, 8        00033 G811
  4 LUCA     NERI     NRILCU77A22F205X VIA TORINO, 30         20120 F205
  5 AMBROGIO COLOMBO                   PIAZZA DUOMO, 1        20100 F205
  6 GENNARO  ESPOSITO SPSGNN71B10F839X VIA CARACCIOLO, 100    80100 F839
  7 PASQUALE RUSSO    RSSPSQ70C14F839X VIA GIULIO CESARE, 119 80125 F839
  8 VINCENZO AMATO                     VIA NAPOLI, 234        80022 G964
 22 Luca     Bianchi                   Via Po, 1              00123 H501

Selezionate 9 righe.

WTO >rollback;
Rollback completato.
```

Nell'esempio seguente si verifica se il comune indicato in fase di inserimento di un cliente ha una descrizione che comincia con la lettera R. Se la condizione non è verificata il trigger impedisce l'inserimento sollevando un errore. Una tale regola non è implementabile mediante un check constraint.

```
WTO >create or replace trigger clienti_bir
  2  before insert on clienti
  3  for each row
  4  declare
  5    des comuni.des_comune%type;
  6  begin
  7    select des_comune
  8      into des
  9      from comuni
 10     where cod_comune = :new.comune;
 11
 12   if des not like 'R%' then
 13    RAISE_APPLICATION_ERROR(-20001,des||' Non inizia per R.');
 14   else
```

```
 15    select seq_clienti.nextval
 16      into :new.cod_cliente
 17      from dual;
 18   end if;
 19  end;
 20  /

Trigger creato.

WTO >insert into clienti (nome, cognome, indirizzo, cap, comune)
  2  values('Luca','Bianchi','Via Po, 1', '00123', 'F839');
insert into clienti (nome, cognome, indirizzo, cap, comune)
            *
ERRORE alla riga 1:
ORA-20001: NAPOLI Non inizia per R.
ORA-06512: at "WTO_ESEMPIO.CLIENTI_BIR", line 10
ORA-04088: error during execution of trigger 'WTO_ESEMPIO.CLIENTI_BIR'


WTO >insert into clienti (nome, cognome, indirizzo, cap, comune)
  2  values ('Luca','Bianchi','Via Po, 1', '00123', 'H501');

Creata 1 riga.

WTO >rollback;

Rollback completato.
```

Il tentativo di inserimento di un cliente sul codice comune F839 (Napoli) solleva l'errore "ORA-20001: NAPOLI Non inizia per R.", mentre il comune H501 (Roma) può essere inserito senza problemi.

Come terzo esempio si mostra un trigger che salva nella tabella LOG_CLIENTI le modifiche apportate alla tabella CLIENTI.

Innanzitutto definiamo la tabella LOG_CLIENTI.

```
WTO >create table LOG_CLIENTI (
  2  data_modifica date,
  3  cliente number(8),
  4  modifica varchar2(20)
  5  );

Tabella creata.
```

Poi creiamo il trigger.

```
WTO >create or replace trigger clienti_a_iud
  2  after insert or update or delete on clienti
  3  for each row
  4  declare
  5    evento varchar2(20);
  6  begin
  7    if inserting then
  8       evento := 'Inserimento';
  9    elsif updating then
 10       evento := 'Aggiornamento';
 11    else
 12       evento := 'Cancellazione';
 13    end if;
 14
 15    insert into log_clienti values
 16    (sysdate, nvl(:old.cod_cliente,:new.cod_cliente),evento);
```

```
 17  end;
 18  /

Trigger creato.
```

Eseguiamo alcune operazioni DML.

```
WTO >insert into clienti (nome, cognome, indirizzo, cap, comune)
  2  values ('Matteo','Bianchi','Via Po 1', '00100', 'H501');

Creata 1 riga.

WTO >update clienti set nome='Franco'
  2  where cod_cliente=6;

Aggiornata 1 riga.

WTO >update clienti set nome='Michele'
  2  where cod_cliente=3;

Aggiornata 1 riga.

WTO >select seq_clienti.currval from dual;

   CURRVAL
----------
        26

WTO >delete clienti where cod_cliente=26;

Eliminata 1 riga.
```

E verifichiamo il contenuto della tabella LOG_CLIENTI.

```
WTO >select * from log_clienti;

DATA_MODI    CLIENTE MODIFICA
--------- ---------- --------------------
30-MAR-11         26 Inserimento
30-MAR-11          6 Aggiornamento
30-MAR-11          3 Aggiornamento
30-MAR-11         26 Cancellazione

WTO >rollback;

Rollback completato.
```

MUTATING TABLE

Un trigger FOR EACH ROW scatta durante l'esecuzione di un comando DML. Durante il trigger, dunque, la tabella è in fase di cambiamento (*mutating* in inglese) e non può essere letta o modificata. Facciamo un esempio. Aggiungiamo al trigger dell'esempio precedente una banale query che conta il numero di record presenti in CLIENTI.

```
WTO >create or replace trigger clienti_a_iud
  2  after insert or update or delete on clienti
  3  for each row
  4  declare
  5    evento varchar2(20);
  6    n number;
```

```
  7  begin
  8    if inserting then
  9       evento := 'Inserimento';
 10    elsif updating then
 11       evento := 'Aggiornamento';
 12    else
 13       evento := 'Cancellazione';
 14    end if;
 15
 16    insert into log_clienti values
 17    (sysdate, nvl(:old.cod_cliente,:new.cod_cliente),evento);
 18
 19    select count(*) into n
 20      from clienti;
 21  end;
 22  /

Trigger creato.
```

Se adesso facciamo scattare il trigger con un UPDATE si verifica un errore.

```
WTO >update clienti
  2  set nome='pippo';
update clienti
       *
ERRORE alla riga 1:
ORA-04091: table WTO_ESEMPIO.CLIENTI is mutating, trigger/function may
not see it
ORA-06512: at "WTO_ESEMPIO.CLIENTI_A_IUD", line 16
ORA-04088: error during execution of trigger
'WTO_ESEMPIO.CLIENTI_A_IUD'
```

Oracle ci comunica che la tabella sta cambiando e dunque il trigger non la può vedere, lo stato non è consistente.

Con un trigger di tipo STATEMENT questo problema non si verifica.

```
WTO >create or replace trigger clienti_a_iud
  2  after insert or update or delete on clienti
  3  declare
  4    n number;
  5  begin
  6  select count(*) into n
  7    from clienti;
  8  end;
  9  /

Trigger creato.

WTO >update clienti
  2  set nome='pippo';

Aggiornate 8 righe.

WTO >rollback;

Rollback completato.
```

Infatti, il trigger di tipo STATEMENT può essere eseguito prima (se è BEFORE) o dopo (se è AFTER) le modifiche del comando DML quando la tabella ha uno stato comunque consistente.

Trigger INSTEAD OF

Abbiamo visto nel capitolo dell'SQL che alcune viste sono modificabili, su di esse è possibile eseguire i comandi DML. La maggior parte delle viste, però, non consente di eseguire comandi DML. Sulle viste non modificabili è possibile definire uno speciale tipo di trigger, il trigger INSTEAD OF.

Questo tipo di trigger scatta al posto dell'operazione DML per cui è definito. Nell'esempio seguente si definisce la vista CLI_COM.

```
WTO >select * from cli_com;

NOME     COGNOME  DES_COMUNE
-------- -------- -------------------------------------
LUCA     NERI     MILANO
AMBROGIO COLOMBO  MILANO
PASQUALE RUSSO    NAPOLI
GENNARO  ESPOSITO NAPOLI
MATTEO   VERDI    POMEZIA
VINCENZO AMATO    POZZUOLI
GIOVANNI BIANCHI  ROMA
MARCO    ROSSI    ROMA

Selezionate 8 righe.
```

Questo tipo di vista è aggiornabile ma non inseribile. In caso di aggiornamento scatta il trigger posto sulla tabella CLIENTI sottostante, in caso di INSERT Oracle solleva un errore.

```
WTO >update cli_com
  2  set nome='Pippo'
  3  where cognome='NERI';
Cliente 4 vecchio nome LUCA nuovo Pippo

Aggiornata 1 riga.

WTO >rollback;

Rollback completato.

WTO >insert into cli_com values ('Pippo','Pluto','Roma');
insert into cli_com values ('Pippo','Pluto','Roma')
*
ERRORE alla riga 1:
ORA-01776: cannot modify more than one base table through a join view
```

Sulla vista è possibile definire un trigger INSTEAD OF INSERT in cui il programmatore può decidere come gestire i comandi di inserimento eseguiti sulla vista.

```
WTO >create or replace trigger test_instead
  2  instead of insert on CLI_COM
  3  FOR EACH ROW
  4  begin
  5    dbms_output.put_line('Tentato inserimento in CLI_COM');
  6  end;
  7  /
```

```
Trigger creato.

WTO >insert into cli_com values ('Pippo','Pluto','Roma');
Tentato inserimento in CLI_COM

Creata 1 riga.
```

Invece di effettuare l’inserimento (ed ottenere l’errore, Oracle esegue il trigger e si ferma, restituendo il messaggio “Creata 1 riga” che ha ovviamente un valore relativo perché il trigger potrebbe avere fatto qualunque cosa nel DB, in questo caso non ha fatto null’altro che visualizzare il messaggio “Tentato inserimento in CLI_COM”.

8.8.5 Dipendenze tra oggetti del DB

Gli oggetti sul DB non sono tutti indipendenti tra loro. Giusto per fare un esempio una vista è dipendente dalle tabelle e viste su cui è eseguita la query. Se tali tabelle e viste cambiano, l’istruzione SQL che definisce la vista potrebbe non essere più sintatticamente corretta. Allo stesso modo una procedura o funzione PL/SQL è dipendente dagli oggetti di database che utilizza e dagli altri programmi PL/SQL che chiama.

Man mano che gli utenti creano o modificano oggetti nel database, Oracle costruisce un grafo delle dipendenze in modo da poter velocemente determinare, dato un oggetto, quali sono gli oggetti che da esso dipendono. Quando un oggetto viene modificato, qualunque sia la modifica, oppure eliminato Oracle marca come “non validi” tutti gli oggetti di database da esso dipendenti. Oracle non verifica immediatamente se gli oggetti dipendenti sono diventati effettivamente inutilizzabili, li marca tutti come “non validi” a priori. Quando un oggetto viene utilizzato, se è marcato “non valido”, Oracle verifica se effettivamente può essere nuovamente validato oppure no. Quest’attività ovviamente richiede un certo tempo di elaborazione che appesantisce l’istruzione SQL che l’ha generata.

Le dipendenze tra oggetti sono conservate nella vista di dizionario USER_DEPENDENCIES. La vista contiene una riga per ogni dipendenza.

```
WTO >select name, type, REFERENCED_NAME, REFERENCED_TYPE
  2  from user_dependencies
  3  ;

NAME            TYPE         REFERENCED_NAME REFERENCED_TYPE
--------------- ------------ --------------- ---------------
CLIENTI_BIR     TRIGGER      DUAL            SYNONYM
CLIENTI_A_IUD   TRIGGER      STANDARD        PACKAGE
COM_A_U_E       TRIGGER      STANDARD        PACKAGE
COM_A_U_S       TRIGGER      STANDARD        PACKAGE
COM_B_U_E       TRIGGER      STANDARD        PACKAGE
COM_B_U_S       TRIGGER      STANDARD        PACKAGE
TEST_INSTEAD    TRIGGER      STANDARD        PACKAGE
TEST_PKG        PACKAGE      STANDARD        PACKAGE
CLIENTI_BIR     TRIGGER      STANDARD        PACKAGE
TEST_PKG        PACKAGE BODY STANDARD        PACKAGE
MYXML           TABLE        STANDARD        PACKAGE
CLIENTI_BIR     TRIGGER      DBMS_STANDARD   PACKAGE
COM_A_U_E       TRIGGER      DBMS_OUTPUT     SYNONYM
COM_A_U_S       TRIGGER      DBMS_OUTPUT     SYNONYM
```

```
COM_B_U_E        TRIGGER      DBMS_OUTPUT      SYNONYM
COM_B_U_S        TRIGGER      DBMS_OUTPUT      SYNONYM
TEST_INSTEAD     TRIGGER      DBMS_OUTPUT      SYNONYM
TEST_PKG         PACKAGE BODY DBMS_OUTPUT      SYNONYM
COM_A_U_E        TRIGGER      COMUNI           TABLE
COM_A_U_S        TRIGGER      COMUNI           TABLE
COM_B_U_E        TRIGGER      COMUNI           TABLE
COM_B_U_S        TRIGGER      COMUNI           TABLE
CLI_COM_OJ       VIEW         COMUNI           TABLE
CLI_COM          VIEW         COMUNI           TABLE
CLIENTI_BIR      TRIGGER      COMUNI           TABLE
CLI_COM          VIEW         CLIENTI          TABLE
CLI_COM_OJ       VIEW         CLIENTI          TABLE
CLIENTI_A_IUD    TRIGGER      CLIENTI          TABLE
CLIENTI_BIR      TRIGGER      CLIENTI          TABLE
CLIENTI_BIR      TRIGGER      SEQ_CLIENTI      SEQUENCE
COM_A_U_E        TRIGGER      DBMS_OUTPUT      NON-EXISTENT
COM_A_U_S        TRIGGER      DBMS_OUTPUT      NON-EXISTENT
COM_B_U_E        TRIGGER      DBMS_OUTPUT      NON-EXISTENT
COM_B_U_S        TRIGGER      DBMS_OUTPUT      NON-EXISTENT
TEST_INSTEAD     TRIGGER      DBMS_OUTPUT      NON-EXISTENT
TEST_PKG         PACKAGE BODY DBMS_OUTPUT      NON-EXISTENT
TEST_INSTEAD     TRIGGER      CLI_COM          VIEW
TEST_PKG         PACKAGE BODY TEST_PKG         PACKAGE
TEST_PKG         PACKAGE BODY FATTURE          TABLE
```

Le colonne NAME e TYPE individuano un oggetto, i campi REFERENCED_NAME e REFERENCED_TYPE individuano l'oggetto da cui esso dipende. Nel campo in cui la dipendenza sia da un oggetto collocato in un altro schema si può leggere il campo REFERENCED_SCHEMA.

Il package STANDARD è un package di sistema che contiene tutte le procedure e funzioni predefinite che si utilizzano senza nome di package, ad esempio RAISE_APPLICATION_ERROR.

Le righe aventi REFERENCED_TYPE = 'NON-EXISTENT' possono essere ignorate. Sono dei record di servizio che Oracle aggiunge quando si crea una dipendenza da un sinonimo pubblico (DBMS_OUTPUT in questo caso).

La vista USER_DEPENDENCIES contiene solo le dipendenze dirette ma, ovviamente, il database è spesso ricco di dipendenze indirette. Costruiamo un esempio.

La procedura P1 richiama la funzione F1. La funzione F1 richiama la procedura P2. La procedura P2 utilizza la tabella CLIENTI.

```
WTO >create or replace procedure P2 is
  2  a number;
  3  begin
  4    select count(*) into a from clienti;
  5  end;
  6  /

Procedure created.

WTO >create or replace function F1 return number is
  2  begin
  3    p2;
  4    return 0;
```

```
  5  end;
  6  /

Function created.

WTO >create or replace procedure P1 is
  2  a number;
  3  begin
  4    a := f1;
  5  end;
  6  /

Procedure created.

WTO >select name, type, REFERENCED_NAME, REFERENCED_TYPE
  2  from user_dependencies
  3* where name in ('P1','P2','F1')

NAME TYPE         REFERENCED_NAME              REFERENCED_TYPE
---- ------------ ---------------------------- ---------------
P2   PROCEDURE    STANDARD                     PACKAGE
P1   PROCEDURE    STANDARD                     PACKAGE
F1   FUNCTION     STANDARD                     PACKAGE
P2   PROCEDURE    SYS_STUB_FOR_PURITY_ANALYSIS PACKAGE
P1   PROCEDURE    SYS_STUB_FOR_PURITY_ANALYSIS PACKAGE
F1   FUNCTION     SYS_STUB_FOR_PURITY_ANALYSIS PACKAGE
F1   FUNCTION     P2                           PROCEDURE
P1   PROCEDURE    F1                           FUNCTION
P2   PROCEDURE    CLIENTI                      TABLE

9 rows selected.
```

Il package SYS_STUB_FOR_PURITY_ANALYSIS, come dice il nome, è un oggetto di sistema utilizzato per la verifica di “purezza” delle procedure e funzioni. Di quest’argomento si parlerà più avanti nel paragrafo dedicato alla PRAGMA RESTRICT_REFERENCES.

Dalla query non emerge, ad esempio, che la procedura P1 dipende sia dalla tabella CLIENTI che dalla procedura P2.

Per vedere le dipendenze in maniera gerarchica si può utilizzare una CONNECT BY.

```
WTO >select lpad(' ',level-1,' ')||REFERENCED_NAME REF_NAME,
  2         REFERENCED_TYPE
  3  from user_dependencies
  4  connect by prior REFERENCED_NAME = name
  5      and prior REFERENCED_TYPE = type
  6  start with name='P1';

REF_NAME                                 REFERENCED_TYPE
---------------------------------------- ---------------
STANDARD                                 PACKAGE
SYS_STUB_FOR_PURITY_ANALYSIS             PACKAGE
F1                                       FUNCTION
 STANDARD                                PACKAGE
 SYS_STUB_FOR_PURITY_ANALYSIS            PACKAGE
 P2                                      PROCEDURE
  STANDARD                               PACKAGE
  SYS_STUB_FOR_PURITY_ANALYSIS           PACKAGE
  CLIENTI                                TABLE
```

```
9 rows selected.
```

La query precedente, partendo da P1, mostra dapprima le dipendenze dirette (SYS_STUB_FOR_PURITY_ANALYSIS, STANDARD, F1) e, per ognuna di queste, mostra le dipendenze di secondo livello. Ne ha solo F1 che dipende da P2 e dagli stessi oggetti di sistema. A questo punto vengono mostrate le dipendenze di P2 (le solite più CLIENTI) che sono dipendenze di terzo livello per P1. Per evidenziare graficamente la differenza nel livello di dipendenza è stato aggiunto uno spazio in testa per ogni livello superiore al primo. Se si preferisce si può visualizzare esplicitamente il livello.

```
WTO >select lpad(level||')',level+1,' ')||REFERENCED_NAME
  2  REF_NAME, REFERENCED_TYPE
  3  from user_dependencies
  4  connect by prior REFERENCED_NAME = name
  5      and prior REFERENCED_TYPE = type
  6  start with name='P1';

REF_NAME                                 REFERENCED_TYPE
---------------------------------------- ---------------
1)STANDARD                               PACKAGE
1)SYS_STUB_FOR_PURITY_ANALYSIS           PACKAGE
1)F1                                     FUNCTION
 2)STANDARD                              PACKAGE
 2)SYS_STUB_FOR_PURITY_ANALYSIS          PACKAGE
 2)P2                                    PROCEDURE
  3)STANDARD                             PACKAGE
  3)SYS_STUB_FOR_PURITY_ANALYSIS         PACKAGE
  3)CLIENTI                              TABLE

9 rows selected.
```

In alternativa si può utilizzare una DISTINCT per estrarre solo una lista degli oggetti da cui P1 dipende, a qualunque livello.

```
WTO >select distinct REFERENCED_NAME, REFERENCED_TYPE
  2  from user_dependencies
  3  connect by prior REFERENCED_NAME = name
  4      and prior REFERENCED_TYPE = type
  5  start with name='P1';

REFERENCED_NAME              REFERENCED_TYPE
---------------------------- ---------------
F1                           FUNCTION
P2                           PROCEDURE
SYS_STUB_FOR_PURITY_ANALYSIS PACKAGE
STANDARD                     PACKAGE
CLIENTI                      TABLE
```

8.8.1 La clausola ACCESSIBLE BY

In Oracle 12c è possibile indicare, a fronte di un programma PL/SQL, quali sono gli altri programmi PL/SQL che lo possono utilizzare. Facciamo un esempio.

La funzione TEST_WL, di seguito definita, può essere utilizzata solo dalle due program unit TEST_CALL_1 e TEST_CALL_2, la TEST_CALL_3 non ne può fare uso:

```
O12c>create or replace function test_wl return number
  2  ACCESSIBLE BY (test_call_1, test_call_2) is
  3  begin
  4    return 1;
  5  end;
  6  /
Funzione creata.

O12c>create or replace function test_call_1 return number is
  2  begin
  3    return test_wl;
  4  end;
  5  /
Funzione creata.

O12c>create or replace procedure test_call_2 is
  2  a number;
  3  begin
  4    a:=test_wl;
  5  end;
  6  /

Procedura creata.

O12c>create or replace function test_call_3 return number is
  2  begin
  3    return test_wl;
  4  end;
  5  /

Avvertimento: funzione creata con errori di compilazione.

O12c>sho err
Errori in FUNCTION TEST_CALL_3:

LINE/COL ERROR
-------- -------------------------------------------------------
3/3      PL/SQL: Statement ignored
3/10     PLS-00904: privilegio non sufficiente per avere accesso
         a oggetto TEST_WL
```

8.8.2 Diritti d’esecuzione

Come tutti gli oggetti del database, anche i programmi PL/SQL possono essere condivisi con altri utenti. In questo caso è importante tenere presente che, come impostazione predefinita, il programma è eseguito con i privilegi dell’utente che lo possiede, anche se viene lanciato da un altro utente.

Facciamo un esempio. Connessi con l’utente WTO_ESEMPIO creiamo una tabella TEST avente una sola colonna numerica ed inseriamo un record con il valore 1.

```
WTO >create table test (a number);

Table created.

WTO >insert into test values (1);

1 row created.
```

```
WTO >commit;

Commit complete.
```

Nello stesso schema WTO_ESEMPIO definiamo anche una funzione che restituisce il valore appena inserito.

```
WTO >create or replace function leggi return number is
  2  a number;
  3  begin
  4    select a into a from test;
  5    return a;
  6  end;
  7  /

Function created.

WTO >select leggi from dual;

    LEGGI
---------
        1
```

A questo punto possiamo autorizzare l'utente SCOTT ad utilizzare questa funzione.

```
WTO >grant execute on leggi to scott;

Grant succeeded.
```

Per verificare che tutto torni colleghiamoci con SCOTT e lanciamo la funzione leggi.

```
WTO >conn scott/tiger
Connected.
WTO >select wto_esempio.leggi from dual;

    LEGGI
---------
        1

WTO >desc wto_esempio.test
ERROR:
ORA-04043: object wto_esempio.test does not exist
```

La funzione restituisce il valore 1 letto dalla tabella TEST di WTO_ESEMPIO. SCOTT non ha il diritto di accedere direttamente a quella tabella, quando esegue la funzione acquisisce i diritti di WTO_ESEMPIO e, dunque, riesce a leggere il dato.

Aggiungiamo adesso nello schema di SCOTT la stessa tabella TEST ma con un valore differente.

```
WTO >create table test (a number);

Table created.
```

```
WTO >insert into test values (2);

1 row created.

WTO >commit;

Commit complete.
```

Come prima, se eseguiamo da SCOTT la funzione LEGGI essa gira con i diritti del proprietario (WTO_ESEMPIO) e dunque legge i dati dalla tabella TEST di WTO_ESEMPIO, non dall'omonima tabella di SCOTT.

```
WTO >select wto_esempio.leggi from dual;

    LEGGI
---------
        1
```

Per cambiare questo comportamento si può utilizzare la clausola AUTHID CURRENT_USER nella definizione della funzione. Questa clausola dice ad Oracle che la funzione deve essere invocata sempre con i diritti di chi la lancia, non con quelli del suo proprietario.

Colleghiamoci nuovamente con l'utente WTO_ESEMPIO e ridefiniamo la funzione.

```
WTO >conn wto_esempio/esempio
Connected.
WTO >create or replace function leggi return number
  2  authid current_user is
  3  a number;
  4  begin
  5    select a into a from test;
  6    return a;
  7  end;
  8  /

Function created.

WTO >select leggi from dual;

    LEGGI
---------
        1
```

Il comportamento in WTO_ESEMPIO non cambia (l'utente che ha lanciato la funzione è lo stesso utente che ne è proprietario).

Collegandoci con SCOTT e lanciando la funzione otteniamo un risultato differente.

```
WTO >conn scott/tiger
Connected.
WTO >select wto_esempio.leggi from dual;

    LEGGI
---------
        2
```

Adesso la funzione è stata eseguita con i diritti di SCOTT e dunque ha letto la tabella TEST di SCOTT, non quella di WTO_ESEMPIO.

8.8.3 La clausola BEQUEATH delle viste

In Oracle 12c è stata introdotta la clausola BEQUEATH nella creazione delle viste. Questa clausola consente di far eseguire le funzioni richiamate nella vista in modalità AUTHID CURRENT_USER.

Per capire bene come funziona è necessario richiamare un comportamento standard di Oracle non molto noto.

Definiamo due utenti, USR1 ed USR2:

```
O12c>grant connect, resource, create view, unlimited tablespace
   2 to usr1 identified by usr1;

Concessione riuscita.

O12c>grant connect, resource, create view, unlimited tablespace
   2 to usr2 identified by usr2;

Concessione riuscita.
```

Connettiamoci ad USR1 e creiamo una tabella contenente un solo valore (1):

```
O12c>conn usr1/usr1@corsopdb

usr1>create table test_beq (a number);

Tabella creata.

usr1>insert into test_beq values (1);

Creata 1 riga.
```

Poi creiamo una funzione che restituisce l'unico valore presente in tabella TEST_BEQ. La funzione ha AUTHID CURRENT_USER:

```
usr1>create or replace function fun_beq return number
  2  authid current_user is
  3  a number;
  4  begin
  5    select a into a from test_beq;
  6    return a;
  7  end;
  8  /
Funzione creata.

usr1>select  fun_beq from dual;

   FUN_BEQ
----------
         1
```

A questo punto creiamo una vista, VIEW_BEQ, che legge il solito valore attraverso la funzione FUN_BEQ:

```
usr1>create view view_beq as
  2  select  fun_beq from dual;
Vista creata.

usr1>select * from view_beq;
```

```
   FUN_BEQ
----------
         1
```

Adesso concediamo all'utente USR2 il diritto di eseguire sia SELECT sulla tabella TEST_BEQ, sia EXECUTE sulla funzione FUN_BEQ:

```
usr1>grant select on view_beq to usr2;

Concessione riuscita.

usr1>grant execute on fun_beq to usr2;

Concessione riuscita.
```

Colleghiamoci con l'utente USR2 e definiamo anche nel suo schema la tabella TEST_BEQ, inserendo però il valore 2:

```
usr1>conn usr2/usr2@corsopdb
Connesso.

usr2>create table test_beq (a number);

Tabella creata.

usr2>insert into test_beq values (2);

Creata 1 riga.
```

Ovviamente, poiché la funzione FUN_BEQ è definita AUTHID CURRENT_USER, se USR2 la esegue ottiene il valore 2:

```
usr2>select usr1.fun_beq from dual;

   FUN_BEQ
----------
         2
```

Però, ed è questo il comportamento di Oracle che non molti conoscono, se USR2 legge il valore attraverso la vista VIEW_BEQ non ottiene 2, bensì 1:

```
usr2>select * from usr1.view_beq;

   FUN_BEQ
----------
         1
```

In pratica l'AUTHID CURRENT_USER definito nella funzione è annullato dal fatto che la funzione è richiamata non direttamente ma attraverso una vista. Questo è il comportamento di Oracle sia nella versione 11g che nella 12c.

In Oracle 12c, però, è stata introdotta la clausola BEQUEATH nelle viste. Una vista definita in questo modo non annulla l'effetto degli AUTHID CURRENT_USER. Rifacciamo l'esempio. Connettiamoci con USR1:

```
usr2>conn usr1/usr1@corsopdb
```

```
Connesso.
```

Droppiamo e creiamo di nuovo la vista, questa volta con la clausola BEQUEATH:

```
usr1>drop view view_beq;

Vista eliminata.

usr1>create view view_beq
  2  bequeath current_user as
  3  select  fun_beq from dual;

Vista creata.
```

Concediamo di nuovo ad USR2 il permesso di fare SELECT sulla vista:

```
usr1>grant select on view_beq to usr2;

Concessione riuscita.
```

Colleghiamoci ad USR2:

```
usr1>conn usr2/usr2@corsopdb
Connesso.
```

La query dalla funzione restituisce sempre 2, ma adesso anche la query dalla vista restituisce lo stesso valore:

```

usr2>conn usr2/usr2@corsopdb;
Connesso.
usr2>select usr1.fun_beq from dual;

   FUN_BEQ
----------
         2

usr2>select * from usr1.view_beq;

   FUN_BEQ
----------
         2
```

L'AUTHID CURRENT_USER è stato correttamente propagato attraverso la vista.

8.8.4 Controllo del codice PL/SQL

Quando si crea un programma PL/SQL nel database Oracle analizza il codice e lo salva nel DB. Per leggere il codice si può accedere alla vista di dizionario USER_SOURCE. In questa vista sono conservate separatamente le singole linee di programma. La vista contiene quattro colonne: NAME (il nome della procedura, funzione o package), TYPE (il tipo di programma), LINE (il numero di linea) e TEXT (il codice PL/SQL).

Ad esempio guardiamo il codice PL/SQL della funzione LEGGI definita nel paragrafo precedente.

```
WTO >select * from user_source
  2* where name='LEGGI'

NAME  TYPE          LINE TEXT
----- ------------ ----- -----------------------------
LEGGI FUNCTION         1 function leggi return number
LEGGI FUNCTION         2 authid current_user is
LEGGI FUNCTION         3 a number;
LEGGI FUNCTION         4 begin
LEGGI FUNCTION         5   select a into a from test;
LEGGI FUNCTION         6   return a;
LEGGI FUNCTION         7 end;

7 rows selected.

WTO >select text
  2  from user_source
  3  where name='LEGGI'
  4  order by line;

TEXT
-------------------------------
function leggi return number
authid current_user is
a number;
begin
  select a into a from test;
  return a;
end;

7 rows selected.
```

Il codice dei trigger non è conservato in questa vista ma nella vista USER_TRIGGERS.

```
WTO >desc user_triggers
 Name                               Null?    Type
 ------------------------------- -------- ----
 TRIGGER_NAME                               VARCHAR2(30)
 TRIGGER_TYPE                               VARCHAR2(16)
 TRIGGERING_EVENT                           VARCHAR2(227)
 TABLE_OWNER                                VARCHAR2(30)
 BASE_OBJECT_TYPE                           VARCHAR2(16)
 TABLE_NAME                                 VARCHAR2(30)
 COLUMN_NAME                                VARCHAR2(4000)
 REFERENCING_NAMES                          VARCHAR2(128)
 WHEN_CLAUSE                                VARCHAR2(4000)
 STATUS                                     VARCHAR2(8)
 DESCRIPTION                                VARCHAR2(4000)
 ACTION_TYPE                                VARCHAR2(11)
 TRIGGER_BODY                               LONG
```

La colonna TRIGGER_BODY contiene il codice PL/SQL.

```
WTO >select TRIGGER_BODY
  2  from user_triggers
  3  where trigger_name='CLIENTI_A_IUD';

TRIGGER_BODY
-------------------------------------------
declare
```

```
  n number;
begin
select count(*) into n
  from clienti;
end;
```

Il codice PL/SQL viene conservato nel database IN CHIARO e dunque accessibile a chiunque abbia accesso allo specifico schema di database. Nel caso in cui fosse necessario “oscurare” il codice, ad esempio quando si vende un software in licenza d’uso senza diritto s’accesso ai sorgenti per l’acquirente, è possibile utilizzare l’utility WRAP. Questa utility prende in input un file contenente la definizione di un programma PL/SQL e lo cifra in modo che sia sempre leggibile dal database ma illeggibile per l’essere umano.

Per fare un esempio prendiamo il codice PL/SQL della funzione LEGGI e copiamolo in un file leggi.sql nella directory BIN del database (ad esempio D:\oracle\product\11.2.0\Db_1\BIN).

Stando nella directory BIN dal prompt dei comandi eseguiamo l’istruzione.

```
wrap iname=leggi.sql oname=leggi_wrap.sql
```

Questo comando legge il file leggi.sql e crea il file leggi_wrap.sql. Nell’immagine seguente i due file sono messi a confronto.

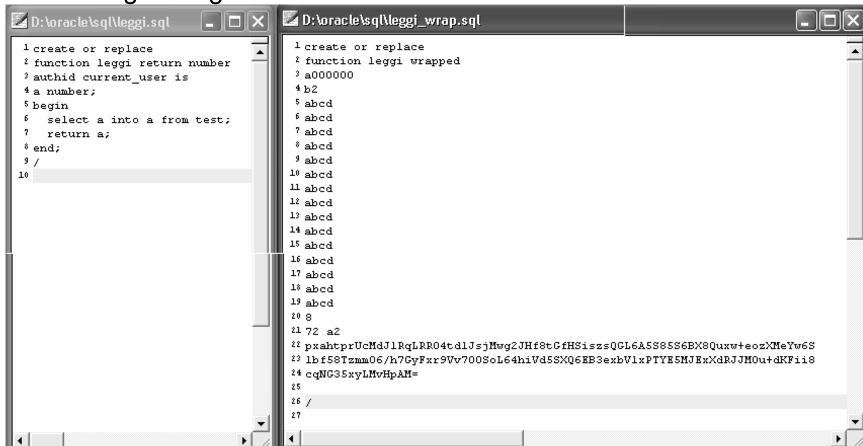


Figura 8-3 Uno script PL/SQL in chiaro ed offuscato.

Il file leggi_wrap.sql è uno script sql valido e perfettamente equivalente al leggi.sql che possiamo eseguire da SQL*Plus

```
WTO >@leggi_wrap

Function created.
```

Ottenendo la stessa funzione di prima, ma con il codice offuscato.

```
WTO >select leggi from dual;

    LEGGI
---------
        1
```

```
WTO >select text
  2  from user_source
  3  where name='LEGGI'
  4  order by line;

TEXT
----------------------------------------------------------
function leggi wrapped
a000000
b2
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
8
72 a2
pxahtprUcMdJlRqLRR04tdlJsjMwg2JHf8tGfHSiszsQGL6A5S85S6BX8Quxw+eozXMeYw6S
lbf58TzmmO6/h7GyFxr9Vv700SoL64hiVd5SXQ6EB3exbVlxPTYE5MJExXdRJJM0u+dKFii8
cqNG35xyLMvHpAM=
```

Attenzione: non esiste un’utility UNWRAP che consenta di ottenere il codice in chiaro partendo dal codice offuscato, quindi bisogna conservare accuratamente gli script sql in chiaro per le modifiche future.

8.8.5 Le direttive di compilazione PRAGMA

Come abbiamo visto nel paragrafo precedente a partire dalla versione 10 del DB Oracle ha introdotto nel PL/SQL la possibilità di effettuare compilazione condizionale.

Esistono però altre direttive di compilazione presenti in PL/SQL da molto prima, le PRAGMA.

In questo paragrafo analizzeremo una per una le cinque PRAGMA presenti in Oracle11g. Le prime 4 esistono almeno dalla versione 8i mentre l'ultima è stata introdotta proprio nella versione 11g del PL/SQL.

Iniziamo con la PRAGMA EXCEPTION_INIT. Questa direttiva consente di associare un codice d'errore ORA ad una user defined exception.

Fatta l'associazione sarà possibile gestire nel codice l'eccezione come se fosse un'eccezione predefinita (ad esempio NO_DATA_FOUND o TOO_MANY_ROWS).

Vediamo un esempio.

Immaginiamo di dover creare un funzione che riceve in input una stringa e la trasforma in data assumendo come formato 'YYYY-MM-DD':

```
WTO> create or replace function stringa2data
```

```
  2  (str in varchar2) return date is
  3  retDate date;
  4  begin
  5    retDate := to_date(str,'yyyy-mm-dd');
  6    return retDate;
  7  end;
  8  /

Funzione creata.

WTO> select stringa2data('2010-01-31')
  2  from dual;

STRINGA2D
---------
31-GEN-10

WTO> select stringa2data('werrwer')
  2  from dual;
select stringa2data('werrwer')
       *
ERRORE alla riga 1:
ORA-01841: l'anno (completo) deve essere compreso tra -4713 e +9999, ed
essere
diverso da 0
ORA-06512: a "WTO_ESEMPIO.STRINGA2DATA", line 4
```

Come abbiamo visto nell'esempio, se la stringa in input non è valida rispetto al formato otteniamo l'errore ORA-1841.

Ci proponiamo di gestire quest'errore con l'aiuto della direttiva PRAGMA EXCEPTION_INIT:

```
WTO> create or replace function stringa2data (str in varchar2)
  2  return date is
  3  retDate date;
  4  data_non_corretta exception;
  5  PRAGMA EXCEPTION_INIT(data_non_corretta,-1841);
  6  begin
  7    retDate := to_date(str,'yyyy-mm-dd');
  8    return retDate;
  9  exception
 10    when data_non_corretta then
 11     dbms_output.put_line('Attenzione: la stringa '||str||
 12                     ' non può essere convertita in data!');
 13     return null;
 14  end;
 15  /

Funzione creata.

WTO> set serverout on
WTO> select stringa2data('werrwer')
  2  from dual;

STRINGA2D
---------



Attenzione: la stringa werrwer non può essere convertita in data!
```

È stata definita una nuova eccezione DATA_NON_CORRETTA, ma nessuno l'avrebbe mai sollevata se non l'avessimo associata mediante PRAGMA all'errore ORA-1841.

A questo punto Oracle sa che in caso si verifichi l'errore ORA-1841 deve passare il controllo al blocco exception dove abbiamo provveduto a gestire la situazione.

La PRAGMA RESTRICT_REFERENCES consente di dichiarare esplicitamente il livello di “purezza” di un programma PL/SQL, cioè se esso legge/scrive oggetti di database o variabili di package.

In alcuni contesti, infatti, solo funzioni che garantiscono tali restrizioni possono essere utilizzate.

Un esempio classico è il seguente.

Definiamo un package composto da un'unica funzione che aggiorna una tabella di DB e torna un numero:

```
WTO> create or replace package pack is
  2  function a return number;
  3  end;
  4  /

Package creato.

WTO> create or replace package body pack is
  2  function a return number is
  3  begin
  4    update emp set empno=0 where 1=2;
  5    return 2;
  6  end;
  7  end;
  8  /

Creato package body.
```

Se cerchiamo di utilizzare la funzione pack.a in una query otteniamo un errore:

```
WTO> select pack.a from dual;

         A
----------
         2
WTO> select pack.a from dual;
select pack.a from dual
       *
ERRORE alla riga 1:
ORA-14551: impossibile eseguire un'operazione DML all'interno di una
query
ORA-06512: a "WTO_ESEMPIO.PACK", line 4
```

Infatti le funzioni PL/SQL possono essere utilizzate in una query solo se non modificano né oggetti di DB né variabili di package.

L'errore però è individuato solo a runtime, quando la select che utilizza A viene eseguita.

Come facciamo a controllare meglio la cosa fin dalla compilazione?

Utilizzando, appunto, PRAGMA RESTRICT_REFERENCES.

Se sappiamo che la funzione A dovrà essere utilizzata in SQL, infatti, possiamo definirla come segue:

```
WTO> create or replace package pack is
  2  function a return number;
  3  pragma restrict_references(a,'WNDS');
  4  end;
  5  /

Package creato.
```

Asserendo che la funzione non modificherà il DB.

A questo punto se un programmatore, non sapendo che la funzione deve essere usata in SQL, cerca di aggiungere nel codice di A un update sul db:

```
WTO> create or replace package body pack is
  2  function a return number is
  3  begin
  4    update emp set empno=0 where 1=2;
  5    return 2;
  6  end;
  7  end;
  8  /

Avvertenza: package body creato con errori di compilazione.

WTO> sho err
Errori in PACKAGE BODY PACK:

LINE/COL ERROR
-------- -------------------------------------------------------
2/1      PLS-00452: Il sottoprogramma 'A' viola il pragma associato
```

Ottiene un errore in fase di compilazione...

La PRAGMA RESTRICT_REFERENCES è attualmente deprecata e potrebbe essere rimossa in prossime versioni di Oracle.

La PRAGMA SERIALLY_REUSABLE comunica al compilatore che alcune variabili di package, che per default restano vive e valorizzate per tutta la sessione, possono essere deallocate dopo il primo utilizzo.

Ciò consente di risparmiare memoria. Facciamo un esempio.

Definiamo un package con una sola variabile numerica var non inizializzata:

```
WTO> create or replace package pack is
  2  var number;
  3  end;
  4  /

Package creato.
```

Se valorizziamo la variabile var in qualunque modo, questa resterà valorizzata per tutta la sessione:

```
WTO> begin
  2  pack.var := 1;
  3  end;
  4  /

Procedura PL/SQL completata correttamente.

WTO> exec dbms_output.put_line('Var='||pack.var);
Var=1

Procedura PL/SQL completata correttamente.
```

Ma se invece utilizziamo la PRAGMA SERIALLY_REUSABLE la variabile resta valorizzata solo all'interno del programma che la inizializza, ed invece vale NULL nelle chiamate successive:

```
WTO> create or replace package pack is
  2  PRAGMA SERIALLY_REUSABLE;
  3  var number;
  4  end;
  5  /

Package creato.

WTO> begin
  2  pack.var := 1;
  3  dbms_output.put_line('Var='||pack.var);
  4  end;
  5  /
Var=1

Procedura PL/SQL completata correttamente.

WTO> exec dbms_output.put_line('Var='||pack.var);
Var=

Procedura PL/SQL completata correttamente.
```

La PRAGMA SERIALLY_REUSABLE è dunque un modo per modificare il comportamento di default delle variabili dei package che è utile ma molto dispendioso in termini di utilizzo di memoria.

La PRAGMA AUTONOMOUS_TRANSACTION dichiara al compilatore che una certa procedura deve girare in una sessione tutta sua, ignorando quindi le modifiche non committate fatte dalla sessione che la chiama.

La somma degli stipendi in EMP è:

```
WTO> select sum(sal) from emp;

  SUM(SAL)
----------
     29025
```

Definiamo due funzioni che fanno esattamente la stessa cosa, leggono e restituiscono la somma degli stipendi di EMP:

```
WTO> create or replace function getsal return number is
  2  s number;
  3  begin
  4    select sum(sal) into s from emp;
  5    return s;
  6  end;
  7  /

Funzione creata.

WTO> create or replace function getsal_AT return number is
  2  PRAGMA AUTONOMOUS_TRANSACTION;
  3  s number;
  4  begin
  5    select sum(sal) into s from emp;
  6    return s;
  7  end;
  8  /

Funzione creata.

WTO> select sum(sal), getsal, getsal_AT
  2  from emp;

  SUM(SAL)     GETSAL  GETSAL_AT
---------- ---------- ----------
     29025      29025      29025
```

A questo punto abbassiamo radicalmente gli stipendi a tutti i dipendenti e vediamo che succede:

```
WTO>  update emp set sal=10;

Aggiornate 14 righe.

WTO> select sum(sal), getsal, getsal_AT
  2  from emp;

  SUM(SAL)     GETSAL  GETSAL_AT
---------- ---------- ----------
       140        140      29025
```

La GETSAL vede i dati modificati e non ancora committati, mentre la GETSAL_AT, definita con la PRAGMA AUTONOMOUS_TRANSACTION, ci fa vedere i dati come erano prima dell'update...

L'unica PRAGMA aggiunta di recente (nella 11g) è la PRAGMA INLINE.

Nella versione 11g di PL/SQL è stata introdotta una nuova funzionalità che l'ottimizzatore può utilizzare per ottenere migliori performances, si chiama *Subprogram Inlining* .

Consiste nel fatto che l'ottimizzatore può decidere (autonomamente oppure a richiesta) di sostituire una chiamata ad un sottoprogramma con una copia del sottoprogramma stesso.

Ad esempio se il codice è

```
declare
totale number;
```

```
begin
 totale := calcola_nominali + calcola_interessi;
end;
```

Dove CALCOLA_NOMINALI e CALCOLA_INTERESSI sono due funzioni definite da:

```
function calcola_nominali return number is
s number;
begin
  select sum(nominale)
    into s
    from operazioni;

  return s;
end;

function calcola_interessi return number is
s number;
begin
  select sum(interessi)
    into s
    from operazioni;

  return s;
end;
```

L'ottimizzatore può trasformare il codice in qualcosa tipo:

```
declare
totale number;
v_calcola_nominali number;
v_calcola_interessi number;
begin
  select sum(nominale)
    into v_calcola_nominali
    from operazioni;

  select sum(interessi)
    into v_calcola_interessi
    from operazioni;

 totale := v_calcola_nominali + v_calcola_interessi;
end;
```

Includendo quindi una copia dei sottoprogrammi nel chiamante.

La PRAGMA INLINE è appunto lo strumento che abbiamo noi per guidare questa nuova funzionalità.

Se non vogliamo che la chiamata di CALCOLA_NOMINALI sia trasformata nel modo che abbiamo appena visto, basta fare:

```
declare
totale number;
begin
 PRAGMA INLINE(calcola_nominali,'NO');
 totale := calcola_nominali + calcola_interessi;
end;
```

Se, invece, vogliamo suggerire al compilatore di usare il subprogram inlining su CALCOLA_NOMINALI facciamo:

```
declare
```

```
totale number;
begin
 PRAGMA INLINE(calcola_nominali,'YES');
 totale := calcola_nominali + calcola_interessi;
end;
```

Il subprogram inlining si comporta in modo diverso in funzione del livello di ottimizzazione che è definito nella variabile di inizializzazione del DB PLSQL_OPTIMIZE_LEVEL.

Se questa variabile vale 2 (che è il default) l'ottimizzatore non effettua mai il subprogram inlining a meno che non sia il programmatore a richiederlo utilizzando PRAGMA INLINE YES.

Se invece PLSQL_OPTIMIZE_LEVEL=3 l'ottimizzatore può decidere autonomamente di ottimizzare utilizzando il subprogram inlining. In questo caso PRAGMA INLINE YES non vincola l'ottimizzatore ma è considerato solo un suggerimento.

9 Cloud e Multitenant Architecture

9.1 C come Cloud

Il cloud computing è, nel settore informatico, la moda del momento. Come sempre succede con i termini alla moda, anche questo è straordinariamente abusato. Viene utilizzato in diversi contesti con differenti significati.

Per dare una definizione generica, e dunque imprecisa, possiamo affermare che il cloud computing è un insieme di tecnologie e pratiche organizzative che consentono di ottimizzare le risorse informatiche (hardware, software, infrastrutture, etc…) in modo da consentirne la fornitura agli utenti interessati a richiesta e, soprattutto, ad un costo più basso.

Facciamo un esempio: ho una serie di documenti, brani musicali, eBook eccetera che voglio poter utilizzare in mobilità, mediante il mio tablet; a casa, su televisore e pc; in ufficio, sul pc aziendale. Per ottenere questo risultato posso dotarmi di un'infrastruttura hardware (HDD portatili, schede SD ecc…) e/o software (programmi di sincronizzazione dei vari dispositivi) oppure posso mettere tutti i miei file in un'infrastruttura cloud fornita da un'azienda e connettere a questa tutti i miei dispositivi, dovunque mi trovi, mediante internet. La seconda soluzione è sicuramente più comoda ed economica, l'azienda che mi fornisce il servizio cloud può permettersi di applicare bassi costi perché fa economia di scala.

L'esempio precedente fa riferimento al cosiddetto *public cloud*. Le risorse messe a disposizione dell'utente sono prese in un ambiente pubblico condiviso con altri utenti.

Per motivi di sicurezza, un'azienda difficilmente può usufruire di un cloud pubblico. Può però trovare comunque beneficio dalla creazione di un cloud privato. Una banca che deve mettere in produzione diversi sistemi informatici può "consolidare" la propria infrastruttura hardware e software creando un pool di risorse che i diversi sistemi condividono. Il consolidamento dei datacenter, ad esempio, è già una realtà per molte aziende. Invece di comprare server dedicati per ogni sistema da mettere in produzione, si fa abbondante utilizzo delle tecnologie di virtualizzazione per avere pochi server più potenti su cui girando molte macchine virtuali.

La lettera “c” di Oracle 12c sta per Cloud e fa riferimento appunto al “private cloud”. La funzionalità principale introdotta in Oracle 12c per il supporto del cloud computing è la Multitenant Architecture, che illustreremo nel prossimo paragrafo.

Come la “i” (internet) di Oracle8i ed Oracle9i e la “g” (grid computing) di Oracle10g ed Oracle11g, anche la “c” (cloud computing) di Oracle 12c è essenzialmente un’operazione di marketing. La multitenant architecture è sicuramente un’utile funzionalità che consente di consolidare e gestire più agevolmente diversi database, ma non si tratta, a mio modesto avviso, di una rivoluzione copernicana.

È importante sottolineare che la Multitenant Architecture di Oracle 12c è un’opzione. Se non è attiva, Oracle 12c continua a comportarsi esattamente come nelle versioni precedenti, ogni db richiede un’istanza dedicata. I database creati in questa modalità “vecchio stile” vengono detti non-CDB.

9.2 Introduzione alla Multitenant Architecture

Una caratteristica di Oracle fino alla versione 8i è stata l’assoluta corrispondenza biunivoca tra Istanza di Database e Database. Ogni istanza un db, ogni db un’istanza. A partire da oracle9i, l’introduzione di Oracle RAC ha dato la possibilità di condividere un unico db tra diverse istanze per ragioni di alta affidabilità. Restava però il vicolo di non poter gestire più di un database con una sola istanza.

L’architettura Multitenant introdotta in Oracle 12c rompe questo vincolo consentendo di collegare più database alla stessa istanza. In particolare, se si decide di avvalersi di quest’architettura, a fronte di un’istanza Oracle si ha un unico database contenitore (*Container database*, CDB) all’interno del quale possono essere inseriti (*plugged*) fino a 252 altri database (P*luggable database*, PDB). Ogni PDB è analogo ad un “vecchio” db Oracle delle precedenti versioni.

L’istanza è unica per il CDB e per tutti i PDB. Lo stesso vale per Redo Log File e Control File, mentre i datafile (ed i tablespace come strutture logiche, inclusi SYSTEM e SYSAUX) sono diversi per ogni singolo PDB e per il CDB.

I vantaggi principali di questa nuova architettura sono:

- La capacità di creare molto velocemente un db come copia di un altro già esistente.
- La capacità di spostare velocemente un PDB da un CDB ad un altro e quindi da un’infrastruttura ad un’altra.
- La capacità di applicare patch a più database in una volta sola.
- La capacità di applicare patch oppure innalzare di versione un PDB semplicemente scollegandolo dal CDB corrente e collegandolo ad un CDB che si trova già alla versione più avanzata.

- Il minor consumo di risorse per singolo PDB, con un risparmio dei costi complessivi.

9.3 Containers in un CDB

Un container è una collezione di schemi, oggetti ed altre strutture collegate all'interno di un CDB. Dal punto di vista dell'utilizzatore, un container è logicamente equivalente ad un database separato. All'interno di un CDB ogni container ha un nome univoco.

Ogni CDB ha un container creato per default noto come Root.

9.3.1 Il container CDB$ROOT

Al container CDB$ROOT (spesso chiamato semplicemente Root) appartengono tutti i PDB. Ogni CDB ha un unico Root che contiene tutti i dati di sistema necessari a gestire i PDB.

Dentro Root non possono essere archiviati dati specifici degli utenti applicativi ma possono essere creati degli schemi/utenti che saranno pubblici, cioè condivisi tra tutti i PDB contenuti nel CDB..

9.3.2 I container PDB

Un PDB, come detto, è logicamente equivalente ad un database Oracle classico (non-CDB database). Ogni PDB è di proprietà di SYS, che è l'utente di sistema definito nel CDB e condiviso tra tutti i PDB. L'utente SYS ha uno schema dedicato in ogni PDB.

Ogni PDB è un container.

I PDB vengono utilizzati essenzialmente per isolare i dati di un'applicazione specifica, seguendo la stessa logica con cui, nelle precedenti versioni di Oracle, si definiva un'istanza dedicata ad un'applicazione.

In un CDB, ogni PDB deve avere un nome univoco, i nomi devono seguire le stesse naming convention applicate ai nomi di servizio db. Il primo carattere deve essere un alfanumerico, i restanti caratteri possono essere alfanumerici oppure underscore. I nomi sono case-insensitive. Ogni PDB ha un servizio db associato che ha lo stesso nome del PDB stesso, tale nome di servizio è utilizzato dai client che intendono collegarsi, tramite listener, al PDB. Collegandosi in questo modo, i client non hanno nessun modo per sapere se sono connessi ad un PDB in un'architettura multitenant oppure ad un database non-CDB classico.

Ovviamente schemi ed oggetti all'interno di diversi PDB possono anche avere nomi in comune, come nell'architettura classica succede per schemi ed oggetti all'interno di db differenti.

L'unico modo per far accedere un utente di un PDB ai dati di un altro PDB è attraverso un database link, come per database separati nell'architettura classica.

9.3.3 Container corrente

In una sessione il container corrente è quello in cui la sessione è in esecuzione. Il container corrente può essere Root oppure un PDB.

La sessione può cambiare container corrente mediante l'istruzione ALTER SESSION SET CONTAINER come nell'esempio che segue. La sessione è connessa all'utente SYSTEM. Per visualizzare il container corrente si utilizza la variabile d'ambiente CON_NAME:

```
SQL> show con_name

CON_NAME
------------------------------
CDB$ROOT
```

Per leggere i servizi accessibili da questo container si può utilizzare la vista dinamica V$SERVICES:

```
SQL> select name, pdb from v$services;

NAME                           PDB
------------------------------ ----------------------------
corsopdb                       CORSOPDB
corsocdbXDB                    CDB$ROOT
corsocdb                       CDB$ROOT
SYS$BACKGROUND                 CDB$ROOT
SYS$USERS                      CDB$ROOT
```

Si tratta dei servizi di default del container Root e del servizio collegato al PDB che abbiamo creato in fase d'installazione.

Per modificare il container corrente si può utilizzare il comando ALTER SESSION SET CONTAINER:

```
SQL> alter session set container=CORSOPDB;

Modificata sessione.

SQL> show con_name

CON_NAME
------------------------------
CORSOPDB
```

Nel container PDB si può accedere solo al servizio relativo al PDB:

```
SQL> select name, pdb from v$services;

NAME                           PDB
------------------------------ ----------------------------
corsopdb                       CORSOPDB
```

9.3.4 Data Dictionary in un CDB

Ogni container ha un Data Dictionary dedicato. Le viste DBA_ elencano tutti gli oggetti presenti nello specifico PDB a cui si è collegati e non danno visibilità degli oggetti definiti negli altri PDB.

Per evitare un inutile spreco di spazio, i dati di dizionario comuni a tutti i PDB vengono conservati solo in Root. In questo caso ogni PDB ha nel suo dizionario un semplice link ad un record presente nel dizionario di Root. In più, ovviamente, ogni PDB ha nel proprio dizionario i dati reali relativi ai propri oggetti.

Ovviamente ci sono viste di dizionario che consentono di leggere trasversalmente le informazioni relative a più container/PDB. Tra queste troviamo le V$ e GV$, già presenti nelle precedenti versioni di Oracle, e le nuove viste il cui nome comincia per CDB_.

In queste viste la Colonna CON_ID consente di individuare il container in cui lo specifico oggetto si trova.

Quando il container corrente è un PDB, le viste CDB_ possono essere lette ma forniscono informazioni solo relative al PDB sesso. Se il container corrente è Root, invece, le viste CDB_ forniscono informazioni relative al container Root ed a tutti i PDB per cui l'utente che esegue la query ha privilegi d'accesso.

9.4 Utenti e ruoli in un CDB

In un CDB alcuni utenti e/o ruoli possono essere comuni a tutti i PDB contenuti nel CDB stesso, Viceversa altri utenti/ruoli sono locali ad uno specifico PDB.

9.4.1 Utenti comuni o locali

Ogni utente di sistema è comune, un utente applicativo può essere comune o locale.

Un utente comune ha la stessa identità e caratteristiche sia in Root che in tutti i PDB presenti al momento, oppure in futuro, nel CDB.

SYS e SYSTEM sono esempi di utenti comuni di sistema.

Possono essere aggiunti altri utenti comuni nel container Root.

Nell'esempio seguente si è connessi con SYSTEM al container Root e si crea un utente comune:

```
SQL> show con_name

CON_NAME
------------------------------
CDB$ROOT

SQL> create user c##corso identified by corso container=ALL;

Utente creato.
```

Il nome degli utenti comuni (quelli aggiuntivi, non quelli di sistema) deve necessariamente cominciare per c##

Gli utenti comuni non devono necessariamente avere gli stessi privilegi su tutti i container, possono connettersi a tutti i container (incluso Root) per cui hanno il privilegio CREATE SESSION.

Come detto, hanno le medesime identità e caratteristiche in tutti i container ma in ogni container possono avere uno schema differente.

Un utente locale esiste solo all'interno di uno specifico PDB.

```
SQL> alter session set container=CORSOPDB;

Modificata sessione.

SQL> create user corso identified by corso;

Utente creato.
```

Il nome di un utente locale non può cominciare con c##.

Un utente locale può accedere ai dati di un altro PDB esclusivamente mediante un database link.

9.4.2 Ruoli comuni o locali

I ruoli di sistema forniti da Oracle (come PUBLIC o DBA) sono comuni. I ruoli creati dagli utenti possono essere comuni o locali.

Un ruolo comune è definito in Root ed in tutti i PDB.

I ruoli comuni aggiuntivi devono seguire le stesse regole di nomenclatura degli utenti comuni aggiuntivi.

Un ruolo locale esiste solo in uno specifico PDB.

Un ruolo, o un privilegio, può essere assegnato ad un utente per un singolo PDB oppure per tutti i container allo stesso tempo. Ciò si realizza mediante la clausola CONTAINER=CURRENT oppure CONTAINER=ALL, rispettivamente, da apporre alla fine dell'istruzione GRANT.

Nell'esempio seguente, connessi con SYSTEM sul CDB, si concede all'utente comune C##CORSO di poter fare select da qualunque tabella nel CDB ed in tutti i PDB:

```
O12c>GRANT SELECT ANY TABLE to c##corso CONTAINER=ALL;

Concessione riuscita.
```

Successivamente ci si sposta sul PDB denominato CORSOPDB e si concede a C##CORSO di agire da DBA solo nel PDB corrente:

```
O12c>alter session set container=CORSOPDB;

Modificata sessione.

O12c>GRANT DBA to c##corso CONTAINER=CURRENT;

Concessione riuscita.
```

Utenti locali non possono assegnare privilegi per tutti i container, quindi non possono utilizzare la clausola CONTAINER=ALL.

Per default l'istruzione GRANT utilizza la clausola CONTAINER=CURRENT, quindi i privilegi vengono assegnati localmente in un unico container.

9.5 Architettura fisica di un CDB

Dal punto di vista dell'architettura fisica, un CDB ed un non-CDB sono molto simili. L'unica differenza è che, in un CDB, i tablespace (ed i corrispondenti data file) sono partizionati per PDB. Ogni PDB ha i suoi e questi non possono essere condivisi da diversi PDB.

Un CDB contiene:

- Un control file
- Un online redo log composto da più file
- Uno o più temp file

Per default, il CDB ha un unico temporary tablespace (TEMP) condiviso da tutti i PDB. È possibile creare ulteriori temporary tablespace sia in Root (utilizzabili da tutti i PDB) sia nei singoli PDB.

- Un insieme di undo data file
- Un insieme di data file di sistema per ogni singolo container (Root e PDB)
- Altri data file creati per esigenze applicative, dedicati agli specifici PDB.

Il fatto che il data dictionary sia, per ogni PDB, archiviato in data file di sistema specifici di quel PDB consente di ottenere la portabilità veloce del PDB tra diversi CDB.

9.6 Operazioni amministrative in un CDB

9.6.1 Avviare ed arrestare Oracle

Per avviare ed arrestare il CDB si utilizzano come al solito i comandi STARTUP e SHUTDOWN.

A CDB avviato, connessi con un utente DBA, per avviare un PDB si può utilizzare il comando:

```
O12c>ALTER PLUGGABLE DATABASE CORSOPDB OPEN READ WRITE;

Database collegabile modificato.
```

E per fermarlo:

```
O12c> alter pluggable database corsopdb close;

Database collegabile modificato.
```

9.6.2 Connettersi al database

Per connettersi al CDB si procede come in un normale non-CDB. Per collegarsi PDB, invece, è necessario utilizzare il nome del servizio ad esso associato. Il tnsnames.ora, quindi, dovrà essere configurato in questo modo:

```
CORSOPDB =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST = localhost)(PORT = 1521))
    )
    (CONNECT_DATA =
       (UR=A)
      (SERVICE_NAME = corsopdb)
    )
  )
```

La configurazione UR=A indica al listener di accettare le connessioni anche se il db è in restricted mode.

A partire da Oracle 10g il listener, per default, non accetta connessioni se il db è in restricted mode. In tale modalità il dba normalmente si collega direttamente dalla macchina che ospita il db, senza passare dal listener (e quindi senza specificare una stringa di connessione). Se si vuole forzare le connessioni in restricted mode attraverso il listener bisogna aggiungere, appunto, il parametro UR=A nel tnsnames.

In Oracle 12c, in architettura multitenant, abbiamo detto che per connettersi al PDB bisogna sempre utilizzare il nome del servizio e quindi passare dal listener. Di conseguenza, senza il parametro UR=A, diventa praticamente impossibile collegarsi direttamente al PDB quando il CDB è in restricted mode.

Per verificare quanto appena detto avviamo il CDB in modalità RESTRICTED:

```
O12c> startup restrict
Istanza ORACLE avviata.

Total System Global Area 1670221824 bytes
Fixed Size                  2403352 bytes
Variable Size            1006633960 bytes
Database Buffers          654311424 bytes
Redo Buffers                6873088 bytes
MOUNT del database eseguito.
Database aperto.
```

Una volta che il CDB è partito in RESTRICTED MODE anche il PDB può partire solo in questa modalità:

```
O12c> alter pluggable database corsopdb open;
alter pluggable database corsopdb open
                         *
ERRORE alla riga 1:
ORA-65054: Impossibile aprire il database collegabile nella modalitÓ
desiderata.

O12c> alter pluggable database corsopdb open restricted;

Database collegabile modificato.
```

A questo punto, nel tnsnames rimuoviamo il parametro UR=A e proviamo a connetterci a corsopdb:

```
O12c> conn corso/corso@corsopdb
ERROR:
ORA-12526: TNS: listener: tutte le istanze appropriate sono in modalitÓ
limitata
```

Oracle ci avvisa che non si può fare perché tutte le istanze sono aperte in restricted mode e il listener non accetta connessioni.

Adesso aggiungiamo al tnsnames l’opzione UR=A e proviamo di nuovo:

```
O12c> conn corso/corso@corsopdb
Connesso.
```

9.6.3 Unplug e trasferimento di un PDB

La funzionalità principale della multitenant architecture è la possibilità di scollegare (unplug) un PDB da un CDB e ricollegarlo (plug) ad un altro con due banali istruzioni.

Per fare unplug di CORSOPDB si esegue il seguente comando:

```
O12c>alter pluggable database corsopdb
  2  unplug into 'd:\oracle\corsopdb.xml'
  3  ;
alter pluggable database corsopdb
*
ERRORE alla riga 1:
ORA-65025: Il database collegabile CORSOPDB non Þ chiuso in tutte le
istanze.
```

L’errore è dovuto al fatto che prima di scollegare un PDB bisogna chiuderlo:

```
O12c>alter pluggable database corsopdb close;

Database collegabile modificato.

O12c>alter pluggable database corsopdb
  2  unplug into 'd:\oracle\corsopdb.xml';

Database collegabile modificato.
```

Una volta scollegato, il PDB non può essere più aperto:

```
O12c>alter pluggable database corsopdb open read write;
alter pluggable database corsopdb open read write
*
ERRORE alla riga 1:
ORA-65086: impossibile aprire/chiudere il database collegabile
```

Per trasferirlo in un altro CDB bisogna copiare nella destinazione il file xml generato col comando di UNPLUG e tutti i datafile, poi eseguire il comando:

```
create pluggable database corsopdb
using 'd:\oracle\corsopdb.xml' NOCOPY;
```

Nel CDB da cui l’abbiamo scollegato non lo possiamo più utilizzare. Il PDB, infatti, è in stato UNPLUGGED:

```
O12c>select PDB_NAME, STATUS from CDB_PDBS;

PDB_NAME             STATUS
-------------------- -------------
CORSOPDB             UNPLUGGED
PDB$SEED             NORMAL
```

Per ricollegarlo bisogna prima droppare il vecchio PDB:

```
O12c>drop pluggable database corsopdb keep datafiles;

Database collegabile eliminato.
```

E poi crearlo nuovamente:

```
O12c>create pluggable database corsopdb
  2  using 'd:\oracle\corsopdb.xml' NOCOPY tempfile reuse;

Database collegabile creato.

O12c>select PDB_NAME, STATUS from CDB_PDBS;

PDB_NAME             STATUS
-------------------- -------------
PDB$SEED             NORMAL
CORSOPDB             NEW
```

Dove la clausola NOCOPY specifica di non copiare i datafile (riutilizziamo quelli precedenti) e la clausola TEMPFILE REUSE indica di riutilizzare anche il file temporaneo (visto che c’è…).

A questo punto si può riaprire:

```
O12c>alter pluggable database corsopdb open read write;

Database collegabile modificato.

O12c>select PDB_NAME, STATUS from CDB_PDBS;

PDB_NAME             STATUS
-------------------- -------------
PDB$SEED             NORMAL
CORSOPDB             NORMAL
```

E tutto è tornato esattamente come prima.

9.6.4 Duplicazione di un PDB

Per duplicare un PDB si può utilizzare la semplicissima istruzione:

```
O12c>create pluggable database CORSOPDB_COPIA from CORSOPDB;
create pluggable database CORSOPDB_COPIA from CORSOPDB
                                                     *
ERRORE alla riga 1:
ORA-65016: Þ necessario specificare FILE_NAME_CONVERT
```

L'errore indica che bisogna specificare come gestire i nomi dei datafile. I datafile presenti in questo momento sono quelli relativi al CDB e quelli relativi a CORSOPDB:

```
O12c>select name from v$datafile;

NAME
--------------------------------------------------------------
D:\ORACLE\ORADATA\CORSOCDB\SYSTEM01.DBF
D:\ORACLE\ORADATA\CORSOCDB\PDBSEED\SYSTEM01.DBF
D:\ORACLE\ORADATA\CORSOCDB\SYSAUX01.DBF
D:\ORACLE\ORADATA\CORSOCDB\PDBSEED\SYSAUX01.DBF
D:\ORACLE\ORADATA\CORSOCDB\UNDOTBS01.DBF
D:\ORACLE\ORADATA\CORSOCDB\USERS01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB\SYSTEM01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB\SYSAUX01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB\SAMPLE_SCHEMA_USERS01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB\CORSO01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB\TEST.DBF
```

Bisogna specificare come modificare i nomi dei datafile di CORSOPDB per creare quelli di CORSOPDB_COPIA.

La clausola FILE_NAME_CONVERT è molto semplice, prevede l'indicazione di due stringhe. Nei nomi dei datafile Oracle utilizzerà la seconda stringa al posto della prima:

```
O12c>create pluggable database CORSOPDB_COPIA
  2  from CORSOPDB
  3  file_name_convert=('CORSOPDB','CORSOPDB_COPIA');
create pluggable database CORSOPDB_COPIA
*
ERRORE alla riga 1:
ORA-65081: Il database o il database collegabile non Þ aperto in
modalitÓ di
sola lettura
```

L'errore è dovuto al fatto che CORSOPDB deve essere aperto in sola lettura per poterlo copiare:

```
O12c>alter pluggable database corsopdb close;
Database collegabile modificato.

O12c>alter pluggable database corsopdb open read only;
Database collegabile modificato.

O12c>create pluggable database CORSOPDB_COPIA
  2  from CORSOPDB
  3  file_name_convert=('CORSOPDB','CORSOPDB_COPIA');

Database collegabile creato.

O12c>select PDB_NAME, STATUS from CDB_PDBS;

PDB_NAME             STATUS
-------------------- -------------
CORSOPDB_COPIA       NEW
PDB$SEED             NORMAL
CORSOPDB             NORMAL
```

Il nuovo PDB è stato creato, apriamolo e poi guardiamo i nomi dei datafile:

```
O12c>alter pluggable database corsopdb_copia open read write;

Database collegabile modificato.

O12c>select PDB_NAME, STATUS from CDB_PDBS;

PDB_NAME             STATUS
-------------------- -------------
CORSOPDB_COPIA       NORMAL
PDB$SEED             NORMAL
CORSOPDB             NORMAL

O12c>select name from v$datafile;

NAME
--------------------------------------------------------------
D:\ORACLE\ORADATA\CORSOCDB\SYSTEM01.DBF
D:\ORACLE\ORADATA\CORSOCDB\PDBSEED\SYSTEM01.DBF
D:\ORACLE\ORADATA\CORSOCDB\SYSAUX01.DBF
D:\ORACLE\ORADATA\CORSOCDB\PDBSEED\SYSAUX01.DBF
D:\ORACLE\ORADATA\CORSOCDB\UNDOTBS01.DBF
D:\ORACLE\ORADATA\CORSOCDB\USERS01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB\SYSTEM01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB\SYSAUX01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB\SAMPLE_SCHEMA_USERS01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB\CORSO01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB\TEST.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB_COPIA\SYSTEM01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB_COPIA\SYSAUX01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB_COPIA\SAMPLE_SCHEMA_USERS01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB_COPIA\CORSO01.DBF
D:\ORACLE\ORADATA\CORSOCDB\CORSOPDB_COPIA\TEST.DBF

16 righe selezionate.
```

Come si vede, i datafile di CORSOPDB_COPIA hanno gli stessi nomi di quelli di CORSOPDB con l'unica eccezione che la stringa CORSOPDB è stata sostituita da CORSOPDB_COPIA

Le stesse istruzioni possono essere eseguite per copiare il PDB attraverso un database link:

```
O12c>create pluggable database CORSOPDB_LINK
  2  from CORSOPDB@dbl
  3  file_name_convert=('CORSOPDB','CORSOPDB_LINK');
```

9.6.5 PDB Trigger

L'introduzione dei PDB ha fatto sì che siano stati definiti anche nuovi eventi gestibili con db trigger. L'evento AFTER CLONE scatta quando un PDB viene copiato, l'evento BEFORE UNPLUG quando il PDB viene scollegato.

A titolo d'esempio definiamo un database trigger sul PDB CORSOPDB_COPIA che ne impedisca lo scollegamento tra le 23 e le 23.59.59. Connettiamoci con SYSTEM al CDB:

```
O12c>conn system/corso
Connesso.
```

```
O12c>show con_name

CON_NAME
------------------------------
CDB$ROOT
```

E verifichiamo quali PDB sono disponibili:

```
O12c>select pdb_name, status from cdb_pdbs;

PDB_NAME                       STATUS
------------------------------ -------------
CORSOPDB_COPIA                 NORMAL
PDB$SEED                       NORMAL
CORSOPDB                       NORMAL
```

Apriamo CORSOPDB_COPIA e connettiamoci ad esso (ricordiamo che bisogna definire la connessione nel TNSNAMES.ORA prima):

```
O12c>alter pluggable database CORSOPDB_COPIA open read write;

Database collegabile modificato.

O12c>conn system/corso@CORSOPDB_COPIA
Connesso.
```

Poi definiamo il trigger.

```
O12c>create trigger no_unplug_time
  2  before unplug on pluggable database
  3  begin
  4    if to_char(sysdate,'HH24')='23' then
  5         raise_application_error(-20001,
  6         'Unplug non ammesso dalle 23 alle 23.59');
  7    end if;
  8  end;
  9  /
Trigger creato.
```

Se l'ora della SYSDATE è 23, il trigger solleva un errore, di conseguenza andrà in errore l'operazione che lo ha scatenato ed il comando di UNPLUG fallirà:

```
O12c>conn sys as sysdba
Immettere la password:
Connesso.

O12c>alter pluggable database corsopdb_copia close;

Database collegabile modificato.

O12c>alter pluggable database corsopdb_copia
  2  unplug into 'd:\oracle\corsopdb_copia.xml';
alter pluggable database corsopdb_copia
*
ERRORE alla riga 1:
ORA-00604: errore riscontrato in SQL ricorsivo livello 1
ORA-20001: Unplug non ammesso dalle 23 alle 23.59
ORA-06512: a line 3


O12c>select pdb_name, status from cdb_pdbs;

PDB_NAME                       STATUS
```

```
------------------------------ -------------
CORSOPDB_COPIA                 NORMAL
PDB$SEED                       NORMAL
CORSOPDB                       NORMAL
```

Allo stesso modo possiamo prevenire la copia del PDB CORSOPDB_COPIA. Ci colleghiamo al CDB ed apriamo il PDB:

```
O12c>alter pluggable database corsopdb_copia open read write;

Database collegabile modificato.
```

Ci colleghiamo al PDB e creiamo il trigger:

```
O12c>conn system/corso@corsopdb_copia
Connesso.
O12c>create trigger no_copy
  2  AFTER CLONE on pluggable database
  3  begin
  4    raise_application_error(-20001,
  5    'Questo PDB non può essere copiato!!!');
  6  end;
  7  /

Trigger creato.
```

Il trigger manderà in errore qualunque tentativo di copia:

```
O12c>conn sys as sysdba
Immettere la password:
Connesso.

O12c>alter pluggable database corsopdb_copia close;

Database collegabile modificato.

O12c>alter pluggable database corsopdb_copia open read only;

Database collegabile modificato.

O12c>create pluggable database CORSOPDB_COPIA2
  2  from CORSOPDB_COPIA
  3  file_name_convert=('CORSOPDB_COPIA','CORSOPDB_COPIA2');
create pluggable database CORSOPDB_COPIA2
*
ERRORE alla riga 1:
ORA-00604: errore riscontrato in SQL ricorsivo livello 1
ORA-20001: Questo PDB non può essere copiato!!!
ORA-06512: a line 2
```

Il problema è che, poiché l'evento è un AFTER CLONE, esso scatta solo dopo che la copia è stata effettuata. Di conseguenza il PDB CORSOPDB_COPIA2 esiste ma è inutilizzabile:

```
O12c>select pdb_name, status from cdb_pdbs;

PDB_NAME                       STATUS
------------------------------ -------------
CORSOPDB_COPIA                 NORMAL
PDB$SEED                       NORMAL
CORSOPDB                       NORMAL
```

```
CORSOPDB_COPIA2                    UNUSABLE
```

E sul filesystem sono stati creati i relativi data file e temp file:

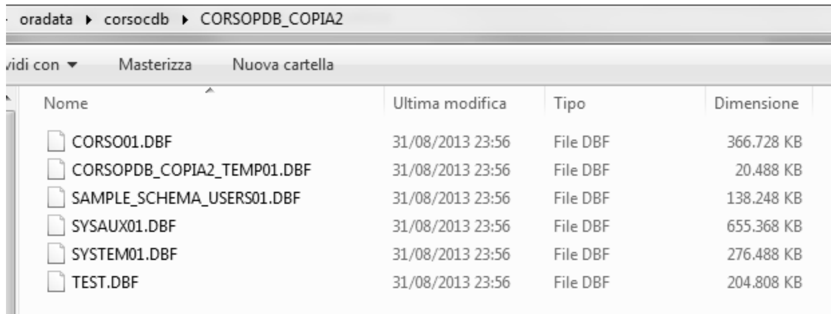


Figura 9-1 Datafile e Tempfile del PDB

10 Prerequisiti

Quest'ultimo capitolo raccoglie alcune definizioni e concetti di base utilizzati nel resto del corso. L'obiettivo è rendere il più possibile facile la lettura del testo, riducendo al minimo la necessità di accedere ad altre fonti. Ovviamente i concetti di base sono qui trattati in maniera molto sintetica, chi volesse approfondire non avrà difficoltà a trovare risorse in rete oppure in libreria.

In tutto il manuale è stato utilizzato il simbolo , seguito da un numero di paragrafo, per indicare i concetti introdotti in questo capitolo.

10.1 Database e database relazionali

10.1.1 database

Un database (o più precisamente un DBMS, *database management system)* è un software che consente di archiviare dati di qualunque tipo. Il database deve essere munito di funzionalità di ricerca che consentano di estrarre le informazioni in maniera semplice quando serve.

L'informatica, come suggerisce il suo nome di origine francese (*informatique = information automatique*), è la scienza che si occupa della gestione automatica delle informazioni. Un sistema informatico, dovendo gestire informazioni, cioè dati, non può prescindere dalla presenza di un sistema ben organizzato che consenta di archiviare e ricercare i dati, cioè da un database. Dunque pressoché qualunque sistema informatico, da quello utilizzato nella videoteca sotto casa a quelli utilizzati dalle più grandi aziende al mondo, include tra le proprie componenti un database.

Ovviamente non tutti i database sono uguali, varie teorie ed implementazioni pratiche hanno arricchito la storia dell'informatica, ma la categoria di database che ha riscosso i maggiori successi è quella dei cosiddetti "database relazionali" definiti da E. F. Codd nel 1970 in un articolo dal titolo "A Relational Model of Data for Large Shared Data Banks" ("Un modello relazionale dei dati per grandi banche dati condivise", una traduzione in italiano dell'articolo è disponibile all'indirizzo web http://oracleitalia.wordpress.com/codd/).

10.1.2 database relazionali

Un database relazionale (RDBMS, *relational database management system*) è un software che consente di organizzare i dati in strutture tabellari collegate tra loro. Un esempio chiarirà il concetto. Immaginiamo di dover archiviare nome, cognome, numero di telefono dei nostri clienti e numero, importo e data scadenza delle fatture che abbiamo emesso ai clienti. Questi dati si possono strutturare in forma relazionale come segue:

Tabella CLIENTI

Codice Cliente	**Nome**	**Cognome**	**Telefono**
111111	Alberto	Rossi	0666778899
222222	Giovanni	Verdi	0233445566
333333	Stefano	Bianchi	0819988774

Tabella FATTURE

Numero Fattura	**Importo**	**Data scadenza**	**Codice Cliente**
1	1000,00	01/10/2010	111111
2	1200,00	01/11/2010	111111
3	800,00	21/10/2010	222222
4	1000,00	31/10/2010	222222
5	2000,00	01/12/2010	222222

Ogni tabella descrive un'entità, cioè un insieme di oggetti omogenei le cui caratteristiche sono interessanti per il nostro sistema. Ogni tabella ha un numero fisso di colonne che rappresentano le informazioni dell'entità che vogliamo gestire, ed un numero teoricamente illimitato di possibili *righe* (o *record)* ognuna delle quali descrive un particolare oggetto: un cliente, una fattura, ecc...

Per consentire l'individuazione univoca di ogni singolo oggetto/riga, solitamente ogni tabella è munita di una chiave primaria (in inglese *primary key*). Una chiave primaria non è altro che una colonna, oppure un insieme di colonne, che non può assumere lo stesso valore in righe differenti della tabella. Nel nostro esempio la chiave primaria della tabella CLIENTI è il campo "Codice Cliente", la chiave primaria della tabella FATTURE è il numero della fattura. Non ci possono essere due clienti con lo stesso codice, non ci possono essere due fatture con lo stesso numero.

Le tabelle possono essere legate tra loro da relazioni, nel nostro esempio la tabella delle fatture è legata alla tabella dei clienti perché in ogni

fattura è presente il dato “Codice Cliente” mediante il quale è possibile ricavare, in maniera univoca poiché il Codice Cliente è chiave primaria nella tabella CLIENTI, le informazioni del cliente a cui è stata emessa la fattura. La colonna “Codice Cliente” presente nella tabella FATTURE è detta chiave esterna (*foreign key*). La chiave esterna così definita fa riferimento alla chiave primaria definita sulla tabella CLIENTI. Una chiave esterna e la chiave primaria a cui essa fa riferimento sono di fatto lo stesso dato, nel nostro esempio il codice del cliente.

I migliori database relazionali garantiscono automaticamente l’integrità referenziale, cioè

- non è possibile assegnare un valore specifico ad una chiave esterna se quello stesso valore non è stato già definito tra i valori della chiave primaria a cui essa fa riferimento. Non possiamo assegnare il valore 555555 al codice cliente di una fattura perché non esiste un cliente in CLIENTI che ha quel codice.
- Non è possibile cancellare una riga da una tabella se ci sono chiavi esterne in altre tabelle che fanno riferimento a quello specifico valore. Nel nostro esempio non sarebbe possibile cancellare i clienti 111111 e 222222 perché essi hanno fatture collegate, ma sarebbe possibile cancellare il cliente 333333 perché non ha fatture collegate.

10.1.3 DBA

Il DBA (database administrator) è un professionista dell’informatica che si occupa di installare, aggiornare e gestire uno o più database. Tutte le grandi organizzazioni hanno alle proprie dipendenze, direttamente o mediante contratti di consulenza stipulati con altre aziende, un gruppo di DBA che gestisce i database aziendali. Si tratta di una figura professionale molto critica visto che il vero patrimonio di tante grandi aziende sta soprattutto nei dati che esse gestiscono.

10.2 Progettazione di un db relazionale

Il processo di progettazione di un database relazionale passa solitamente attraverso i seguenti passaggi:

- Definizione del modello concettuale mediante la creazione di un diagramma entità/relazioni.
- Definizione di un modello logico mediante il processo di normalizzazione.
- Definizione del modello fisico in funzione dello specifico database che si intende utilizzare.

10.2.1 Modello entità/relazioni

Il modello di disegno basato su diagrammi entità/relazioni è stato introdotto da Peter Chen nell’articolo “*The entity-relationship model - toward a unified view of data*” pubblicato nel marzo del 1976. Questo modello è

diventato, negli anni, uno standard *de facto* per la realizzazione del modello concettuale dei dati di un database. Il modello si basa sull'individuazione delle entità del sistema e delle relazioni che collegano le diverse entità. Per ogni entità vanno poi specificati gli attributi, cioè i singoli dati da gestire, e gli identificatori, cioè le potenziali chiavi primarie ed univoche. Anche le relazioni in taluni casi possono avere degli attributi.

10.2.2 Normalizzazione

Completato il modello concettuale bisogna trasformarlo in uno schema tabellare relazionale. Una prima trasformazione grossolana può essere effettuata applicando alcune regole fisse allo schema concettuale, ad esempio:

- Da ogni entità si ottiene una tabella,
- da ogni relazione avente attributi si ottiene una tabella,
- le altre relazioni diventano chiavi esterne.

Si individua, in questo modo, un insieme di tabelle relazionali. Tali tabelle però potrebbero presentare forti limitazioni oppure delle ridondanze di dati. Per eliminare le ridondanze si applica il processo di *normalizzazione*. Tale processo consiste nella trasformazione graduale delle tabelle in modo che queste rispettino le regole imposte dalle forme normali. Tra le molte che sono state enunciate dagli esperti di analisi dei dati, nel seguito saranno descritte le forme normali che si considerano indispensabili per ottenere un database a bassa ridondanza: la prima, la seconda e la terza forma normale.

Per ragionare su un esempio concreto si consideri la seguente tabella:

Tabella CLIENTI – Versione 1

Codice	Cognome	Indirizzo	Comune	Prov.	Telefono
111111	Rossi	Via Rossi, 2	Pomezia	RM	Casa 0666778899 Cell 3351234567
222222	Verdi	Via Verdi, 1	Rho	MI	Casa 0233445566 Cell 3387654321
333333	Bianchi	Via Bianchi, 3	Casoria	NA	Casa 0819988774 Cell 3491234567
444444	Gialli	Via Gialli, 1	Rho	MI	Cell 3177654321

Osservando la colonna "Telefono" si nota che essa contiene più informazioni per ogni cella: il telefono di casa ed il cellulare. Questo implica svariati problemi, primo tra tutti la difficoltà di eseguire ricerche su quel dato. Un attributo che contiene diverse informazioni si dice *non atomico* ed una tabella che contiene attributi non atomici si dice che viola la *prima forma normale*.

Una tabella per essere in prima forma normale non deve contenere attributi non atomici. Per porre in prima forma normale la tabella clienti la si può ridisegnare come segue:

Tabella CLIENTI – Versione 2

Codice	Cognome	Indirizzo	Comune	Prov.	Tel. Casa	Tel. Cell
111111	Rossi	Via Rossi, 2	Pomezia	RM	0666778899	3351234567
222222	Verdi	Via Verdi, 1	Rho	MI	0233445566	3387654321
333333	Bianchi	Via Bianchi, 3	Casoria	NA	0819988774	3491234567
444444	Gialli	Via Gialli, 1	Rho	MI		3177654321

La versione 2 della tabella CLIENTI è in prima forma normale.

Una tabella si dice essere in *seconda forma normale* se è in prima forma normale e non esistono attributi della tabella che dipendono solo da una parte della chiave primaria. Chiariamo il concetto con un esempio, si consideri la tabella

Tabella ORDINI – Versione 1

Num. Ordine	Cod. Prodotto	data	Descr. prodotto	Quantità
111111	XY123	1/9/2010	Bulloni mis. 8	200
111111	AZ321	1/9/2010	Viti da 5mm	100
222222	AS333	3/9/2010	Viti da 7mm	130
333333	AZ321	5/9/2010	Viti da 5mm	230
444444	XY123	7/9/2010	Bulloni mis. 8	150

La tabella ORDINI contiene degli ordini, per ogni record sono specificati il numero e la data dell'ordine, il codice, la descrizione e la quantità dell'articolo ordinato. Per ogni ordine esistono in tabella tante righe quanti sono gli articoli ordinati. La chiave primaria della tabella è costituita dalla coppia (Numero Ordine, Codice Prodotto) che è univoca, mentre sia il numero ordine che il codice prodotto presi singolarmente non sono univoci in tabella e dunque non potevano costituire, presi singolarmente, chiave primaria. La ridondanza di dati risulta evidente quando, ad esempio, si volesse aggiornare la descrizione dell'articolo XY123. Sarebbe, infatti, necessario aggiornare tutti i record che fanno riferimento a tale codice. Ciò è causato dal fatto che la descrizione del prodotto ordinato non dipende dall'intera chiave della tabella (Numero Ordine, Codice Prodotto) ma solo dal codice del prodotto, tutte le righe riportanti lo stesso codice prodotto avranno necessariamente anche la stessa descrizione. Lo stesso vale per la data ordine, essa dipende solo dal numero ordine, ma non dal prodotto ordinato. La quantità ordinata invece dipende dall'intera chiave perché è un dato specifico riferito ad un particolare prodotto in un particolare ordine.

Per eliminare tali ridondanze e portare la tabella in seconda forma normale è necessario spaccarla in più tabelle secondo la seguente regola: tutti gli attributi dipendenti da una porzione della chiave devono essere eliminati dalla tabella ed inseriti in una nuova tabella avente come chiave primaria la porzione di chiave. Nel nostro esempio c'era una doppia

dipendenza parziale, della descrizione dal codice articolo e della data dal numero ordine, di conseguenza saranno individuate due nuove tabelle:

Tabella DETTAGLIO ORDINI

Num. Ordine	Cod. Prodotto	Quantità
111111	XY123	200
111111	AZ321	100
222222	AS333	130
333333	AZ321	230
444444	XY123	150

Tabella ORDINI

Num. Ordine	data
111111	1/9/2010
222222	3/9/2010
333333	5/9/2010
444444	7/9/2010

Tabella PRODOTTI

Cod. Prodotto	Descr. prodotto
XY123	Bulloni mis. 8
AZ321	Viti da 5mm
AS333	Viti da 7mm

Le nuove tabelle avranno ovviamente un numero minore di righe rispetto a quella da cui si è partiti perché ogni articolo ed ogni ordine sarà in esse censito una sola volta. Tutte queste tabelle sono in seconda forma normale.

Un'altra forma di ridondanza si verifica quando un attributo dipende dalla chiave primaria per mezzo di un altro attributo. In questo caso si dice che la dipendenza dell'attributo dalla chiave è *transitiva*. Una tabella è in terza forma normale quando è in seconda forma normale e non esistono dipendenze transitive dalla chiave. Chiariamo con un esempio. Si riconsideri la tabella CLIENTI – Versione 2 ottenuta in applicazione della prima forma normale. In quella tabella si consideri il campo "Provincia". Sicuramente la provincia di residenza è un attributo dipendente dal codice cliente, clienti diversi abitano in generale in province diverse, sebbene in particolare alcuni clienti abitino nella stessa provincia. La provincia però è anche dipendente dal comune, che a sua volta dipende dal codice cliente. Se due clienti abitano nello stesso comune sicuramente abitano anche nella stessa provincia, visto che fissato il comune la provincia si ottiene di conseguenza. Questa situazione è detta dipendenza transitiva, la provincia dipende sì dal

cliente ma non direttamente, bensì per mezzo del comune. La tabella non è dunque in terza forma normale. Per normalizzarla è necessario definire una nuova tabella che abbia come chiave l'attributo che fa da tramite nella dipendenza transitiva (nell'esempio il comune) e come attributi tutti gli attributi che erano transitivamente dipendenti (nell'esempio la provincia)

Tabella CLIENTI – Versione 3

Codice	Cognome	Indirizzo	Codice Comune	Tel. Casa	Tel. Cell
111111	Rossi	Via Rossi, 2	G811	0666778899	3351234567
222222	Verdi	Via Verdi, 1	H264	0233445566	3387654321
333333	Bianchi	Via Bianchi, 3	B990	0819988774	3491234567
444444	Gialli	Via Gialli, 1	H264		3177654321

Tabella COMUNI

Codice Comune	Nome Comune	Prov.
G811	Pomezia	RM
H264	Rho	MI
B990	Casoria	NA

Oltre alla normalizzazione è stato anche introdotto il codice comune solo per una questione di comodità. È infatti sconsigliabile utilizzare come chiave di una tabella un campo descrittivo (il nome del comune). Le due tabelle ottenute sono in terza forma normale.

10.2.3 Definizione dello schema fisico

Completato il processo di normalizzazione si è definito lo schema logico del database relazionale. Questo schema è valido indipendentemente dallo specifico software di database che si intende utilizzare. Alcune scelte progettuali però dipendono dallo specifico database, ad esempio la definizione dei tipi di dato, dei tablespace e dei parametri di memorizzazione dei dati. Nel seguito di questo corso saranno approfonditi tutti questi aspetti facendo sempre riferimento soltanto al database Oracle. Le scelte progettuali specifiche del db che si utilizza fanno evolvere lo schema logico in schema fisico e chiudono la fase di progettazione.

10.3 Architettura di un computer

Per installare ed utilizzare un database è necessario avere a disposizione un calcolatore. Per fare un po' di prove ed imparare può essere sufficiente un personal computer come quelli che si trovano in quasi tutte le case. Per scopi professionali, invece, vengono utilizzati computer particolarmente potenti detti *server*. In questo paragrafo saranno introdotte le strutture di base di un calcolatore, sia esso personal computer o server.

10.3.1 Architettura Hardware

Con grande approssimazione si può affermare che un computer sia formato dalle seguenti componenti elettroniche (Hardware).

- Un processore (CPU, Central Processing Unit) che è in grado di compiere ad altissima velocità poche operazioni elementari come accedere alla memoria per leggere o scrivere un dato. I computer moderni sono spesso dotati di più CPU in modo da poter compiere più operazioni contemporaneamente.
- Una memoria volatile (RAM, Random Access Memory), limitata nella dimensione ma che garantisca una velocità di accesso molto alta. Volatile significa che i dati in essa immagazzinati andranno persi allo spegnimento del computer.
- Una memoria fissa (Hard Disk Drive, disco rigido) molto più ampia ma meno veloce della RAM in cui i dati possano restare immagazzinati per un tempo indeterminato. La memoria fissa può essere interna o esterna rispetto al calcolatore e solitamente utilizza una tecnologia basata su dischi magnetici.
- Una serie di periferiche di input/output che consentano all'essere umano o ad altri computer di interagire con il calcolatore: tastiera, monitor, mouse, porte di comunicazione, schede di rete, stampanti, ecc.
- Una rete interna di interconnessioni (BUS) che consentano al processore di controllare tutte le periferiche ed utilizzare le memorie.

10.3.2 Architettura Software

La CPU ha la capacità di compiere ad altissima velocità poche semplici operazioni elementari. Per compiere un'operazione un po' più complessa, ma che a prima vista sembrerebbe semplice, ad esempio la copia di un file, bisogna fornire al processore l'elenco delle istruzioni elementari da eseguire guidandolo così, passo per passo, nell'esecuzione. Un tale elenco di istruzioni si chiama programma e deve essere necessariamente scritto in un linguaggio comprensibile al processore. Per essere in grado di eseguire le operazioni leggermente più complesse di quelle elementari, come la gestione delle periferiche, la gestione dei file e tante altre, il computer deve essere dotato di un insieme di programmi (Software) di base che prende il nome di Sistema Operativo.

I sistemi operativi più diffusi sono Microsoft Windows, nelle sue varie versioni per personal computer o server; Unix, anch'esso in varie versioni ed utilizzato soprattutto per i server; Linux, una versione per personal computer di Unix che ultimamente si sta diffondendo anche nei server.

All'interno del sistema operativo si installano ed utilizzano le applicazioni, prodotti software utili per la realizzazione di specifiche attività. Esempi di applicazione sono

- gli strumenti di Office Automation come Microsoft Office ed OpenOffice, che consentono di scrivere documenti, fogli di calcolo e presentazioni;
- i browser internet come Internet Explorer, Mozilla Firefox e Google Chrome che consentono di navigare in internet;
- i database come Oracle, Microsoft SQL Server ed IBM DB2 che consentono di archiviare dati;
- gli IDE (Integrated Development Environment) come Eclipse o NetBeans, che consentono di creare software utilizzando uno dei tanti linguaggi di programmazione esistenti.

10.4 Memorizzazione dei dati

10.4.1 I sistemi binario ed esadecimale

Il sistema binario è un sistema di numerazione che consente di rappresentare qualunque numero utilizzando solo due cifre, 0 ed 1, invece delle dieci a cui siamo abituati nel sistema decimale che utilizziamo tutti i giorni. Nel sistema decimale ogni cifra del numero è implicitamente moltiplicata per una potenza di dieci a partire da 1 (=10^0) sulla destra e procedendo incrementando le potenze verso sinistra.

Il numero 123 rappresenta la somma

$$1\times10^2+2\times10^1+3\times10^0$$

Nel sistema binario, il numero 123 deve essere ottenuto come somma di potenze di due. Poiché 123 = 64+32+16+8+2+1, si può scrivere:

$$123=1\times2^6+1\times2^5+1\times2^4+1\times2^3+0\times2^2+1\times2^1+1\times2^0$$

Sottintendendo le potenze, come facciamo solitamente nel sistema decimale, si ottiene la rappresentazione binaria del numero 123:

1111011

Poiché nel sistema binario sono disponibili solo due cifre i numeri crescono molto rapidamente di lunghezza, ma è solo un problema di rappresentazione grafica: cinque dita sono sempre cinque, sia che le si rappresenti con la cifra “5” che con la stringa di cifre “101”!

Esattamente nello stesso modo si ragiona con il sistema esadecimale dove le cifre sono ben sedici, da 0 a 9 più le lettere A, B, C, D, E, F. Ovviamente nel caso del sistema esadecimale i numeri avranno una rappresentazione più corta di quella decimale. La tabella seguente rappresenta i numeri da uno a venti in sistema decimale, binario ed esadecimale. Anche negli altri sistemi di numerazione, come in quello decimale, gli zeri aggiunti alla sinistra del numero non ne cambiano il valore.

Sistema Decimale	Sistema Binario	Sistema Esadecimale
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14

10.4.2 Dal bit al TeraByte

L'unità minima di memorizzazione delle informazioni in un computer è il *bit* o *binary digit*. Si tratta, come dice il suo nome, di una cifra del *sistema binario*.

Come detto in precedenza qualunque numero può essere rappresentato in sistema binario, quindi il fatto di avere a disposizione solo due cifre non influenza i numeri che possono essere rappresentati nel computer.

I bit sono raggruppati logicamente in gruppi di 8 per formare i Byte. Quindi un Byte (abbreviato B) può rappresentare tutti i numeri interi che vanno da zero "00000000" a 255 "11111111".

Nei normali sistemi di misura il prefisso kilo indica la moltiplicazione per mille. Ad esempio un chilogrammo equivale a mille grammi. In informatica la cosa è un po' più complicata. Poiché si ragiona in sistema binario, i moltiplicatori privilegiati non sono le potenze di dieci (10, 100, 1000) ma le potenze di 2. Per questo motivo un KiloByte (abbreviato KB) non equivale a 1000 Byte bensì a 1024 Byte che è la potenza di 2 più vicina al numero mille.

Così procedendo 1024 KiloByte formano un MegaByte (abbreviato MB). 1024 MegaByte formano un GigaByte (GB) e 1024 GigaByte formano un TeraByte (TB). Quindi un TeraByte è formato dalla bellezza di 8x1024x1024x1024x1024=8.796.093.022.208 cifre binarie.

10.4.3 La tabella ASCII

Nel paragrafo precedente si è visto come vengono rappresentati i numeri interi nella memoria di un computer. La maggior parte delle volte però ci interessa memorizzare tutti i caratteri, non solo i numeri. Serve dunque un modo per associare ogni carattere (lettere minuscole, maiuscole, cifre, segni di punteggiatura, caratteri speciali eccetera) ad un numero intero che sia poi rappresentabile con una stringa di bit.

Questa funzione è svolta nei sistemi Windows ed Unix dalla tabella ASCII (American Standard Code for Information Interchange). Si tratta di una tabella che associa ogni numero da 0 a 255 (quindi un Byte) ad un carattere.

Se in un sistema basato su tabella ASCII si intende memorizzare la lettera M di fatto sarà memorizzata la stringa di bit “01001101”.

Ovviamente 255 codici sono ampiamente sufficienti per individuare tutti i caratteri utilizzati nelle lingue occidentali, ma sono pochi per i caratteri di molte altre lingue del mondo. Per questo motivo la codifica ASCII è stata estesa con la codifica UNICODE che è basata attualmente su 21 cifre binarie e consente di rappresentare circa un milione di caratteri.

Non tutti i sistemi operativi sono basati sulla codifica ASCII/Unicode. I mainframe IBM, ad esempio, utilizzano la codifica EBCDIC, anch’essa basata su otto bit ma associati differentemente ai caratteri da rappresentare. La stringa di bit “01001101” che in ASCII rappresenta la lettera “M”, in EBCDIC rappresenta la parentesi tonda aperta “(“.

10.5 Rappresentazioni numeriche

Abbiamo visto che ogni byte rappresenta un numero intero compreso tra zero e 255. Ovviamente abbiamo bisogno di una tecnica che ci consenta di rappresentare anche numeri decimali composti da una parte intera ed una parte frazionaria. Le tecniche di rappresentazione dei numeri più diffuse sono le seguenti.

10.5.1 Virgola fissa

La prima tecnica, più semplice, consiste nel conservare tutti i numeri come interi e fissare un fattore di scala valido per tutti i numeri conservati. Questa tecnica è detta rappresentazione a *virgola fissa*. Ad esempio ipotizziamo di voler memorizzare gli importi degli stipendi mensili degli impiegati di un’azienda. Ipotizziamo che gli importi abbiano al massimo due cifre decimali ed il massimo stipendio sia 20.000 euro. Fissato il fattore di scala 100, tutti gli stipendi possono essere memorizzati con gli interi che vanno da 1 (che rappresenta 0,01 euro) a 2000000 (che rappresenta 20.000,00 euro). In pratica assumiamo sempre l’esistenza di due cifre decimali e moltiplichiamo per cento prima di memorizzare l’importo. È chiaro che quando andremo a leggere l’importo dalla memoria del computer dovremo avere cura di dividere per il fattore di scala in modo da ottenere l’importo esatto con le cifre decimali corrette.

10.5.2 Virgola mobile

Un'altra tecnica di memorizzazione dei numeri consiste nel non assumere un fattore di scala fisso ma determinarlo per ogni numero che si archivia. Un qualunque numero razionale *x* può essere espresso nella notazione esponenziale

$$x = mantissa \times base^{esponente} \qquad (1)$$

Dove la mantissa è un numero maggiore o uguale di 0,1 e minore di 1, la base può essere scelta liberamente (le tre basi più utilizzate sono 2, 10 e 16) e l'esponente è un numero intero relativo.

Accettando l'imprecisione dovuta ad un inevitabile arrotondamento, anche tutti i numeri reali possono essere espressi in notazione esponenziale per di più utilizzando un limitato numero di cifre per rappresentare la mantissa. La rappresentazione a *virgola mobile* consiste dunque nel fissare la base ed archiviare per ogni numero sia la mantissa che l'esponente. Quando il numero dovrà essere ricostruito a partire da mantissa ed esponente sarà utilizzata la formula (1).

10.6 Algoritmi di ricerca

Uno dei problemi di base in informatica è la ricerca di un valore in un insieme. In particolare per gli scopi di questo corso sono interessanti la *ricerca sequenziale* e la *ricerca dicotomica*.

10.6.1 Ricerca sequenziale

La ricerca sequenziale è abbastanza banale, consiste nel leggere tutti gli elementi di un insieme, ovviamente si parla solo di insiemi aventi un numero finito di elementi, e fermarsi al primo che verifica la condizione di ricerca. A questo punto se l'insieme è ordinato la ricerca è terminata, se invece l'insieme non è ordinato bisogna comunque leggerne tutti gli elementi. Se l'insieme di partenza ha N elementi la ricerca sequenziale richiede da un minimo di una lettura ad un massimo di N. La media delle letture necessarie per trovare un elemento è N/2.

10.6.2 Ricerca dicotomica (o binaria)

La ricerca dicotomica (o binaria) si applica soltanto ad insiemi ordinati. La logica è la seguente: prima di tutto si legge l'elemento centrale dell'insieme e si confronta con il valore da cercare, se sono uguali siamo fortunati ed abbiamo finito, se il valore centrale è minore del valore cercato si scartano tutti gli elementi che vanno dall'inizio al centro dell'insieme e si continua la ricerca su quelli che vanno dal centro alla fine. Il secondo accesso avviene al centro dell'insieme residuo e così via, ad ogni iterazione vengono scartati la metà dei valori ancora sotto esame.

Per trovare l'elemento cercato la ricerca dicotomica richiede al più un numero di iterazioni pari a $Log_2(N)$ arrotondato per eccesso ed in media un numero di accessi pari a $Log_2(N)-1$.

10.7 Algoritmi e funzioni di HASH

Un algoritmo di hash è un algoritmo che riceve in input un qualunque insieme di dati (un documento, un flusso di byte, una stringa) di dimensione arbitraria e restituisce una stringa di dimensione predefinita. L'algoritmo di hash deve essere deterministico, cioè sottoponendo lo stesso input all'algoritmo più volte si deve ottenere sempre lo stesso output. Poiché la stringa di output ha dimensione prefissata e solitamente minore dell'input l'algoritmo non può essere invertibile, cioè più stringhe di input possono generare lo stesso valore di output e non è possibile, dato un valore di output, determinare la stringa di input che lo ha determinato.

Una funzione di hash è una funzione, scritta in qualunque linguaggio di programmazione, che implementa un algoritmo di hash.

Le funzioni di hash sono utilizzate, ad esempio, per generare da un file di grandi dimensioni una stringa relativamente piccola allo scopo di garantire l'integrità del file durante una spedizione. Chi invia il file calcola l'hash prima dell'invio e lo spedisce insieme al file. Chi riceve il file ricalcola l'hash e, se il file è giunto integro, deve ottenere lo stesso hash.

Un altro possibile utilizzo è il partizionamento di un insieme di dati. Se si intende partizionare in cento gruppi un insieme ampio a piacere di stringhe, queste possono essere tutte sottoposte ad una funzione di hash che restituisce cento valori differenti. Ogni gruppo sarà costituito da tutte le stringhe che generano lo stesso valore di hash.

10.8 Teoria degli insiemi

Una tabella altro non è che un insieme di righe, ognuna delle quali, a sua volta, è un insieme di singoli dati elementari. Dando per scontato il concetto di insieme ed elemento ad esso appartenente è utile richiamare alcuni concetti di base della teoria degli insiemi che saranno utilizzati più avanti. Da qui in avanti parleremo sempre di insiemi contenenti un numero finito di elementi.

10.8.1 Insiemi e sottoinsiemi

Dato un insieme S, contenente un numero arbitrario di elementi, ed un insieme T contenente solo una parte degli elementi di S e null'altro si dice che T è sottoinsieme di S. In particolare tra i sottoinsiemi di S ce ne sono due particolari: S stesso e l'insieme costituito da nessun elemento, detto *insieme vuoto*.

10.8.2 Intersezione di insiemi

Se S e T sono due insiemi, si chiama *intersezione* di S e T il nuovo insieme costituito dagli elementi di S che sono inclusi anche in T. Ovviamente è possibile che S e T non abbiano elementi in comune, in tal caso l'intersezione di S e T è l'insieme vuoto ed S e T si dicono *disgiunti*. È anche possibile che S e T contengano esattamente gli stessi elementi, in tal caso la loro intersezione coincide con entrambi gli insiemi di partenza. Il numero di elementi contenuti nell'intersezione è sicuramente minore o uguale del numero di elementi contenuti nei singoli insiemi.

10.8.3 Unione di insiemi

Se S e T sono due insiemi, si chiama *unione* di S e T il nuovo insieme costituito da tutti gli elementi di S e da tutti gli elementi di T. Se S e T hanno elementi in comune ovviamente tali elementi saranno presenti una sola volta nell'unione. Per quanto appena detto è evidente che il numero di elementi dell'unione di S e T è pari alla somma del numero degli elementi di S e del numero degli elementi di T solo quando S e T sono disgiunti. In tutti gli altri casi il numero di elementi dell'unione è inferiore alla somma del numero di elementi dei due insiemi di partenza.

10.8.4 Complemento di insiemi

Se S e T sono due insiemi si può individuare un nuovo insieme costituito da tutti gli elementi di S che non sono in T. Tale insieme prende il nome di *complemento di T rispetto ad S* oppure *insieme differenza tra S e T*. Per definizione T ed il suo complemento sono disgiunti e la loro unione è S. Il numero di elementi del complemento è pari al numero degli elementi di S meno il numero degli elementi di T.

10.8.5 Prodotto cartesiano di insiemi

Se S e T sono due insiemi si può individuare un nuovo insieme avente come elementi tutte le possibili coppie (s,t) dove s è un elemento di S e t è un elemento di T. Questo insieme prende il nome di *prodotto cartesiano di S e T*. Il numero di elementi del prodotto cartesiano è sempre pari al prodotto dei numeri degli elementi di S e T.

10.9 Logica delle proposizioni

Una *proposizione* è una affermazione non ambigua che può essere vera o falsa. Esempi di proposizione sono: “5 è un numero pari”, “4 è minore di 6”, “la parola ORACLE è composta di 12 lettere”. Perché un'affermazione sia una proposizione non è necessario che si sappia se è vera o falsa, è sufficiente che si possa con certezza affermare che essa è sicuramente o vera o falsa. Ad esempio è una proposizione “ieri nel mondo sono nati esattamente un milione di bambini” anche se nessuno oggi sa se quest'affermazione sia vera o falsa. Non è una proposizione l'affermazione “Questa frase è falsa” poiché non può essere né vera né falsa.

Le proposizioni possono combinarsi tra loro mediante alcuni operatori per formare nuove proposizioni:

10.9.1 Operatore di congiunzione

Si dice *congiunzione* di due proposizioni una nuova proposizione che è vera quando entrambe le proposizioni di partenza sono vere, falsa quando almeno una delle proposizioni di partenza è falsa. Indicheremo la congiunzione di due proposizione con la parola inglese AND che assumerà il nome di *operatore di congiunzione*. La proposizione “4 è un numero pari AND domani è domenica” è un esempio di congiunzione, poiché la prima delle proposizioni che la compongono è sempre vera, la congiunzione sarà vera solo quando è vero anche che “domani è domenica” cioè ogni sabato.

10.9.2 Operatore di disgiunzione

La disgiunzione di due proposizioni, ottenuta utilizzando l’operatore OR, è vera quando almeno una delle proposizioni è vera, falsa quando entrambe sono false. La proposizione “4 è un numero pari OR domani è domenica” è sempre vera, tutti i giorni, grazie al fatto che 4 è un numero pari.

10.9.3 Operatore di negazione

La *negazione* è proposizione che si ottiene applicando ad una proposizione data l’operatore di negazione NOT. Essa è vera quando la proposizione di partenza è falsa e viceversa. Ad esempio “NOT domani è domenica” è vera tutti i giorni tranne il sabato, è falsa solo di sabato.

10.9.4 Regole di precedenza

Quando una nuova proposizione viene ottenuta combinando più proposizioni ed utilizzando diversi operatori il significato cambia completamente in funzione dell’ordine con cui le diverse proposizioni vengono combinate. Ad esempio si consideri la proposizione

5 è pari AND 4 è minore di 2 OR 7 è dispari **(a)**

Se non specifichiamo l’ordine di combinazione delle proposizioni sono possibili due interpretazioni

5 è pari AND (4 è minore di 2 OR 7 è dispari) **(b)**

(5 è pari AND 4 è minore di 2) OR 7 è dispari **(c)**

La proposizione (b) è falsa perché 5 non è pari.

La proposizione (c) è vera perché 7 è dispari.

Per assegnare alla proposizione (a) un valore anche quando non sono specificate le parentesi si assume la precedenza dell’operatore AND rispetto all’operatore OR. Di conseguenza la proposizione (a) è equivalente alla (c) ed è vera.

10.9.5 Leggi di De Morgan

Gli operatori presentati nei paragrafi precedenti sono legati da due importanti regole che prendono il nome di "leggi di De Morgan". Siano p e q due proposizioni

NOT (p AND q) = (NOT p) OR (NOT q)

NOT (p OR q) = (NOT p) AND (NOT q)

Il senso è molto più semplice di quello che può sembrare dai simboli, la negazione di "oggi piove e domani è sabato" è "oggi non piove oppure domani non è sabato", viceversa la negazione di "oggi piove oppure domani è sabato" è "oggi non piove e domani non è sabato".

10.10 Logica delle proposizioni e teoria degli insiemi

C'è una stretta correlazione tra logica delle proposizioni e teoria degli insiemi. Entrambe le teorie rispettano le regole di un tipo particolare di algebra, l'algebra booleana. Ovviamente non è questo il contesto per trattare l'argomento dal punto di vista matematico, quindi ci limiteremo a mostrare il legame tra queste due specifiche teorie.

Sia S un insieme e siano T ed U due sottoinsiemi di S. Sia infine s un elemento di S.

1. La proposizione "s appartiene ad S" è vera sempre.
2. La proposizione "s appartiene a T" è vera per tutti e soli gli elementi di T.
3. La proposizione "s appartiene a U" è vera per tutti e soli gli elementi di U.
4. La proposizione "s appartiene a T AND s appartiene ad U", congiunzione delle 2 e 3, è vera per tutti e soli gli elementi dell'intersezione di T ed U.
5. La proposizione "s appartiene a T OR s appartiene ad U", disgiunzione delle 2 e 3, è vera per tutti e soli gli elementi dell'unione di T ed U.
6. La proposizione "NOT s appartiene ad U", negazione della proposizione 3, è vera per tutti e soli gli elementi del complemento di U rispetto ad S.
7. La proposizione "s appartiene a T AND NOT s appartiene ad U", congiunzione della 1 e della negazione della 2, è vera per tutti e soli gli elementi dell'insieme differenza T – U.
8. Le leggi di De Morgan possono essere riscritte come segue:
   - il complemento rispetto ad S dell'unione di T ed U coincide con l'intersezione dei complementi.
   - Il complemento rispetto ad S dell'intersezione di T ed U coincide con l'unione dei complementi.

Sia S un insieme. Una proposizione che descrive una condizione generica vera per alcuni elementi di S e falsa per altri si dice *predicato*. Ad esempio se S è l'insieme dei numeri interi ed x è un elemento di S, sono predicati le seguenti proposizioni: "x è pari", "x è maggiore di 10", "x è il

quadrato di un numero intero”. Ogni predicato descrive completamente un sottoinsieme di S.

Da queste definizioni e dai punti illustrati prima si intuisce che non c’è alcuna differenza tra applicare i criteri della logica delle proposizioni ai predicati oppure applicare i criteri della teoria degli insiemi ai sottoinsiemi di S che i predicati descrivono.

Indice Analitico

%

%ISOPEN · 314
%NOTFOUND · 313
%ROWCOUNT · 314
%ROWTYPE · 328
%TYPE · 299

A

ABS · 218
Accassible By · 381
ACOS · 222
ADD_MONTHS · 224
albero bilanciato · 52
Algoritmi di hash · 425
Algoritmi di ricerca · 424
Alias di colonna · 137
ALL · 276
ALTER · 90
AND · 154
ANY · 275
Applicazioni software · 421
APPLY · 265
Architettura di un computer · 420
Architettura Hardware · 420
Architettura Software · 420
Archiver · 16
ARCn · 16
Aritmetica sulle date · 223
Array Associativi · 330
Arrestare Oracle · 20
ASCII · 218
ASCII · 423
ASIN · 222
Assegnazione · 300
ATAN · 222
ATAN2 · 222
AUTHID CURRENT_USER · 384
AVG · 167
Avviare Oracle · 20

B

balanced tree · 52
BEGIN · 296
BETWEEN · 150
binary digit · 422
bit · 422
Bitmap · 53
BLOB · 70
Blocchi anonimi · 295
blocco · 13
BOOLEAN · 298
B-tree · 51
Bulk collect · 336
BUS · 420
Byte · 422

C

CASE · 247; 302
CDB · 17; 399
CDB$ROOT · 400
CEIL · 220
Central Processing Unit · 420
CHAR · 68
check constraint · 50
Check option · 56

checkpoint · 12; 15
chiave esterna · 50; 415
chiave primaria · 50; 414
chiave univoca · 50
CHR · 218
CKPT · 12
client · 22
CLOB · 69
CLOSE · 314
Cloud Computing · 398
Cloud Privato · 398
Cloud Pubblico · 398
Cluster di tabelle · 54
Cluster Index · 55
COALESCE · 239
Collection · 336
Colonne di una tabella · 49
Colonne invisibili · 94
COLUMN_VALUE · 141
Commenti · 298
COMMIT · 281
CONCAT · 208
Concatenazione · 206
Condizioni · 143
Congiunzione · 427
CONNECT BY · 179
connect string · 23
CONNECT_BY_ISCYCLE · 141
CONNECT_BY_ISLEAF · 141
Constraint · 49; 56
Container · 400
control file · 15
Conversioni implicite · 230
COS · 222
Costanti · 300
COUNT · 342
COUNT · 168
CPU · 420
Creare un utente · 30
CREATE · 71
CROSS APPLY · 265
CURRENT_DATE · 227
CURRENT_TIMESTAMP · 228
CURRVAL · 84; 141
CURSOR FOR LOOP · 315
Cursori · 313

D

data block · 13
Data Dictionary · 402
database · 11; 413
database administrator · 415
Database buffer Cache · 13
Database link · 58
database relazionale · 414
  normalizzazione · 416
  prima forma normale · 416
  progettazione · 415
  seconda forma normale · 417
  terza forma normale · 418
Database Writer · 12
datafile · 13
DATE · 70
Date ed orari · 70
Dati alfanumerici · 68
Dati binari · 70
DBA · 415
DBMS · 413
DBWn · 12
DCL · 292
DDL · 71
Deadlock · 291
DECLARE · 296
DECODE · 245
dedicated server · 23
Default Value · 112; 113
DELETE · 337
DELETE · 125
DESC · 36
DESCRIBE · 36
Dipendenze · 378
disco rigido · 420
Disgiunzione · 427
dispatcher · 23
DISTINCT · 164
Dizionario dati · 45
DML · 108
DML Trigger · 365
DROP · 101
DUAL · 139
DUP_VAL_ON_INDEX · 317

E

E. F. Codd · 413
EBCDIC · 423
Eccezioni mute · 324
Eccezioni predefinite · 317
Eccezioni utente · 323
END · 296
Enterprise Manager · 19
entità · 414
Espressioni regolari · 241

Estensioni Object-Oriented · 59
EXCEPTION · 296; 316
EXECUTE IMMEDIATE · 309
EXISTS · 340
EXISTS · 276
EXP · 221
EXTEND · 339
extent · 14
External Table · 49
EXTRACT · 228

F

FETCH · 313
FETCH · 187
file di controllo · 15
Firma · 345; 352
FIRST · 341
FLASHBACK TABLE · 107
FLOOR · 220
FOR · 305
foreign key · 50; 415
Formati
  date · 232
  numeri · 236
Funzioni · 352
Funzioni di gruppo · 165
  AVG · 167
  COUNT · 168
  MAX · 167
  MIN · 166
  STDDEV · 168
  SUM · 165
  VARIANCE · 167
Funzioni di hash · 425
Funzioni PL/SQL · 58
Funzioni predefinite · 206
  ABS · 218
  ACOS · 222
  ADD_MONTHS · 224
  ASCII · 218
  ASIN · 222
  ATAN · 222
  ATAN2 · 222
  CEIL · 220
  CHR · 218
  COALESCE · 239
  CONCAT · 208
  COS · 222
  CURRENT_DATE · 227
  CURRENT_TIMESTAMP · 228
  DECODE · 245
  di conversione · 230
  EXP · 221
  EXTRACT · 228
  FLOOR · 220
  GREATEST · 250
  INITCAP · 213
  INSTR · 211
  LAST_DAY · 225
  LEAST · 251
  LENGTH · 212
  LN · 221
  LNNVL · 240
  LOG · 221
  LOWER · 213
  LPAD · 215
  LTRIM · 214
  MOD · 222
  MONTHS_BETWEEN · 224
  NEXT_DAY · 225
  NULLIF · 241
  NVL · 237
  NVL2 · 238
  POWER · 221
  REGEXP_COUNT · 244
  REGEXP_INSTR · 242
  REGEXP_LIKE · 244
  REGEXP_REPLACE · 242
  REGEXP_SUBSTR · 243
  REMAINDER · 222
  REPLACE · 216
  ROUND · 219; 226
  RPAD · 216
  RTRIM · 214
  SIGN · 222
  SIN · 222
  SINH · 222
  SQRT · 221
  SUBSTR · 209
  sui numeri · 218
  sulle date · 223
  SYSDATE · 223
  SYSTIMESTAMP · 223
  TO_CHAR · 230
  TO_DATE · 231
  TO_NUMBER · 231
  TO_TIMESTAMP · 232
  TRANSLATE · 217
  TRIM · 215
  TRUNC · 219; 227
  UPPER · 213

G

GigaByte · 422
GOTO · 306
GRANT · 292
GREATEST · 250
GROUP BY · 165

H

Hard Disk Drive · 420
Hash · 425
Hash Cluster · 55
HAVING · 170

I

IDE · 421
Identity · 117
IF THEN ELSE · 300
IN · 150
IN (subquery) · 276
Indexed Cluster · 55
Index-Organized Tables · 54
Indice Bitmap · 53
Indice B-tree · 51
Indici · 51; 53
INITCAP · 213
INNER JOIN · 254
INSERT · 109
Installazione del database · 17
Installazione multitenant · 17
INSTR · 211
Integrated Development Environment · 421
integrità referenziale · 50; 415
INTERSECT · 272
INVALID_NUMBER · 318
IOT · 54
IS NOT NULL · 144
IS NULL · 144
istanza · 11; 12

J

Java Pool · 13
JOIN · 251

K

KiloByte · 422

L

Large Pool · 13
LAST · 341
LAST_DAY · 225
LATERAL · 266
LEAST · 251
Leggi di De Morgan · 428
LENGTH · 212
LEVEL · 141
LGWR · 12
LIKE · 152
LIMIT · 342
Linux · 420
listener · 22
Literals · 108
LN · 221
LNNVL · 240
Lock · 290
LOG · 221
LOG ERRORS · 131
log switching · 15; 16
Log Writer · 12
Logica delle proposizioni · 426
LONG · 69
LONG RAW · 71
LOOP · 303
LOWER · 213
LPAD · 215
LTRIM · 214

M

Materialized views · 56
MAX · 167
MegaByte · 422
MERGE · 126
Metodi delle collezioni · 337
MIN · 166
MINUS · 272
MOD · 222
MODEL · 196
MONTHS_BETWEEN · 224
MOUNT · 20
multiplexed · 15; 16
Multitenant Architecture · 399

N

Negazione · 427
Nested Tables · 335
NEXT · 343
NEXT_DAY · 225
NEXTVAL · 84; 141
NO_DATA_FOUND · 319
NOMOUNT · 20
non-CDB · 17; 399
Normalizzazione · 416
NOT · 156
NOT BETWEEN · 150
NOT IN · 150
NOT LIKE · 152
Null · 112; 113
NULL · 307
NULLIF · 241
NUMBER · 69
Numeri ed importi · 69
NVL · 237
NVL · 237
NVL2 · 238

O

OFFSET · 187
Oggetti del DB · 45
Oggetti PL/SQL · 58
OPEN · 313
OPEN · 20
Operatori
  aritmetici · 207
  di concatenazione · 206
  di confronto · 143
  logici · 154
Operatori Insiemistici · 268
Operatori Logici · 426
  congiunzione · 427
  disgiunzione · 427
  negazione · 427
  Regole di precedenza · 427
OR · 155
oradim · 19
ORDER BY · 159
Ordinamento · 159
OTHERS · 320
OUTER APPLY · 266
Outer join · 261
OUTER JOIN · 256

P

Package · 357
Packege PL/SQL · 58
Paginazione dei dati · 187
Partizioni · 49
Pattern Matching · 191
PDB · 17; 399
  Create · 407
  Duplicazione · 407
  Shutdown · 404
  Startup · 404
  Tnsnames · 405
  Trigger · 409
  Unplug · 406
Personalizzare SQL*Plus · 32
PGA · 13
PIVOT · 175
PL/SQL Tables · 330
PMON · 12
POWER · 221
PRAGMA
  autonomous_transaction · 394
  exception_init · 390
  inline · 395
  restrict_references · 392
  serially_reusable · 393
Predicati · 428
Prima forma normale · 416
primary key · 50; 414
PRIOR · 343
Private Cloud · 398
Privilegi · 292
Procedure · 344
Procedure PL/SQL · 58
Process Monitor · 12
processi di base · 11
Prodotto cartesiano
  di insiemi · 426
  di tabelle · 252
Progettazione di un db · 415
Program Global Area · 13
Proiezioni · 137
proposizione · 426
Pseudocolonne · 141
Public Cloud · 398
PURGE · 106

Q

Query · 136
Query gerarchiche · 179

R

Raggruppamenti · 164
RAISE · 322; 327
RAISE_APPLICATION_ERROR · 322
RAM · 420
Random Access Memory · 420
Rappresentazioni numeriche · 423
RAW · 70
RDBMS · 414
Real Application Clusters · 11
RECORD · 327
recovery · 15
Redo Log Buffer · 13
Redo Log file · 15
REGEXP_COUNT · 244
REGEXP_INSTR · 242
REGEXP_LIKE · 244
REGEXP_REPLACE · 242
REGEXP_SUBSTR · 243
relazione · 415
REMAINDER · 222
RENAME · 101
REPLACE · 216
RETURN · 352
REVOKE · 293
Ricerca Binaria · 424
Ricerca Dicotomica · 424
Ricerca sequenziale · 424
Ricerche innestate · 273
Ricerche su più tabelle · 251
ROLLBACK · 283
ROLLUP · 170
Root Container · 400
ROUND · 219; 226
ROWID · 51; 141
ROWNUM · 142; 173
RPAD · 216
RTRIM · 214
Ruoli
    Comuni · 403
    Locali · 403
RUOLI · 294

S

SAVEPOINT · 284
schema · 21
Seconda forma normale · 417
segment · 15
SELECT · 136
SELECT…INTO · 308
Selezioni · 143
Sequence · 116; 118
    di sessione · 87
Sequenze · 57
server · 22
servizio client · 22
SET · 36
SET TRANSACTION · 286
SGA · 12
Shared Pool · 13
shared server · 23
SHOW · 35
Shutdown · 20
    Abort · 20
    Immediate · 20
    Normal · 20
    Transactional · 20
SIGN · 222
SIN · 222
SINH · 222
Sinonimi · 57
Sistema binario · 421
Sistema esadecimale · 421
Sistema operativo · 420
SMON · 12
SQL · 66
SQL Developer · 42
SQL Dinamico · 309
SQL Statico · 308
SQL Types · 298
SQL*Plus · 19; 31
SQLCODE · 321
SQLERRM · 321
SQRT · 221
Startup · 20
STDDEV · 168
Stored Function · 58
Stored Procedure · 58
stringa di connessione · 23
Subprogram Inlining · 395
Subquery · 273
    correlate · 279
    Operatori di confronto · 274
    scorrelate · 278
Subquery factoring · 280
SUBSTR · 209
SUM · 165
SYS · 21
SYSDATE · 223
SYSTEM · 21
System Global Area · 12
System Monitor · 12
System Trigger · 365
SYSTIMESTAMP · 223

T

Tabella ASCII · 423
Tabelle · 48
Tabelle Index-Organized · 54
Table Trigger · 58
tablespace · 14
TCL · 281
Teorema di Bohm-Jacopini · 295
Teoria degli insiemi · 425
  Complemento · 426
  Differenza · 426
  Insieme vuoto · 425
  Intersezione · 426
  Prodotto cartesiano · 426
  Sottoinsiemi · 425
  Unione · 426
TeraByte · 422
Terza forma normale · 418
TIMESTAMP · 70
Tipi di dato SQL · 67
tnsnames.ora · 24
TO_CHAR · 230
TO_DATE · 231
TO_NUMBER · 231
TO_TIMESTAMP · 232
TOO_MANY_ROWS · 319
Transazioni · 281
TRANSLATE · 217
Trigger · 365
Trigger PL/SQL · 58
TRIM · 215; 338
TRUNC · 219; 227
TRUNCATE · 103
Truncate Cascade · 104

U

UNICODE · 423
UNION · 268
UNION ALL · 271
unique key · 50
Unix · 420
UNPIVOT · 177
UPDATE · 123
UPPER · 213
USER_SOURCE · 387
utente · 21
Utenti
  comuni · 402
  locali · 403

V

Valori fissi · 108
Valori Nulli · 237
VALUE_ERROR · 320
VARCHAR2 · 68
Variabili · 299
VARIANCE · 167
Varray · 333
Virgola fissa · 423
Virgola mobile · 424
Viste · 55
Viste materializzate · 56

W

WHERE · 143
WHILE · 306
Windows · 420
WITH · 280
WRAP · 389

X

XML DB · 59
XMLDATA · 141
XMLDB · 59

Z

ZERO_DIVIDE · 320

Indice delle figure

Figura 2-1 Multitenant Architecture: scelta tra CDB o non-CDB 18
Figura 3-1 Shutdown e startup da Sql*Plus 21
Figura 3-2 La maschera principale di Net Manager 25
Figura 3-3 Definizione del nome della connect string 26
Figura 3-4 Scelta del protocollo di rete 26
Figura 3-5 Scelta del server e della porta 27
Figura 3-6 Scelta del nome del database remoto 27
Figura 3-7 Esecuzione del test 28
Figura 3-8 Errore nel test a causa di utente inesistente 28
Figura 3-9 Indicazione delle credenziali corrette 29
Figura 3-10 Test con esito positivo 29
Figura 3-11 Maschera di salvataggio delle modifiche 30
Figura 3-12 Connessione con Sql*Plus da client remoto. 31
Figura 4-1 Menù contestuale. 33
Figura 4-2 Proprietà di SQL*Plus. 34
Figura 4-3 Proprietà del Layout. 34
Figura 4-4 Modifica del collegamento. 35
Figura 4-5 Modifica del collegamento. 35
Figura 4-6 Tutti i parametri di SQL*Plus. 36
Figura 4-7 Impostazione della linesize. 37
Figura 4-8 Il comando DESC. 38
Figura 4-9 Ritorno a capo durante la scrittura di un comando SQL. 38
Figura 4-10 Esecuzione di un comando SQL. 39
Figura 4-11 Apertura di *afiedit.buf* 40
Figura 4-12 Esecuzione del comando SQL modificato. 40
Figura 4-13 I comandi “i”, “del” e “c”. 41
Figura 4-14 I comandi “start” e “@”. 42
Figura 4-15 Il comando “set sqlprompt”. 42
Figura 4-16 Pagina principale di SQL Developer. 43
Figura 4-17 Configurazione delle connessioni. 44
Figura 4-18 Esecuzione di un’istruzione SQL. 45
Figura 5-1 Lettura del dizionario da SQL*Plus. 48
Figura 5-2 Lettura del dizionario da SQL Developer. 48

Figura 5-3 Caratteristiche principali di una tabella. 50
Figura 5-4 Struttura di un indice B-tree...53
Figura 8-1 Esecuzione del programma PL/SQL in SQL Developer. 296
Figura 8-2 Visualizzazione dell’output in SQL Developer............................297
Figura 8-3 Uno script PL/SQL in chiaro ed offuscato. 389
Figura 9-1 Datafile e Tempfile del PDB 412