INTRODUZIONE AI DATA BASE RELAZIONALI
SQL – PL/SQL

Revisione 14 febbraio 2013

Lo scopo di questo manuale è quello di fornire un utile supporto alle lezioni.

Il manuale non intende sostituire né la documentazione del prodotto a cui si fa riferimento, né la lezione del docente.

E' a discrezione del docente l'ordine di trattazione degli argomenti, gli eventuali approfondimenti e rimandi.

Per eventuali informazioni telefonare alla segreteria della Digital Institute ai seguenti numeri:

Tel. 06/63651 oppure Fax 06/99369785.

Digital Institute è la denominazione del servizio formazione di Fata Informatica S.r.l.

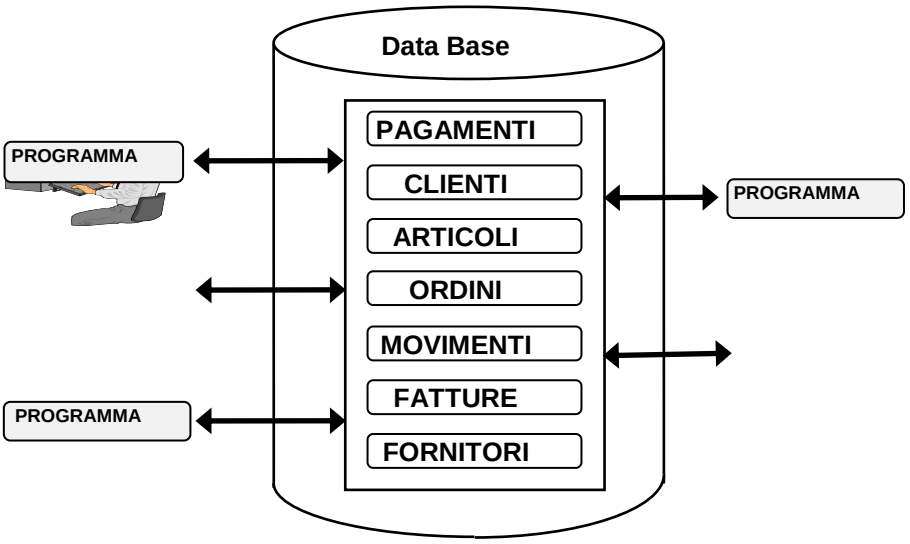
INDICE

1 Introduzione ai Data Base

Uno dei maggiori campi di applicazione degli elaboratori elettronici è costituita dai DATA BASE, specie in ambito aziendale e, comunque, in tutti quei casi in cui si debba gestire una grande quantità di dati disomogenei.

Un sistema informativo integrato presuppone il concetto di data base, cioè di una collezione organica di dati che:

- Fornisce una naturale rappresentazione delle risorse aziendali
- Soddisfa le richieste dati di più applicazioni.
- Presenta i dati in modo condiviso ad una varietà di utenti.
- Permette a un utente di interrogare l'intero data base utilizzando un linguaggio di interrogazione semplice, svincolandosi da programmi prestabiliti, ovvero dando maggiore flessibilità all'utente finale.



Definiamo DATA BASE MANAGEMENT SYSTEM (DBMS) un qualsiasi sistema in grado di memorizzare stabilmente un certo insieme di dati che risiedono su memorie di massa (es. dischi) di adeguata capienza e velocità di accesso, in modo da consentire la consultazione o la manipolazione da parte delle applicazioni e degli utenti che ne hanno necessità.

Gli ultimi anni hanno registrato una forte tendenza verso lo sviluppo di sistemi informativi aziendali di vaste dimensioni che utilizzano la tecnologia delle basi di dati. L'introduzione dei sistemi di gestione di base di dati (DBMS), cioè di sistemi in grado di gestire in modo integrato i dati necessari a tutte le applicazioni, ha di fatto modificato le precedenti metodologie di progettazione delle applicazioni. Al progetto di archivi tra loro indipendenti e legati alle singole applicazioni si è sostituito il progetto, assai più complesso, di

una struttura di dati vista come unica ed integrata risorsa dell'intera impresa, utilizzata da più utenti ed ambienti applicativi.

I principali vantaggi di un DBMS rispetto a una gestione delle informazioni mediante files individuali sono tutti riconducibili al fatto che si istituisce nel sistema informativo un CONTROLLO CENTRALIZZATO dei dati che comporta:

RIDUZIONE DELLE RIDONDANZE: nell'ambiente non Data Base ogni applicazione gestisce autonomamente un suo insieme di archivi; ciò conduce facilmente ad avere lo stesso dato ripetuto varie volte nel sistema informativo. Con l'impiego di un Data Base la proliferazione dei dati viene evitata, sia perchè esiste un disegno delle strutture dati più controllato, sia perché nuove applicazioni possono sfruttare agevolmente i dati già previsti.

MINORI RISCHI DI INCONSISTENZA NEI DATI: eliminando le ridondanze e ponendo gli accessi sotto il controllo di uno specifico software, la integrità logica dei dati risulta meglio assicurata e viene meno il rischio che, in certi momenti del lavoro, esistano copie disallineate dello stesso dato.

PROTEZIONE DA ACCESSI NON AUTORIZZATI: il DBMS, avendo il controllo sia sui dati che sugli utenti del Data Base, è in grado di evitare accessi non autorizzati, cosicché ciascun utente possa utilizzare i dati aziendali in maniera differenziata.

INTEGRITÀ' DEI DATI MEGLIO GARANTITA: i dati gestiti sotto il controllo di un DBMS centralizzato possono essere meglio protetti a fronte di anomalie nel funzionamento del sistema o dei programmi, avvalendosi di meccanismi automatici di controllo degli accessi concorrenti, di ripristino dei dati alternati, etc.

LEGAMI LOGICI INGLOBATI NELLA STRUTTURA DATI: i collegamenti tra i vari tipi di dati (es. Clienti, Ordini, Prodotti) possono essere descritti, e incorporati nel Data Base ,insieme ai dati. In tal modo la codifica delle applicazioni sarà più semplice.

FACILITA' DI CRESCITA DEL SISTEMA: nuovi tipi di dati e nuove componenti applicative possono essere aggiunte più agevolmente grazie alla natura integrata del data base e alla minimizzazione delle ridondanze.

1.1 L'APPROCCIO AI DATI

Il Data Base può essere visto come un unico archivio logico in cui convergono tutti i dati necessari ad una o più applicazioni. Questo non significa, che i vari tipi di dati relativi ad es. Fatture o Fornitori, non sono più distinguibili tra di loro, ma che sono gestiti su spazi fisici comuni, con modalità di allocazione e reperimento controllate globalmente, invece di gestire un archivio a se stante per ogni tipo di dato. Questo permetterà, tra l'altro di gestire, più

agevolmente, e talvolta anche di inglobare nel data base, i collegamenti logici che sussistono fra i diversi tipi di dati.

Un ambiente data base correttamente impostato dovrebbe consentire il raggiungimento dei seguenti obiettivi:

- RELAZIONABILITÀ
- NON-RIDONDANZA
- INTEGRITÀ'
- SICUREZZA
- CONDIVISIBILITÀ
- PRESTAZIONI, AMMINISTRAZIONE E CONTROLLO

1.1.1 Relazionabilità e Non-Ridondanza

È probabile che in una realtà di dati comuni sia possibile avere più file che contengono gli stessi dati, per esempio più schede relative ad uno stesso venditore. Tali dati in eccesso vengono definiti: Ridondanti. Si può quindi affermare che eliminare tale effetto di ridondanza significa effettuare l'ottimizzazione della struttura dati del caso. I vantaggi di tale ottimizzazione sono:

- Una gestione efficace della memoria di massa, visto che si eliminano dei file che non verrebbero utilizzati.
- Una più rapida consultazione del data base, poiché una migliore gestione della memoria ha questo vantaggio.
- Meno possibilità di errori in fase di aggiornamento (up-date), visto che è più semplice aggiornare un singolo file che più di uno contenente le stesse informazioni.

Le uniche ridondanze ammesse sono quelle strettamente necessarie per assicurare la relazionabilità dei dati tra tabelle diverse.

La relazionabilità impone di fatto che i dati in tabelle diverse possono essere messi in relazione attraverso un campo o una chiave di campi comuni alle diverse tabelle. Questo tipo di ridondanza è l'unica ammessa.

ESEMPIO:

Tabella 1

COD	NOME	COGNOME	STATO CIVILE
001	GIACOM	BIANCHI	CELIBE

Tabella 2

COD	NOME	COGNOME	DITTA
001	GIACOMO	BIANCHI	FIAT

In questo esempio i campi NOME e COGNOME sono ridondanti, mentre il campo COD è utile al fine di ottenere un collegamento logico tra le due tabelle.

1.1.2 Integrità

Per integrità si intende tutto ciò che riguarda la salvaguardia fisica delle informazioni.

Devono essere garantiti:

- La consistenza dei valori
- il backup
- il recovery
- i controlli logici (es. Non possono esistere ordini se non esiste il cliente).

Quotidianamente è opportuno fare un backup dei dati.

Il backup consiste nel ricopiare le informazioni su supporti esterni (dischi e nastri).

In caso di danneggiamento le informazioni potrebbero essere ripristinate con una procedura di recovery dai dischi precedentemente conservati.

1.1.3 Sicurezza

Salvaguardare la sicurezza dei dati, in un sistema nel quale una moltitudine di utenti potrebbe accedervi liberamente, significa prevedere funzioni di definizione, gestione e controllo dei:

- diritti di accesso
- livelli di accesso
- ammissibilità delle operazioni sui dati.

Tale divisione è resa necessaria dalla natura dei dati che si immagazzinano, infatti dati inerenti alla contabilità saranno di visione soltanto al personale inerente e non tutti.

1.1.4 Condivisibilità

Le procedure di gestione del data base devono poter accedere contemporaneamente alle diverse tabelle.

Ogni procedura "gira" autonomamente, ignorando la presenza di altri programmi; ogni procedura accede alle informazioni del data base in modo dedicato, cioè alla sola porzione di dati che compete alla procedura e gestendone in modo "personale" la visione.

I dati presenti nel Data Base possono essere utilizzati contemporaneamente da più utenti, senza che questo comprometta la sicurezza ed integrità dei dati presenti.

1.1.5 Prestazioni, Amministrazione e Controllo

A contorno delle funzioni disponibili, devono essere messi a disposizione:

- Supporti strumentali e metodologici al disegno del data base
- Dizionario dei dati
- Capacità di gestire dati sia di produzione che di prova
- Strumenti di tuning
- Funzioni di riorganizzazione.

1.2 Analisi dei Dati

L'analisi dei dati si occupa delle relazioni tra le risorse e delle loro caratteristiche, di come queste siano rappresentabili e di come queste rappresentazioni si possono trattare in sostituzione delle risorse stesse.

Scopo dell'analisi è la produzione di un modello descrittivo che espliciti in modo univoco le risorse e i legami tra di esse sotto forma di strutture dati.

1.2.1 Interpretazione e Rappresentazione Della Realtà

Il dato può essere definito come una descrizione finalizzata di una porzione della realtà.

Il dato nasce dunque da un processo conoscitivo svolto da un osservatore che si pone certi scopi e formula ipotesi sulla realtà.

1.3 Gli Approcci Metodologici

Le metodologie di progettazione di basi di dati consentono di inquadrare l'intero processo di progettazione in una sequenza di operazioni.

I componenti di base essenziali per una metodologia di progetto di base di dati sono quattro:

1. Definire un processo di progettazione strutturato, che consiste in un insieme di passi e consenta di scegliere per ciascun passo fra varie alternative;
2. Disporre di tecniche che consentano di classificare ed enumerare le varie alternative, e i criteri di valutazione per la scelta;
3. Disporre di informazioni adeguate, sia all'inizio del progetto, sia per ciascuna sua fase;
4. Disporre di meccanismi descrittivi per rappresentare le informazioni di ingresso ed i risultati di ciascun passo di progettazione.

1.4 La Progettazione

Gli obiettivi della progettazione di una base di dati in un ambiente applicativo sono la realizzazione di una struttura di dati (o schema) che incorpori tutte le informazioni atte a descrivere l'ambiente applicativo, e di un insieme di transazioni (programmi) che agiscono su tale struttura svolgendo i compiti richiesti dalle varie applicazioni.

Il Data Base vuole essere la risposta globale alle esigenze di organizzazione e gestione dei dati entro il sistema informativo, chiamato anche a tener conto delle correlazioni logiche che sussistono fra le varie entità presenti, correlazione che i programmi applicativi dovranno gestire in accordo con le regole aziendali.

Esso è quindi destinato, in linea di principio, a contenere tutti i dati che riguardano le cose, persone, eventi, ecc. Che costituiscono l'oggetto delle attività aziendali toccate dal progetto di automazione.

Proprio perché il Data Base deve rappresentare la sintesi di molteplice e, sovente, complesse esigenze organizzativi ed applicative, la sua realizzazione richiede una preventiva attività di analisi e di progetto che va' sotto il nome di "DATA BASE DESIGN".

Le proposte più recenti al riguardo, portano a considerare il Data Base design come un processo che si articola in tre momenti fondamentali, ciascuno dei quali corrisponde ad un differente livello di astrazione e di visibilità dei dati.

Sviluppare il Data Base design mediante questi tre passi, permette non solo di separare problemi diversi, ma anche di garantire ampia libertà di scelta e di

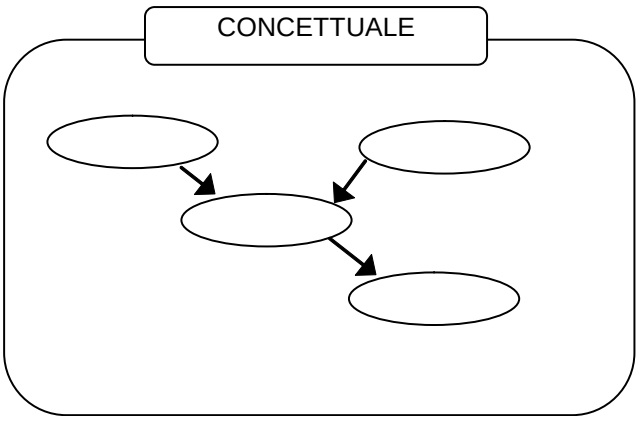
modifica delle soluzioni adottate ai livelli più bassi, senza intaccare la validità di quanto costruito ai livelli precedenti.

1.4.1 I Prerequisiti Informativi

Rappresentano le descrizioni dell'organizzazione per la quale devono essere raccolti i dati, gli obiettivi della base di dati, l'indicazione di quali dati devono essere raccolti e memorizzati.

Essi contribuiranno in prima approssimazione alla creazione della struttura dei dati o schema.

Disegno:



Le fasi della progettazione di una base di dati si possono così riassumere:

- Analisi dei requisiti
- Progettazione concettuale
- Progettazione logica
- Progettazione fisica e implementazione.

1. Analisi dei requisiti,
   in cui si rilevano i requisiti degli utenti del sistema informativo;

2. Progettazione concettuale,
   in cui a partire dai requisiti si crea uno schema concettuale. Tale schema descrive le interrelazioni fra i dati di interesse, ed è indipendente dal DBMS (sistema di gestione di basi di dati) adottato per realizzare il sistema informativo;
3. Progettazione logica,
   in cui si produce lo schema dei dati, definibile nel DDL (linguaggio per la definizione dei dati), dello specifico DBMS utilizzato per l'applicazione. Questa fase viene svolta tenendo conto di problematiche quali l'efficienza del sistema informativo e l'integrità, l'affidabilità e la sicurezza dei dati;
4. Progettazione fisica e implementazione,
   in cui vengono definite le strutture di memorizzazione dei dati; viene successivamente svolta la implementazione effettiva del sistema, utilizzando il DDL per la definizione degli schemi, il DML (linguaggio per la descrizione dei dati) per la scrittura dei programmi.

1.4.2 Il Sistema Dizionario

Il disegno di un modello di dati valido non è sufficiente di per sè a garantire nel tempo la buona funzionalità del DBMS.

Non ha senso aver ridotto tutti i dati elementari aziendali ad una base standardizzata semplice, unica, protetta e condivisibile se la descrizione dei dati stessi e di tutti i processi ad essi correlati non è altrettanto standardizzata, semplice, unica, accessibile a tutti e così via.

È indispensabile in questo caso il dizionario dei dati, strumento di supporto utilissimo per tutte le fasi del disegno di un data base e per le attività successive di sviluppo e manutenzione delle applicazioni.

Possiamo definire un dizionario dei dati come il depositario di tutte le informazioni relative ai dati intesi come entità e fonte univoca di informazioni per l'intera organizzazione aziendale.

Un sistema dizionario dei dati (DDS = data dictionary system) serve pertanto al D.B.A. come al D.A., ai programmatori e analisti come agli utenti.

È uno strumento validissimo per tutte le funzioni aziendali a condizione però che venga tenuto aggiornato. Questo compito specifico è assegnato al D.B.A. (data base administrator) ed al D.A. (data administrator).

Tutti gli altri utenti, salvo casi particolari, accedono al D.D.S. solo per consultazione.

Per la sua chiarezza informativa e per l'elevata produttività nelle fasi di sviluppo e manutenzione di software applicativo, il sistema dizionario, in una

azienda efficiente, si giustifica ancor prima di un DBMS e con esso efficacemente si integra.

Le fasi attraverso cui è necessario passare per progettare un data base si possono sintetizzare in:

- raccolta dei dati
- analisi dei dati
- normalizzazione
- disegno
- scelta del modello
- schema interno ovvero disegno del modello concettuale / schema interno.

Il "modello concettuale" può essere inteso anche come "modello dei dati" e il "disegno del modello concettuale" diventa quindi "disegno logico di data base".

Fondamentale per la ottimale funzionalità di un DBMS è la raccolta e identificazione dei dati.

È necessario quindi che i dati siano esattamente identificati fissandone il nome e il significato e includendoli in un opportuno catalogo.

Questo compito, delicato e importante per il buon andamento di un'azienda, è affidato ad una persona. Il data administrator (D.A.) che collabora con gli utenti e il D.B.A. e possiede una perfetta conoscenza dell'azienda e delle sue strategie di sviluppo.

Il catalogo dei dati potrà poi essere caricato in un dizionario dei dati insieme con altre informazioni che riguardano il loro contesto nel DBMS e che sono fornite dal DBA.

Il dizionario dei dati da un contributo determinante nell'assicurare l'univocità dei dati e nell'economia dei tempi necessari per lo sviluppo di nuove applicazioni.

1.5 La Progettazione Concettuale

La fase della progettazione concettuale delle basi di dati ha come obiettivo principale la descrizione degli eventi che si verificano che siano di interesse per l'ambiente applicativo in cui la base di dati dovrà essere realizzata.

I dati che vengono percepiti all'esterno del sistema informativo devono essere formalizzate in accordo alle regole vigenti nel contesto in esame. Questa fase deve perciò cercare di rappresentare i dati e le loro correlazioni in maniera del tutto generale, astraendo da ogni particolare tecnica di elaborazione propria

di questo o quel prodotto DBMS. Ciò che si vuole conseguire è quindi un modello dei dati il più possibile autonomo dalle successive implementazione, stabile e congruente con le necessità dell'utenza. Il designo concettuale non può tuttavia nascere dal nulla. Esso partirà dai risultati della rilevazione dei REQUISITI UTENTE. Tale analisi definirà gli obiettivi del progetto e fornirà le informazioni circa la realtà da rappresentare nel Data Base. Compito iniziale del Data Base design sarà quindi quello di estrarre da tali specifiche preliminari le strutture informative rilevanti, dalle quali sviluppare poi un progetto di dettaglio con l'aiuto di una metodologia. Tra le metodologie di design concettuale, la più nota e diffusa è quella che va sotto il nome di "ENTETY RELATIONSHIP MODELING" proposta da Chen nel 1976 ed ulteriormente sviluppata da contributi successivi. Secondo l'approccio Entity-Relationship (E-R), il design concettuale del Data Base si sviluppa in modo top-down (dal generale al particolare) secondo tre momenti fondamentali:

- ANALISI DELLE ENTITÀ
- ANALISI DELLE CORRELAZIONI
- ANALISI DEGLI ATTRIBUTI

I risultati del design concettuale vengono usualmente sintetizzati in forma grafica, mediante diagrammi che utilizzano una opportuna simbologia (diagrammi "E-R"). Tali diagrammi, se il progetto non è banale, andranno corredati di prospetti esplicativi atti a documentare dettagliatamente le scelte operate dall'analista.

Come strumenti per la descrizione della realtà vengono utilizzati i modelli dei dati.

Le caratteristiche che si richiedono ai modelli dei dati sono:

- essere formali: cioè non accettare definizioni ambigue
- essere completi: devono poter descrivere ogni aspetto della realtà
- essere semplici: consentire cioè di descrivere la realtà ed essere di semplice interpretazione
- essere infologici: ossia mirare alla descrizione dei dati esclusivamente dal punto di vista del loro contenuto informativo.

1.5.1 Il Modello Entità - Relazioni (Metodologia Chen)

Per arrivare a descrivere in termini di dati la realtà occorre munirsi di strumenti concettuali di interpretazione.

L'analisi E-R consiste nella individuazione dei tipi di ENTITÀ che classificano, oggetti, persone, eventi di interesse per il settore aziendale per cui si progetta il data base. Si tratta quindi di INSIEMI di cose del medesimo genere, per le quali si vogliono mantenere informazioni da elaborare con criteri omogenei.

Il primo di questi concetti è entità.

- Entità

Dato un insieme di oggetti del mondo reale, questi oggetti

Vengono raccolti in classi tali che:

1. Oggetti appartenenti alla medesima classe condividano il maggior numero possibile di caratteristiche
2. Oggetti appartenenti a classi diverse risultino il più possibile differenti tra loro. Obiettivo della classificazione è quindi un ordine ottenuto attraverso il riconoscimento delle eguaglianze e delle differenze.

Possiamo definire come entità:

Persona, avvenimento, luogo, tempo e qualsiasi altra cosa che sia di interesse conoscitivo.

Il tipo entità consiste in una certa classe di oggetti che soddisfa una definizione data.

L'occorrenza di entità è un certo oggetto che soddisfa la definizione di entità data.

Quindi:

- Una entità non è un insieme (lista) di occorrenze, non sono cioè le occorrenze che fanno entità, ma è entità a stabilire le sue occorrenze discriminando tra gli oggetti del mondo reale.
- Aggiungere, cancellare, cambiare un'occorrenza non modifica entità.

ESEMPI DI ENTITA':

Tipo	Definizione	Occorrenz
Città'	Centro di vita sociale in grado di	Milano
	Adempiere a molteplici funzioni	Roma

	Sociali	Avellino
		Perugia
Squadra	Insieme di giocatori di calcio che	Roma
	Partecipano a competizioni per	Juventus
	L'affermazione dei propri colori	Milan

- Attributo

Il momento successivo del designo concettuale, consiste nell'individuare gli ATTRIBUTI di ciascuna entità, vale a dire i dettagli che qualificano o descrivono le proprietà della entità in questione, in accordo alle esigenze informative degli utenti del sistema.

Un oggetto possiede delle proprietà che ne costituiscono i tratti caratteristici; quindi ogni entità è descritta ed identificata da una serie di attributi.

Gli attributi sono elementi identificativi e descrittivi associati a ciascuna occorrenza di entità.

Nella frase: "x ha un'anzianità di servizio di cinque anni"; individuiamo in "anzianità di servizio" un attributo entità "x" e in "cinque anni" uno dei suoi possibili valori.

Il tipo dell'attributo, è la definizione di una funzione sulle occorrenze entità, che trae i suoi valori da un insieme D detto dominio dell'attributo.

Il dominio degli attributi è la descrizione formalizzata dei valori che possono venire assunti da un tipo di attributo, e comprende:

- nome del dominio
- insieme d'appartenenza
- dimensione e formato
- range o lista dei valori
- indicatore di ordinamento.

Si passerà poi ad analizzare ciascun attributo per valutare:

Il suo DOMINIO di definizione, ossia l'insieme dei valori che esso può assumere

La sua FUNZIONE rispetto all'entità che concorre a qualificare

La determinazione del DOMINIO di valori di un attributo serve a facilitare e, potenzialmente, ad automatizzare i controlli di validità sui dati. Ad es. Se il dominio "MATRICOLE" contiene numeri di 4 cifre più un carattere, il sistema dovrà impedire l'inserimento di una matricola di 5 cifre numeriche.

Inoltre, se più attributi condividono lo stesso dominio, essi possono costituire, a secondo dei casi, ridondanze da eliminare o elementi di collegamento fra entità differenti.

Il momento successivo consiste nel verificare la FUNZIONE di ogni singolo attributo nei confronti della entità cui appartiene.

A tale riguardo, un attributo può avere una funzione:

- IDENTIFICATIVA: I suoi valori possono servire per distinguere le diverse occorrenze dell'entità stessa (es. MATRICOLA, NOME)
- PREFERENZIALE: Il suo contenuto permette di identificare una occorrenza di una entità con la quale esiste una correlazione (es. CODICE-DIPARTIMENTO, MANAGER)
- DESCRITTIVA: Se non è identificativo o referenziale (LIVELLO).

ESEMPI DI ATTRIBUTI:

Entità'	Attributo	Valore
Cliente	Nome	Rossi
	Codice fiscale	Rssalfc17f205j
	Rag. Sociale	Rossi & c.
	Indirizzo	Via Roma 10
	Città'	Milano

- Il Dato Elementare

Per eliminare l'ambiguità di interpretazione propria degli attributi e delle entità si può ricorrere a una formalizzazione del concetto attributo.

Perché un attributo sia tale occorre che il suo valore sia esprimibile come valore di un dato elementare si definisce dato elementare un dato che non sia ulteriormente scomponibile in maniera significativa.

Una ulteriore conseguenza della analisi degli attributi consiste nell'evidenziare, nell'ambito di una entità, uno o più sottoinsiemi di occorrenze per le quali soltanto risultano significativi determinati attributi. Tali sottoinsiemi vengono definiti come "SOTTO-TIPI" dell'entità originaria, che diviene allora un

"SUPER-TIPO" nei confronti dei suoi sotto-tipi. Una occorrenza di un sotto-tipo resta sempre una occorrenza del suo super-tipo, pur potendo avere attributi specifici e correlazioni autonome con le altre entità del modello. L'entità super-tipo manterrà solo i dati comuni a tutte le sue occorrenze. Essa dovrà però prevedere un attributo che permetta di identificare se una sua occorrenza appartiene anche ad un sotto-tipo o meno.

- Relazioni e Associazioni

Una relazione è una corrispondenza tra gli elementi di due insiemi.

Si distinguono due tipi di relazione:

- relazione attributo
- associazione

Le relazioni attributo definiscono una relazione tra gli attributi significativi della stessa entità.

La relazione mantiene il proprio significato fin tanto che entità esiste.

Un'associazione definisce una relazione tra diverse entità, attraverso la definizione di una proposizione che stabilisce la relazione.

Ad esempio: se Caio possiede un'auto, "Caio" e "auto" sono entità associate dalla proporzione "possiede".

1.5.1.1 Identificatori Di Entita': Le Chiavi

La chiave di entità consiste di un insieme di attributi propri entità stessa, che consentono l'identificazione univoca delle sue occorrenze.

- Proprietà Delle Chiavi

Definiamo come chiave "k" di entità "e" un insieme di attributi di "e" aventi le seguenti proprietà:

1. Identificazione:
   Ogni occorrenza di "e" è identificata univocamente dall'insieme dei valori di "k".
2. Non ridondanza:
   Non vi sono attributi in "k" che possano essere scartati senza che venga meno la proprietà 1.

Si assume che il valore degli attributi chiave sia sempre definito per ogni occorrenza di entità.

Il concetto di chiave determina la distinzione degli attributi in due categorie:

1. Attributi chiave: identificano univocamente entità a cui appartengono.
2. Attributi semplici: non identificano univocamente le entità a cui appartengono.

- **Chiavi Alternative**

Ogni occorrenza di entità dispone di un attributo o di un insieme di attributi chiave. Se le chiavi sono più di una, si distinguono una chiave primaria e una o più chiavi alternative.

Generalmente, le procedure di gestione devono gestire l'esatta sequenza delle chiavi.

- **Chiavi Semplici e Chiavi Composte**

Una chiave costituita da un solo attributo si definisce chiave semplice; ad esempio le singole occorrenze entità "cliente" si identificano mediante la chiave primaria semplice costituita dall'attributo "codice cliente".

Una chiave costituita da più di un attributo è detta chiave composta; ad esempio per identificare i prodotti di una certa linea commerciale, è necessaria una chiave composta da "codice prodotto" e "codice linea commerciale".

- **Ruolo Delle Chiavi**

Ricapitolando, i diversi ruoli della chiave primaria sono di identificazione nei confronti entità a cui appartiene, di determinazione dei rimanenti attributi semplici entità (relazione attributo) e di associazione con altre entità.

1.5.2 Tipologia Delle Relazioni

Si è dunque visto come una precisa definizione del concetto di chiave sia estremamente utile per esprimere e rappresentare le associazioni tra entità e le relazioni tra attributi.

L'analisi delle relazioni ed associazioni non può però prescindere dal loro aspetto quantitativo.

Anche questo tipo di informazione (quante entità o attributi di un certo tipo sono associate a un'entità o attributo di un altro tipo) deve infatti essere rappresentata.

A. Relazioni semplici (di tipo 1:1):

Ad ogni occorrenza dell'elemento "da" corrisponde ad una ed una sola occorrenza dell'elemento "a".

L'occorrenza "da" svolge una funzione identificativa dell'occorrenza "a"; ad un certo valore "x" dell'insieme "y" deve corrispondere quindi un determinato valore "w" dell'insieme "z".

A più occorrenze "da" può corrispondere una sola occorrenza "a".

Ne segue una nuova definizione di chiave, importante perché è formulata indipendentemente dal concetto di entità:

Una chiave è un elemento che ne identifica un secondo mediante una relazione (o associazione semplice).

Un esempio di relazione 1:1 può essere il seguente:

Da		A
Cognome		**Ruolo**
Lennon		Chitarrista
Mc cartney		Bassista
Harrison		Chitarrista
Starkey		Batterista

A)Relazioni complesse (di tipo 1:n):

Ad ogni occorrenza dell'elemento "da" può corrispondere un qualsiasi numero n di occorrenze dell'elemento "a".

Quindi, l'elemento "da" non identifica univocamente l'elemento "a".

Esempio di relazione 1:n

Da	⇨	**A**
Famiglia	⇨	**Nome componente**
Rossi	⇨⇨ ⇩	**Alberto**
	⇨⇨ ⇩	**Lorena**
	⇨⇨	**Vittorio**

B)Relazioni condizionali (caso particolare delle relazioni o associazioni di tipo 1:n):

Ad ogni occorrenza dell'elemento "da" corrisponde o una o nessuna occorrenza dell'elemento "a".

Esempio di relazione condizionale:

Da	⇨	A

Modello	⇨	Optionals
Delta		
Delta Lx	⇨	Si
Dedra		
Dedra Td	⇨	Si
Thema	⇨	Si

Nell'analisi dei dati incontreremo e dovremo rappresentare tutti i tipi di relazioni o di associazioni considerati, tuttavia le relazioni ed associazioni condizionali possono essere ricondotte a quelle di tipo 1:1 o 1:n.

Restano quindi solo i seguenti tipi:

1:1

1:n

n:1

n: m

Per quanto riguarda le relazioni di tipo n:m, si può rendere chiaro il concetto che sta alla loro base tenendo conto che si scompongono in due relazioni di tipo 1:n.

Si può riscontrare talvolta l'esistenza di una correlazione che coinvolge una stessa entità (correlazioni 'MONADICA'). In tal caso interessa evidenziare i due diversi RUOLI che l'entità in questione viene a svolgere. Ad es. una correlazione che esprime una gerarchia aziendale può rappresentarsi sull'unica entità 'PERSONALE', nell'ambito della quale alcuni soggetti svolgono il ruolo di 'MANAGER' e altri il ruolo di 'IMPIEGATO'.

1.5.3 L'indipendenza Dell'organizzazione Logica Da Quella Fisica

Il concetto di indipendenza logica consiste nel ritrovare delle informazioni all'interno della base di dati indipendentemente della posizione in cui sono memorizzate.

Tra gli obiettivi che si sono posti quanti hanno sviluppato i DBMS, vi è quello di assicurare alle applicazioni un elevata "Data Independence", ossia fare in modo che la codifica dei programmi applicativi risenta il meno possibile gli effetti di modifiche apportate alle strutture che ospitano i dati.

L'utilizzo dei files tradizionali preclude in modo pressoché completo la data independence, poiché la codifica di un programma risulta vincolata alle caratteristiche fisiche e alle modalità di elaborazione dei records imposte dalla specifica organizzazione.

Supponiamo ad esempio di dover elaborare un archivio "DIPENDENTI" allocato su un file a indici, avente una struttura dei records come quella illustrata nella pagina accanto.

Nel programma sorgente il programmatore deve codificare il fatto che si tratta di un file indexed, specificare un certo modo di accesso (ad es. Dynamic), conoscere posizione e lunghezza del campo che funge da chiave di accesso, specificare la successione esatta dei campi del record, la loro dimensione e il formato interno.

Possiamo evidenziare dipendenza dalla struttura dei dati quando le applicazioni devono essere scritte tenendo espressamente conto di caratteristiche quali:

La dipendenza delle applicazioni dalla struttura dei dati, è una delle ragioni per cui, nel Data Processing tradizionale, la manutenzione del software tende a diventare predominante rispetto allo sviluppo delle nuove applicazioni.

La misura in cui la Data independence può essere conseguita è fortemente condizionata dalla tipologia di organizzazione dei dati utilizzata dal DBMS. I data base meno recenti utilizzano organizzazioni dei dati e tecniche di accesso che riflettono collegamenti a livello fisico, limitando notevolmente il livello di. indipendenza dei dati.

1.5.3.1 Le Viste Locali O D'utente

Il problema più rilevante nella progettazione di una base di dati è quello di dare una visione all'utente finale delle Informazioni che a lui possono interessare, ossia una visione parziale dei dati, chiameremo ciò " vista utente ".

1.5.3.2 Integrazione Delle Viste Locali

Consiste nell'integrare le viste utenti in un unico schema pertanto i problemi complessi saranno decomposti in sotto problemi indipendenti con modelli parziali (view).

1.6 La Progettazione Logica

La progettazione logica ha per oggetto la trasformazione dello schema concettuale (Entità Correlazioni, Attributi) nelle strutture e modalità di organizzazione dei dati proprie di un determinato prodotto di DBMS.

Questa fase del DB design produce un modello logico del Data Base che costituisce il punto di riferimento per le applicazioni che manipolano i dati contenuti nel Data Base.

Il Modello Logico è normalmente costituito da uno "SCHEMA", che descrive globalmente la struttura logica del data base, e da un certo numero di "SOTTOSCHEMI" O "VISTE" parziali, che realizzano una ulteriore interfaccia tra il data base e le applicazioni, allo scopo di semplificare la visione dei dati, filtrare gli accessi e migliorare la " Data Independence".

L'evoluzione della tecnologia dei Data Base ha evidenziato diverse tipologie di organizzazione logica dei dati. Quelle più frequentemente citate e utilizzati sono le organizzazioni.

1.6.1 Il Disegno

Oltre ad avere individuato gli elementi essenziali da comporre nello schema concettuale è utile per la scelta del modello conoscere le associazioni tra i dati da un punto di vista quantitativo.

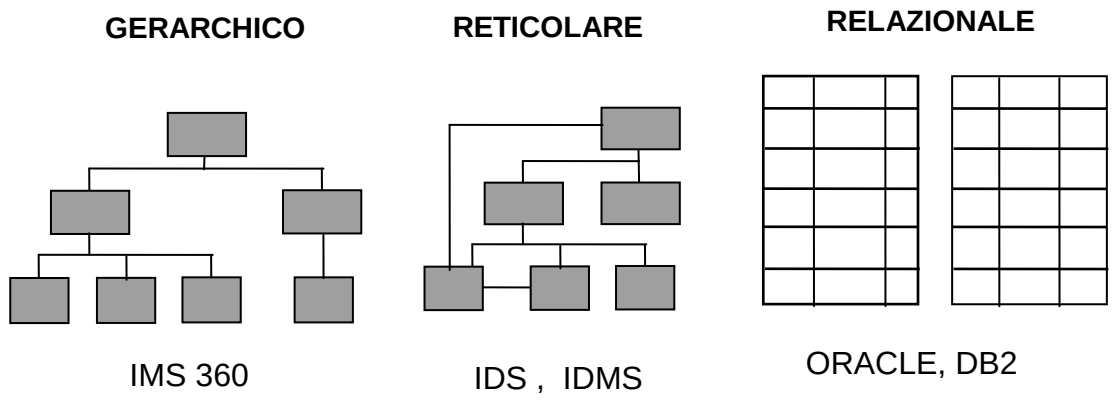
Il disegno è attività di analisi sui legami tra i dati che porta a stabilire la corrispondenza quantitativa (ad esempio "uno a molti") in un senso e nell'altro tra due dati qualsiasi.

Dal disegno deriva un grafo che evidenzia i legami esistenti tra i dati.

1.6.2 I Modelli Implementativi

Sono modelli legati all'implementazione di specifici DBMS pur con diverse varianti, si possono far rientrare in tre categorie:

B. Modelli gerarchici
C. Modelli reticolari
D. Modelli relazionali



1.6.2.1 Il Modello Gerarchico

Il modello nasce dallo sforzo congiunto di IBM e North American Aviation (oggi Rockwell international) per il programma Apollo (1965). Si concretizza nel rilascio (1969) dell'Informatico Management System/360 (IMS/360).

- I dati sono organizzati secondo un albero gerarchico, i cui nodi sono legati da associazioni IM
- Tra ogni coppia di tipi record esiste al massimo un link, il modello può essere quindi visto come un caso particolare di quello reticolare
- Il modello gerarchico non fornisce mezzi per l'implementazione diretta delle associazioni n: m risolte spesso attraverso la duplicazione dei dati.

1.6.2.2 Il Modello Reticolare

Il modello utilizza:

- tipi record per rappresentare le relazioni tra gli attributi
- link, impliciti o espliciti, per costruire le associazioni (set) tra insiemi di entità.

Le associazioni ammesse sono del tipo 1:m , ma uno stesso record può essere collegato con record di tipo diverso.

In questo modo è possibile costruire associazioni di tipo n:m.

Modello reticolare e modello relazionale sono dal punto di vista strutturale perfettamente equivalenti. Strutture reticolari e strutture relazionali possono cioè essere messe in corrispondenza 1:1.

1.6.2.3 Il Modello Relazionale

Il modello relazionale comprende tre aspetti:

- la strutture delle relazioni
- le regole di integrità sui dati delle relazioni
- le manipolazioni ammesse sulle relazioni

L'approccio ai dati è basato sul fatto che file di record che rispettano certi vincoli possono essere considerati come relazioni matematiche e pertanto ad essi si può applicare la teoria degli insiemi e il formalismo della logica matematica. Le relazioni si presentano sotto forma di tabelle, le cui righe sono a due a due diverse almeno in un elemento.

In questo modello la indipendenza dei dati è molto elevata.

1.6.2.4 Modello Relazionale - Operazioni

In un sistema relazionale mediante gli operatori relazionali, nuove relazioni possono venire create a partire da relazioni già esistenti.

a. Unione.
   L'unione di due relazioni (o tabelle) aventi uguali attributi genera una nuova relazione contenente tutti i valori delle due relazioni di partenza (equivalente all'unione tra insiemi).
b. Differenza.
   La differenza di due relazioni aventi uguali attributi genera una nuova relazione contenente tutti i valori della prima relazione che non sono contenute nella seconda relazione.
c. Prodotto cartesiano.
   Sia R1 e R2 due tabelle, il loro prodotto cartesiano R1 x R2 (leggesi R1 cartesiano R2) da come risultato una tabella contenente le colonne di R1 e R2, con valori tutte le possibili combinazioni delle n-tuple di R1 e delle n-tuple di R2.
d. Selezione.
   La selezione è la più semplice delle operazioni relazionali e genera una nuova tabella contenente tutte le colonne della prima ma solo un sottoinsieme delle righe della stessa. Tale sottoinsieme viene determinato individuando le righe in cui il valore di uno o più campi soddisfa la condizione di selezione.
e. Proiezione.
   L'operazione di proiezione genera una tabella contenente tutte le righe della prima ma solo un sottoinsieme delle colonne va notato che la tabella risultante può contenere alcune righe tra loro identiche poiché non è consentito ad una tabella di avere due o più righe tra loro uguali, solo una sarà conservata.
f. Join.
   Lo Join di due relazioni e l'unione dell'operatore di selezione e dell'operatore di prodotto cartesiano.

1.6.2.5 Lo Schema Dei Dati

Lo schema dei dati è la rappresentazione logico/concettuale delle risorse aziendali e dei legami fra le stesse che rivestono interesse conoscitivo e fanno parte del patrimonio dati aziendale definito e sviluppato progressivamente nel tempo sulla base delle esigenze informative. Lo schema dei dati:

- costituisce un modello informativo stabile e facilmente estensibile

- è indipendente dalla visione di ogni singolo utente in quanto rappresenta in modo trasparente una sintesi storica di tutte le view esistenti
- può essere tradotto in qualsiasi DBMS

Lo schema viene costruito a partire dalle relazioni ottenute al termine del processo di normalizzazione e prevede l'uso di apposite notazioni che permettono di evidenziare gli elementi fondamentali del modello:

le relazioni

i legami fra le relazioni

le chiavi primarie

le chiavi esterne.

2 La Normalizzazione

Il nucleo di un DBMS è il modello concettuale dei dati il cui disegno è compito del DBA.

Preliminare per questo lavoro è l'analisi dei dati elementari, raccolti dal D.A., da un punto di vista semantico dei dati stessi oppure partendo dalle attività che su di esse sono previste.

Ne vengono fuori particolari associazioni o relazioni tra i dati che individuano delle entità aziendali distinguibili.

Una entità è identificata da una chiave (uno o più dati elementari) e caratterizzata da uno o Più attributi.

Al fine di evitare inutili e dannose ridondanze di dati nel database, si procede ad una frantumazione delle relazioni secondo un certo formalismo di normalizzazione che avanza a livelli consecutivi.

Normalizzare uno schema significa fare in modo che esso soddisfi alcune regole che mirano soprattutto ad evitare che si creino ridondanze di dati.

Una ridondanza (cioè la duplicazione di un'informazione) è sempre pericolosa, perché ogni modifica di un dato duplicato porta invariabilmente ad un database inconsistente, a meno che non venga modificata in modo analogo anche la sua copia. Quindi per evitare ridondanze è necessario rispettare le regole di normalizzazione, che impongono alcune restrizioni sulla struttura delle tabelle.

Prima forma normale	Tutti i campi delle tabelle devono essere semplici, questo vuol dire che una colonna non può contenere una molteplicità variabile di valori.

Esempio:

COD_CLIENT	IMPORTO FATTURE			
1411	700000	34000	130000	24400
2543	900000	10500	150000	40000

La precedente tabella non rispetta la prima forma normale, la seguente sì:

COD_CLIENTE	IMPORTO FATTURE
1411	700000
1411	340000
1411	1300000
1411	1300000
2543	900000
2543	1050000
2543	1500000
2543	400000

Seconda forma normale

È in prima forma normale e se ogni tabella non contiene colonne i cui valori dipendono solo da una parte della chiave primaria.

Esempio:

COD_MAG	COD_ART	PREZZO	INDIR_MA
34901	324	12000	via Roma
44311	55	23000	via Taranto
5619	97	3400	via

Poiché il codice magazzino è legato all'indirizzo magazzino e codice magazzino e codice articolo costituiscono chiave primaria, allora la tabella non risulta in seconda forma normale. Quindi:

COD_MAG	COD_ART	PREZZO
34901	324	12000
44311	55	23000
5619	97	3400

COD_MAG	INDIR_MAG
34901	via Roma
44311	via Taranto
5619	via Venezia

Terza forma normale	Se nella tabella non esiste alcuna dipendenza funzionale tra le colonne che non fanno parte della sua chiave primaria.

Esempio*:*

C_CLIENT	NOME_CLIENT	TIPO	SCONT
3444	Rossi	Abituale	20%
321	Verdi	Saltuari	10%

Non è in terza forma normale perché esistono dipendenze funzionali tra colonne (tipo, sconto) che non partecipano alla chiave primaria (C_CLIENTE). Quindi bisogna trasformarlo in:

C_CLIENT	NOME_CLIENT	TIPO
3444	Rossi	Abituale
321	Verdi	Saltuari

TIPO	SCONT
Abituale	20%
Saltuari	10%

Si può affermare che un database in terza forma normale è un database ben strutturato comunque esistono altre forme normali più restrittive di cui per completezza diamo qui soltanto gli enunciati.

Quarta forma normale	Se ha una chiave primaria costituita da meno di tre colonne e fissato un valore di una di esse i valori delle altre colonne della chiave, dipendenti dalla prima, non possono comparire con molteplicità differenti nella tabella.
Quinta forma	Ha a che vedere con situazioni simili alla quarta laddove esista una dipendenza funzionale

normale	reciproca fra tutte le colonne che compongono la chiave.

Si fa presente infine che i problemi di normalizzazione vengono risolti automaticamente da alcuni prodotti come Access, si consiglia però di fare le opportune verifiche.

2.1 PROGETTO FISICO DEL DATA BASE

Il Progetto Fisico del data base riguarda le modalità di registrazione dei dati sui supporti di memorizzazione (es. dischi).

L'obiettivo da perseguire è quello di ottimizzare le performance delle applicazioni e l'impiego degli spazi sui supporti fisici.

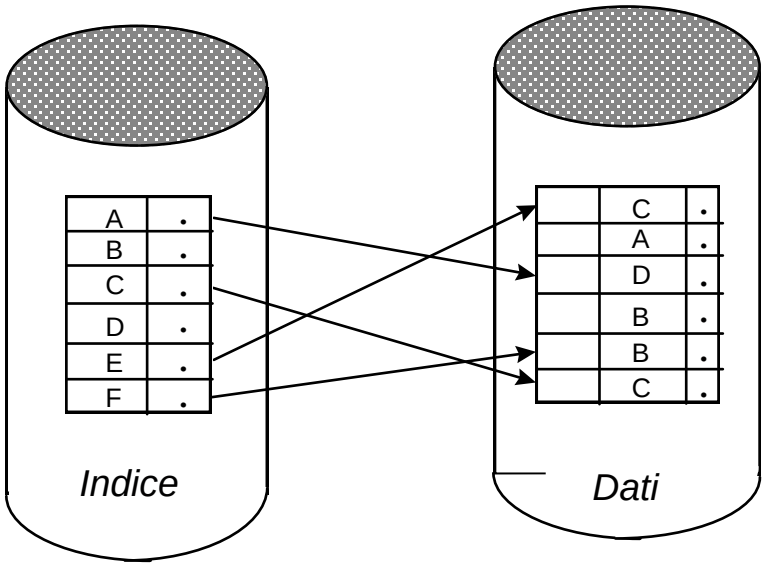
Normalmente le strutture del Data Base vengono allocati su uno o più files del sistema operativo ospite. Il DBMS utilizzerà parte delle funzioni di file/device management offerte dal S.O., alle quali aggiungerà specifiche modalità di gestione degli spazi, degli accessi ai dati, della concorrenza fra gli utenti, ecc.

Il numero e la dimensione dei files utili per memorizzare il DB viene determinato in funzione di varie esigenze: volume di dati da gestire, agevolazione delle operazioni di backup, opportunità di separare i dati di aree applicative differenti.

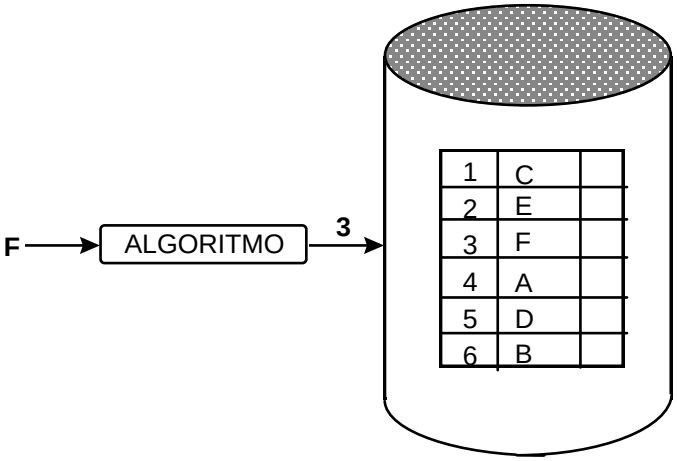
In linea di principio qualsiasi tipo di files su disco può essere utilizzato per costruirvi le strutture necessarie al funzionamento del DBMS. Le organizzazioni fisiche che si ritrovano più frequentemente sono:

STRUTTURE A INDICI: l'accesso ai records avviene tramite uno o più indici che mantengono l'associazione tra la "chiave" identificativa e la posizione fisica del record.

STRUTTURE A INDICI

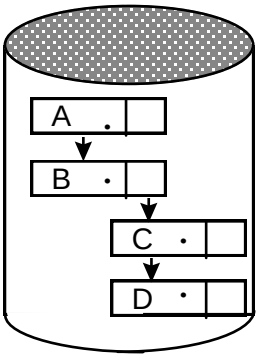


STRUTTURE HASHING



STRUTTURE HASHING: le chiavi identificative dei records vengono associate alle posizioni fisiche tramite algoritmi di "randomizzazione".

STRUTTURE CONCATENATE



STRUTTURE CONCATENATE: i records sono collegati tra loro per mezzo di gruppi di puntatori.

Ciascuna organizzazione fisica presenta vantaggi e svantaggi, soprattutto in funzione dei tipi di accesso utilizzati.

Le strutture a indici, con numerose varianti, sono tra le più utilizzate poichè, oltre all'accesso diretto, consentono anche di mantenere i dati ordinati secondo i valori delle chiavi. Comportano però un impegno aggiuntivo di spazio per gli indici, e un tempo aggiuntivo sia per reperire i dati (scansione-indice) che per aggiornarli (modifica- indici).

Le strutture di tipo hashing permettono un accesso diretto molto veloce, ma non consentono di vedere i records in modo ordinato e devono far fronte "collisione" di indirizzi che gli algoritmi di randomizzazione inevitabilmente generano.

Le organizzazioni di tipo concatenato facilitano il reperimento dei records logicamente collegati fra loro, ma sono penalizzate in fase di aggiornamento del Data Base, poiché ogni aggiunta o eliminazione di record richiede l'aggiornamento dei pointers in tutte le catene di appartenenza.

Alcuni DBMS consentono un uso combinato di più tipi di organizzazioni fisiche, permettono la scelta più idonea nei diversi casi.

2.2 FIGURE PROFESSIONALI

Concorrono diverse figure professionali alla realizzazione e utilizzo di un data base:

- IL DATA BASE ADMINISTRATOR (DBA) La personalizzazione e la manutenzione del DBMS in funzione delle esigenze applicative è compito specifico del DBA. (uno specialista o un gruppo di specialisti

di sistemi data base). Esso sovrintende alla realizzazione e manutenzione del Data Base, al controllo degli utenti e alla ottimizzazione complessiva del sistema. Il DBA detiene perciò le competenze più ampie possibili circa la struttura e il funzionamento del data base.

- L'ANALISTA EDP si occupa del progetto del data base e delle specifiche applicative e di fattibilità.
- Il PROGRAMMATORE APPLICATIVO ha il compito di realizzare i programmi che inseriscono ed elaborano i dati nel Data Base, nell'ambito di procedure ben definite e di uso ricorrente. Per svolgere tale compito si avvale sia di linguaggi tradizionali 3GL come il C, sia di strumenti di sviluppo che assicurano una più elevata produttività come i linguaggi 4GL e i generatori di applicazioni.
- L'UTENTE FINALE è una persona che usufruisce dei servizi offerti dal DBMS per eseguire specifiche operazioni e transazioni richieste dal proprio lavoro, oggi lo si vede più direttamente coinvolto nella realizzazione di quelle elaborazioni di interrogazione ed analisi dei dati aziendali, che servono a supportare le decisioni inerenti la sua specifica attività. Ciò viene reso possibile dalla disponibilità di linguaggi e di interfacce grafiche di accesso ai dati molto semplici ed immediate, che permettono di estrarre dal Data Base le informazioni volute, senza codificare programmi.

3 Come affrontare e risolvere un problema reale

Si riprendono alcuni concetti fondamentali per meglio affrontare il problema.

Il modello Entità-Relazioni è un modello molto semplice gli elementi base sono tre:

- **Entità:** rappresentate graficamente come rettangoli
- **Relazioni:** rappresentate graficamente come rombi
- **Attributi:** rappresentate graficamente dalle voci legate alle relazioni.

Un'entità rappresenta un elemento del mondo reale, come una persona un'automobile o una fattura. Ogni entità è dotata di una serie di attributi, ognuno dei quali può assumere un valore all'interno di un determinato dominio. Una persona, per esempio può essere dotata di un attributo denominato età, che può assumere qualsiasi valore nel dominio 0-130.

- **Chiave:** attributo o insieme di attributi che identificano univocamente un'entità

Per esempio l'attributo codice fiscale potrebbe costituire la chiave dell'entità persona.

Una relazione è un legame che coinvolge due o più entità: per esempio, un'automobile *appartiene ad* una persona. Ogni relazione, inoltre ha una cardinalità.

Tenendo presente i concetti sopra sintetizzati cerchiamo di risolvere il seguente esercizio. Si fa presente che è solo una soluzione ma ne esistono diverse.

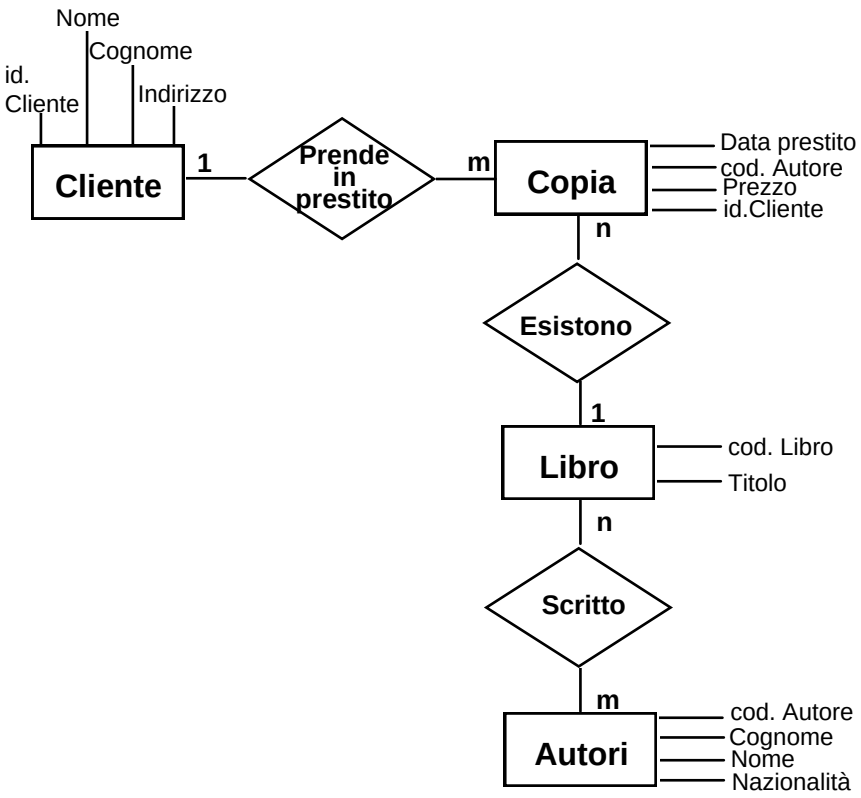
- **Caso di studio:**

L'applicazione deve gestire una biblioteca, memorizzando i libri con i relativi autori e gestendo i prestiti effettuati ai clienti.

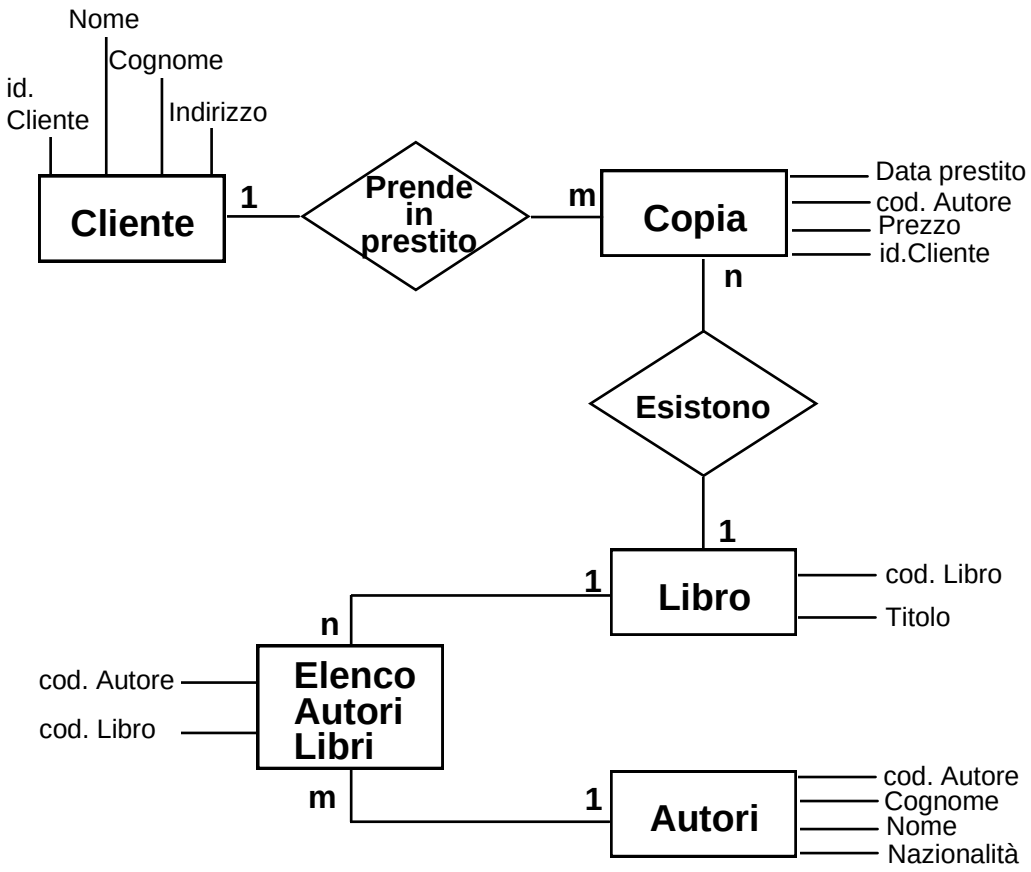
Soluzione:

- Primo passo: Suddividere la realtà di interesse in uno o più domini.
- Secondo passo: Individuare le entità all'interno di ogni dominio.
- Terzo passo: Individuare le relazioni
- Quarto passo: Definizione degli attributi e scelta delle eventuali chiavi primarie di ogni entità.

L'obbiettivo è quello di ottenere un modello completo del sistema in esame. Una prima implementazione grafica risulta essere la seguente.


Nome
Cognome
id.
Cliente
Indirizzo
Cliente
1
Prende
in
prestito
m
Copia
Data prestito
cod. Autore
Prezzo
id.Cliente
n
Esistono
1
Libro
cod. Libro
Titolo
n
Scritto
m
Autori
cod. Autore
Cognome
Nome
Nazionalità

È sempre meglio procedere alla normalizzazione prima di passare alla traduzione dal modello Entità-Relazioni a quello relazionale, quindi come prima cosa, è necessaria l'eliminazione delle relazioni N:M. Risulterà quindi il seguente grafico:



A questo punto si può passare alla traduzione del diagramma Entità-Relazioni a quello relazionale.

I principali passi da seguire per effettuare la traduzione sono:

- Per ogni entità, viene creata una tabella.
- Per ogni relazione binaria, occorre introdurre all'interno di una delle due tabelle i campi che costituiscono la chiave esterna.
- Per ogni relazione binaria di cardinalità N:M, e per ogni relazione che coinvolga più di due entità, occorre creare una nuova tabella, i cui campi siano tutti e soli i campi che costituiscono la chiave primaria nelle entità coinvolte.

3.1 JOIN

Una Join è un operazione che combina righe di una o più tabelle viste o snapshots. Viene eseguita una join ogniqualvolta più tabelle compaiono nella clausola FROM di un comando SELECT. Per ulteriori dettagli sul linguaggio SQL si fa riferimento ai manuali relativi.

3.1.1 Join Semplici

Il tipo più comune di Join, SIMPLE-JOIN o EQUI-JOIN, restituisce righe da due o più tabelle basandosi sulla condizione di uguaglianza.
Per semplicità ci baseremo sul caso di due tabelle per mostrarne la sintassi:

```
SELECT tabella colonna,......FROM tabella1, tabella2
WHERE tabella1. Colonna = tabella2.colonna;
```

3.1.2 Outer Join

Le outer join estendono il risultato di una simple-join.
In alcuni casi è rilevante non perdere nella join la nozione di mancata corrispondenza (OUTER JOIN).

L'outer join può mantenere i valori per cui non esiste corrispondenza per una, per l'altro o per entrambe le tabelle.

Esempio:

```
SELECT tabella.colonna FROM tabella1, tabella2
WHERE  tabella1.colonna = tabella2.colonna (+)
```

oppure:

```
SELECT tabella.colonna FROM tabella1, tabella2
WHERE  tabella1.colonna (+) = tabella2.colonna
```

Nella clausola WHERE il nome della colonna che non è seguito dal simbolo (+) sarà totalmente visualizzata, mentre dall'altra saranno visualizzati solo i record che troveranno corrispondenza.

Un esempio per chiarire:

```
SELECT ENAME, EMP.DEPTNO, DNAME
FROM EMP, DEPT
WHERE EMP.DEPTNO (+) = DEPT.DEPTNO
```

L'output è il seguente:

ENAME	DEPTNO	DNAME
CLARK	10	ACCOUNTING
KING	10	ACCOUNTING
MILLER	10	ACCOUNTING
SMITH	20	RESEARCH
SCOTT	20	RESEARCH
JONES	20	RESEARCH
ADAMS	20	RESEARCH
FORD	20	RESEARCH
		OPERATIONS
ALLEN	30	SALES
BLAKE	30	SALES
MARTIN	30	SALES
TURNER	30	SALES
JAMES	30	SALES
WARD	30	SALES

4 Il Linguaggio SQL

Questo manuale descriverà il linguaggio SQL standard per la creazione, l'accesso e la manipolazione dei dati.

L'introduzione all'SQL inizia con una breve descrizione dei database relazionali, corredata da una gamma di esempi che dimostreranno la semplicità e il potere di SQL.

4.1 CENNI SUGLI OPERATORI RELAZIONALI

I database relazionali si basano su degli operatori principali quali:

OPERATORE	DESCRIZIONE
SELEZIONE	(Select) estrae una n-pla da una relazione.
PROIEZIONE	(Project) estrae tutti i valori di uno o più attributi in una relazione eliminando le n-ple che risultano duplicate.
UNIONE	(Union) è l'insieme delle n-ple appartenenti ad una, o ad entrambe, le relazioni compatibili.
INTERSEZIONE	(Intersect) è l'insieme delle n-ple che appartengono ad entrambe le relazioni.
JOIN	(Join) connette tra loro delle relazioni.
SOTTRAZIONE	(Minus) applicato a due relazioni compatibili, produce la relazione composta da tutte le tuple che appartengono alla prima, ma non alla seconda.
PRODOTTO	(Times) produce il prodotto cartesiano di due relazioni.

Le ultime tre operazioni possono essere costruite a partire dalle primitive e sono considerate comunque di base.

4.1.1 Select come operatore

L'operazione `SELECT` è l'operatore con il quale si possono effettuare estrazioni mediante richiesta di dati (query).

La sintassi base è la seguente:

```
SELECT <attributi>
FROM <nome-tabella>
[WHERE <condizione>]
```

SELECT è l'operatore di base per effettuare qualsiasi tipo di operazione relazionale enunciata in precedenza.

4.1.2 unione

Sintassi:

```
<query1>
<UNION>
<query2>
. . .
<UNION>
<queryN>
```

Nel nostro caso avremo:

```
SELECT ename, job, sal FROM  emp
UNION
SELECT loc FROM dept ORDER BY 1;
```

4.1.3 Proiezione

Sintassi:

```
SELECT <attributo> FROM <nome tabella>
```

Nel nostro esempio:

```
SELECT distinct dname FROM dept;
```

si otterrà:

DNAME
accounting
research
sales
operations

Cioè viene prodotta una sola colonna della tabella originaria.

4.1.4 Intersezione

La sintassi:

```
SELECT<attributi>
FROM <tabella/e>
WHERE <condizioni>
INTERSECT
SELECT<attributi>
FROM <tabella/e>
WHERE <condizioni>
```

Si avranno delle n-ple in output soltanto se dalle due `SELECT` vengono generate delle n-ple comuni.

4.1.5 Join

Nei database relazionali la capacità di unire più tabelle tra loro è fondamentale. In SQL, questa capacità si ottiene comparando i valori di una colonna di una tabella con i valori di una colonna di un'altra tabella. Per esempio per trovare i nomi e le località di tutti i dipendenti che fanno parte del settore 30 bisogna effettuare un'operazione (relazionale) JOIN sulle righe della tabella DEPT con quelle della tabella EMP che hanno lo stesso valore di DEPTNO.

Il comando per eseguire tale operazione è:

```
SELECT LOC, ENAME
FROM DEPT, EMP
WHERE EMP.DEPTNO = DEPT.DEPTNO
AND EMP.DEPTNO = 30
```

e si otterrà:

LOC	ENAME
Chicago	allen
Chicago	ward
Chicago	martin
Chicago	turner
Chicago	james

L'operazione JOIN può essere effettuata anche su più di due tabelle contemporaneamente. Tramite l'operazione JOIN si possono effettuare anche altri tipi di congiunzioni quali:

```
3 EQUI JOIN;
```

```
3 OUTER JOIN;
3 CROSS JOIN.
```

4.1.5.1 EQUI JOIN:

La condizione imposta per il JOIN è l'uguaglianza :

```
. . .
WHERE EMP.DEPTNO = DEPT.DEPTNO
```

come abbiamo già visto nell'esempio sopra.

4.1.5.2 OUTER JOIN:

Aggiunge alle righe prodotte dall'EQUI JOIN tutte quelle righe di una delle tabelle che non hanno corrispondenze (match) con alcuna riga dell'altra.

4.1.5.3 CROSS JOIN:

Produce il prodotto cartesiano delle due tabelle prese in considerazione. Viene realizzato effettuando la selezione senza imporre condizioni con WHERE.

4.2 INTRODUZIONE A SQL

Le applicazioni che hanno l'esigenza di comunicare con database diversi, possono riscontrare problemi nella gestione e condivisione dello schema dei dati, nella stessa consultazione, nella manipolazione dei dati e nelle operazioni amministrative dei dati stessi. Altro problema è quello legato alla gestione dei diversi livelli di utenza: l'utente finale e l'amministratore possono, con SQL, utilizzare lo stesso linguaggio per le operazioni suddette avendo ,su gli stessi dati, privilegi diversi.

SQL, acronimo di Structured Query Language, è diventato linguaggio standard per la gestione dei database relazionali. Tuttavia non è un linguaggio procedurale ma solo un linguaggio dichiarativo, infatti al suo interno non sono presenti strutture di controllo come If...Then...Else...End , While ...,For ..., etc. In pratica con SQL viene definito COSA si vuole cercare ma non COME ciò deve essere fatto.

4.2.1 Un po' di storia

Nel 1974 al laboratorio IBM venne elaborato un tipo di linguaggio chiamato SEQUEL (Structured English QUEry Language), primo linguaggio relazionale. Nelle successive elaborazioni del linguaggio, venne prodotto il SEQUEL/2 rinominato poi SQL, sul quale IBM basò il progetto di ricerca che portò alla

realizzazione del database relazionale SYSTEM R . Il successo del SYSTEM R portò alla commercializzazione di prodotti come SQL/DS nel 1981 e DB2 nel 1983. Negli anni successivi arrivò anche la standardizzazione del linguaggio da parte dell'ANSI che creò un linguaggio denominato RDL (relational Database Language). Nel 1987 fu pubblicata la specifica dello standard SQL da parte dell'ISO.

4.2.2 SQL per punti

Alcune delle caratteristiche che hanno portato al largo uso di SQL.

- Linguaggio dichiarativo non procedurale per la manipolazione dei database;
- Lavora in maniera interattiva e non sono previste routine batch;
- Si basa su una visione tabellare dei dati dove le colonne sono gli attributi e le righe sono le istanze degli attributi
- Può essere considerato anche un linguaggio HOST di un linguaggio procedurale.

E' un linguaggio facile e versatile che si presenta come interfaccia virtuale al database, viene utilizzato dall'amministratore del sistema e dal singolo utente, permettendo sia sofisticati comandi di amministrazione che semplici operazioni di richiesta di dati (query).

Sintassi SQL:

```
              VERBO SQL
Clausola 1
              Clausola 2
                    Clausola n
```

SELECT empno, sal		Verbo SQL (Select)
FROM emp		Clausola1 (From)
WHERE depno = 10		Clausola2 (Where)
ORDER BY empno		Clausola3 (Order by)

4.2.3 I verbi SQL

I verbi più utilizzati dal linguaggio SQL sono i seguenti:

VERBO	DESCRIZIONE
SELECT	utilizzato per effettuare le interrogazioni (query).
UPDATE	utilizzato per modificare i dati delle tabelle già esistenti.
DELETE	utilizzato per cancellare i dati presenti nel database.
INSERT	utilizzato per inserire dei nuovi dati nel database.
CREATE	il comando crea le tabelle, le views, gli indici del database.
DROP	cancella la struttura base delle tabelle, delle viste e degli indici del database.
ALTER	modifica la struttura della tabella
COMMIT	utilizzato per comunicare al sistema di portare a termine la transazione corrente.
ROLLBACK	annulla la transazione corrente.
GRANT	assegna autorizzazioni per l'utilizzo delle tabelle o delle views.
REVOKE	annulla le autorizzazioni precedentemente consentite.

4.3 I COMANDI SQL

Diamo di seguito la lista dei comandi SQL divisi per categorie:

- Comandi di Data Definition Language (DDL)
- Comandi di Data Manipulation Language (DML)
- Comandi di Data Control Language (DCL)
- Comandi di Transaction Control

4.3.1 Comandi di Data Definition Language:

La tabella qui di seguito riporta tutti i comandi del DDL, comandi che servono cioè a definire tabelle, indici, profili ,utenti e strutturazione interna dei dati.

COMANDI		
Alter Cluster	Create Database	Drop Index
Alter Database	Create Database Link	Drop Package
Alter Function	Create Function	Drop Procedure
Alter Index	Create Index	Drop Profile
Alter Package	Create Package	Drop Role

Alter Procedure	Create Package Body	Drop Rollback Segment
Alter Profile	Create Procedure	Drop Sequence
Alter Resource Cost	Create Profile	Drop Snapshot
Alter Role	Create Role	Drop Snapshot Log
Alter Rollback Segment	Create Rollback Segment	Drop Synonym
Alter Sequence	Create Schema	Drop Table
Alter Snapshot	Create Sequence	Drop Tablespace
Alter Snapshot Log	Create Snapshot	Drop Trigger
Alter Table	Create Snapshot Log	Drop User
Alter Tablespace	Create Synonym	Drop View
Alter Trigger	Create Table	Noaudit
Alter User	Create Tablespace	Rename
Alter View	Create Trigger	Truncate
Analyze	Create User	Update
Audit	Create View	
Comment	Drop Cluster	
Create Cluster	Drop Database Link	
Create Controlfile	Drop Function	

4.3.2 Comandi di Data Manipulation Language:

Sono necessari per l'immissione e la modifica dei dati nelle tabelle. Si hanno i seguenti:

SELECT

INSERT

UPDATE

DELETE

4.3.3 Comandi di Data Control Language:

Sono necessari per qualificare gli utenti ed i processi per garantire o meno all'accesso alle tabelle del database.Sono:

```
GRANT
REVOKE
SET ROLE
```

4.3.4 Comandi Transaction Control (transizione e concorrenza):

Fondamentali nella gestione delle transazione sul database, permettono la corretta prosecuzione ed eventualmente conclusione di una o più operazioni di inserimento o aggiornamento dei dati. Abbiamo:

```
COMMIT
ROLLBACK
SAVEPOINT
LOCK TABLE
```

4.4 DDL - DATA DEFINITION LANGUAGE

Per acquisire più familiarità con i comandi sopra citati, qui di seguito sono riportati alcuni esempi di comandi DDL.

4.4.1 Il comando create

Il comando create è utilizzato per generare e definire le tabelle:

```
CREATE TABLE dept(deptno number(2),
dname char(14),loc char(13));
```

Sono gestiti i seguenti tipi di dati:

TIPO	DESCRIZIONE
CHAR(n)	stringa di caratteri alfanumerici di lunghezza massima n;
VARCHAR(n)	stringa di caratteri alfanumerici di lunghezza corrispondente alla lunghezza effettiva del dato, ovviamente non superiore ad n;
SMALLINT	numero compreso tra -32767 a +32767;

INTEGER	numero compreso tra -2.147.483.647 e +2.147.483.647;
DECIMAL(n, m)	numero reale, con al massimo n cifre di cui m decimali;
FLOAT	numero reale in notazione esponenziale;

4.4.2 Il Comando Alter

Per modificare la tabella creata in precedenza si può usare il comando ALTER:

```
ALTER TABLE dept
     MODIFY (dname CHAR(25))
Per esempio:
UPDATE emp
SET job='MANAGER', SAL = SAL + 1000000
WHERE ename='FORD';
```

Con questa modifica il sig. Ford assume un incarico manageriale con un aumento di stipendio di £1000000

4.4.3 Il Comando Drop

DROP consente di eliminare le tabelle, le view e gli indici che precedentemente sono state create con il comando SQL Create.

esempio :

```
DROP TABLE dept;
```

Il comando DROP ha la funzione di eliminare sia la struttura delle tabelle che le strutture delle le viste ed agisce anche se nelle tabelle sono contenuti dei dati, che vengono quindi cancellati.

4.5 DML - DATA MANIPULATION LANGUAGE

Vediamo ora alcuni esempi di comandi DML.

4.5.1 Il Comando Select

Il comando SQL Select si usa per ricercare e visualizzare i dati contenuti in una tabella o in una vista. Questo comando copre le funzionalità delle otto operazioni relazionali (SELECT, PROJECT, JOIN, UNION, INTERSECT,

MINUS, TIMES) viste in precedenza. Si noti che il comando SQL Select non ha lo stesso significato dell'operazione relazionale SELECT.

La sintassi del comando SQL Select è:

```
SELECT <ATTRIBUTI DA SELEZIONARE>
FROM <NOME TABELLA>
WHERE <CONDIZIONE DI SELEZIONE SUI DATI>;
```

Il carattere "*" (asterisco) utilizzato al posto di <ATTRIBUTI DA SELEZIONARE>, permette di selezionare contemporaneamente tutti gli attributi di una tabella. Nelle condizioni di selezione che coinvolgono dei dati di tipo char ORACLE è case-sensitive.

4.5.1.1 Operatori logici e derivati

La <CONDIZIONE DI SELEZIONE> del comando SELECT può essere una combinazione logica di condizioni elementari.

Questa può essere realizzata con gli operatori logici standard AND, OR e !=(NOT equal) e utilizzano gli operatori <, >, =, <=, >=, <> per i confronti. Select può essere usato anche su tabelle di tipo view.

L'uso di Distinct come opzione fa sì che ogni istanza delle righe di output sia unica. (Si noti che questa è una caratteristica intrinseca dell' 'operatore relazionale' Select, ma viene implementata esternamente nel 'comando' Select, appunto, con l'opzione Distinct.)

Esempio:

```
SELECT DISTINCT   job
FROM emp;
```

nel quale vengono mostrati tutti i possibili job della tabella Emp.

La condizione verificata dalla clausola WHERE può essere costruita anche con l'utilizzo di operatori quali:

- BETWEEN - stabilisce un range di valori finiti, specificati nella condizione tramite l'ausilio dell'operatore AND.

Esempio:

```
SELECT  ename, sal
FROM emp
WHERE sal BETWEEN 1200000 AND 1400000;
```

si ottiene così:

ENAME	SAL
ward	1250000

martin	1250000
miller	1300000

- IN - seleziona solamente le righe che soddisfano la condizione che compare nella lista, ad esempio:

```
SELECT *
FROM emp
WHERE job IN ('MANAGER', 'ANALYST'))
```

- LIKE - permette di selezionare le righe che trovano corrispondenza ad un certo pattern.

Nel pattern possono essere utilizzati due metacaratteri particolari:

'%'(percento) sostituisce qualunque stringa di caratteri che è compresa nella stringa selezionata;

'_'(sottolineatura) sostituisce esattamente un singolo carattere.

Esempio:

```
SELECT *
FROM emp
WHERE ename like 'm_l%';
```

si ottiene:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7934	miller	clerk	7782	23-Gen-82	1300000		10

IS NULL - viene usato per selezionare righe che contengono campi con valore NULL(nullo).

Esempio:

```
SELECT * FROM emp
WHERE comm IS NOT NULL;
```

Il risultato sarà:

EMPNO	ENAME	JOB		HIREDATE	SAL	COMM	DEPTNO
7499	allen	salesman		20-Feb-81	160000	300000	30
7521	ward	salesman		22-Feb-81	125000	500000	30
7654	martin	salesman		20-Set-81	125000	140000	30

7844	turner	salesman		08-Set-81	150000	0	30

4.5.1.2 Altre Funzioni Disponibili Con Il Comando Sql Select

Il comando SQL SELECT può essere usato sia per elaborare che per richiamare dati.

Può essere specializzato con l'uso delle funzioni incorporate (Built-in functions). Per esempio, la funzione `COUNT` fornisce il numero di righe in una tabella che soddisfano una data condizione.

Per cercare il numero di righe nella tabella EMP, il comando è:

```
SELECT COUNT(*)
FROM emp;
```

La funzione `COUNT` può essere usata anche con la clausola `DISTINCT`.

Il comando:

```
SELECT COUNT (DINSTINCT ename)
FROM emp;
```

Altre funzioni incluse nello standard SQL sono:

- SUM
- AVG
- MAX
- MIN

Per esempio:

```
SELECT SUM (sal)
FROM emp;
```

Allo stesso modo, sostituendo nella prima riga del comando, SUM(<campo>) con AVG(<campo>) oppure MAX(<campo>) o ancora MIN(<campo>) si otterranno come risultati, rispettivamente, la media matematica, il valore massimo ed il valore minimo di <campo> contenuti nella tabella EMP.

Il software addizionale di Oracle contiene le funzioni STDDEV e VARIANCE che rilevano la Deviazione Standard e Varianza di una determinata serie di valori appartenenti ad una colonna. Queste funzioni non sono incluse comunemente in altri prodotti SQL.

Il comando COL di SQL*Plus può essere usato per ottenere delle migliorie sull'output.

Per esempio il comando SQL*Plus:

```
COLUMN SUM(sal) HEADING Salary_Sum
```

genererà un comando SQL:

```
SELECT SUM(sal)
FROM EMP;
```

e il risultato dell'intestazione Sal(Sum).

E' ovviamente possibile ottenere risultati multipli da una singola funzione.

Per raggruppare i componenti della tabella secondo una determinata logica, si può usare l'operatore SQL GROUP BY.

Il comando potrebbe essere il seguente:

```
SELECT ename,SUM(sal)
FROM emp
GROUP BY ename;
```

L'operatore `GROUP BY` riorganizza logicamente la tabella specificata nella parte `FROM` del comando in gruppi determinati dalle colonne specificate in `GROUP BY`. Di conseguenza, nella richiesta precedente, la tabella EMP è suddivisa in gruppi di righe con lo stesso valore di ENAME. La funzione specificata nella linea di `Select` è quindi eseguita su ciascuno dei gruppi.

4.5.2 Il Comando Insert

`Insert` si usa per inserire nuove righe in una tabella.

La sintassi è la seguente:

```
INSERT INTO <NOME-TABELLA>  VALUES
(<VALORI-PER-OGNI-COLONNA>);
```

Considerando la tabella DEPT del nostro esempio, dopo la sua creazione, si sarebbero digitati i seguenti comandi:

```
INSERT INTO dept VALUES (30, 'Sales', 'Chicago');
```

- Le virgole separano i valori; l'ordine dei valori da inserire corrisponde all'ordine di creazione delle colonne.
- Gli apici, che racchiudono i nomi dei clienti e delle città, indicano che i valori sono di tipo testuale.
- Valori numerici non richiedono l'uso di apici.
- Le righe delle tabelle vengono visualizzate nell'ordine in cui sono state inserite.

Una variante del comando Insert consente di inserire le righe specificando valori solo per alcune colonne.

La sintassi è la seguente:

```
INSERT INTO <NOME-TABELLA>
(<COLONNA 1>, <COLONNA 2>, ... )
VALUES
(<VALORE-COLONNA1>,<VALORE-COLONNA 2>, ... );
```

In ogni caso dovrà essere rispettato il vincolo di NOT NULL quando esiste.

4.5.3 Il comando Update

Il comando SQL UPDATE è usato per aggiornare il contenuto di righe precedentemente inserite.

La sintassi è la seguente:

```
UPDATE <nome tabella>
SET <assegnamenti>
[WHERE<condizioni>]
```

4.5.4 Il comando delete

Il comando SQL `DELETE` viene utilizzato per rimuovere righe da una tabella. Tale comando non influisce sulla definizione della tabella. Di conseguenza, si possono rimuovere tutte le istanze dalla tabella con il comando `DELETE`, ma la struttura della tabella continua ad esistere finché non viene usato il comando `DROP TABLE`(elimina tabella).

La sintassi è la seguente:

```
DELETE FROM dept;
```

in questo caso vengono cancellate tutte la n-ple della tabella.

Invece con :

```
DELETE FROM dept
WHERE deptno = 40;
```

verrà cancellata solo la o le n-ple che hanno come deptno il numero 40.

4.6 DCL - DATA CONTROL LANGUAGE

Vediamo ora alcuni esempi di comandi DCL.

4.6.1 Il comando Grant

Sintassi

```
GRANT object_privilege [(column[,column[,…]])]
ON object
TO {user | role | PUBLIC}
[WITH GRANT OPTION];
```

4.6.2 Il comando revoke

Sintassi

```
REVOKE {object_privilege [, object_privilege[, ..]] | ALL}
ON   object
FROM       {user | role | PUBLIC}
[CASCADE CONSTAINTS];
```

4.7 TCL - TRANSACTION CONTROL LANGUAGE

Vedremo qui di seguito alcuni esempi per la comprensione dei comandi TCL.

4.7.1 Il comando commit

Questo comando fa in modo di segnalare che la transazione sia avvenuta con successo.

La sintassi:

```
COMMIT WORK oppure COMMIT
```

Il comando può essere eseguito in modo automatico, al termine di una transazione, tramite la clausola `AUTOCOMMIT ON/OFF`.

```
SET AUTOCOMMIT ON
SET AUTOCOMMIT OFF
```

L'esecuzione di questo comando fa in modo che vengano memorizzate definitivamente tutte le operazioni di manipolazione (Insert, Update, Delete) che sono appena state effettuate.

4.7.2 Il comando rollback

Questo comando viene usato per ripristinare la situazione precedente, annullando tutte le operazioni effettuate dall'ultimo `COMMIT`.

Le sintassi possibili sono:

```
ROLLBACK
ROLLBACK WORK
ROLLBACK TO SAVEPOINT <nome-savepoint>
```

4.7.3 Il comando SAVEPOINT

Durante le fasi di lavoro, spesso si ha l'esigenza, a causa di qualche errore di inserimento o manipolazione, di dover tornare ad un determinato punto del lavoro effettuato. Il comando SAVEPOINT, permette di indicare dei riferimenti lungo l'attività svolta, in modo da poterci ritornare in seguito. Un SAVEPOINT viene richiamato con il comando di ROLLBACK; il SAVEPOINT ha la stessa funzione dei segnalibri. Questo comando potrebbe portare l'utente a confondersi con il COMMIT: la differenza tra i due è nel fatto che il COMMIT archivia i comandi dati mentre il SAVEPOINT permette di fare a ritroso i passi effettuati senza archiviazione.

```
SAVEPOINT <nome-ripristino>
```

4.8 RISOLUZIONI AMBIGUITÀ DEI NOMI DELLE COLONNE

Uno degli errori più frequenti è l'ambiguità sul nome delle colonne. Il più delle volte infatti, nel dare il comando, si deve ritornare sui propri passi per specificare su quale tabella cercare quella determinata colonna. Se si considerata la seguente interrogazione:

```
SELECT ename , job, sal, deptno
FROM emp, dept
WHERE deptno = deptno;
```

La select darà sicuramente un errore, dato che nella condizione WHERE c'è due volte lo stesso nome di colonna. Per la suddetta ambiguità il campo DEPTNO verrebbe ricercato per due volte nella stessa tabella. Per aiutare il motore SQL nella risoluzione della richiesta è possibile effettuare delle modifiche per rendere più comprensibile l'interrogazione quali:

- Durante la selezione dei campi si può specificare da quale tabella desideriamo che il campo venga prelevato.
- Nella condizione WHERE è opportuno specificare, anteponendolo al campo, il nome della tabella.

Per rendere più leggibile l'istruzione, nel caso in cui il nome della tabella sia troppo lungo, è buona norma definire un alias per tale nome ed utilizzarlo facendolo precedere al nome del campo. Un esempio di assegnazione di alias potrebbe far corrispondere alla tabella ANAGRAFICA_CLIENTI l'alias A_CLI

Vediamo ora la struttura dell'istruzione:

```
SELECT a.ename,a.job,a.sal,b.deptno
FROM emp a, dept b
WHERE a.deptno=b.deptno;
```

Il risultato dell'interrogazione è:

ENAME	JOB	SAL	DEPTNO
blake	manager	2450000	10
clark	manager	2450000	10
miller	clerk	1300000	10
king	president	5000000	10
smith	clerk	800000	20
Jones	Manager	2975000	20
scott	analyst	3000000	20
adams	clerk	1100000	20
Ford	analyst	3000000	20
allen	salesman	1600000	30
martin	salesman	1250000	30
james	clerk	950000	30
turner	salesman	1500000	30
ward	salesman	1250000	30

4.9 SUBQUERY (NESTED QUERIES)

SQL fornisce un modo facile di scrivere richieste all'interno di altre richieste.

Per esempio, per cercare i nomi di tutti quei dipendenti che lavorano nello stesso dipartimento di 'ALLEN', si potrebbe formulare la seguente interrogazione:

```
SELECT deptno,ename FROM emp
WHERE deptno=(
     SELECT deptno
     FROM emp WHERE ename='allen')
```

che restituisce:

DEPTNO	ENAME
30	allen

30	ward
30	martin
30	turner
30	james

La seconda affermazione SELECT viene definita come Sottorichiesta (Subquery). Una sottorichiesta è una ramificazione interna, cioè un'altra richiesta, interna alla richiesta principale e viene racchiusa tra parentesi tonde. La sintassi che regola la subquery è la stessa delle richieste standard, e il risultato è lo stesso - viene visualizzata la serie di righe che soddisfano la richiesta data.

Quindi, nella prima affermazione, sono riportati solo quegli ENAME dei DEPTNO ai quali appartengono alle righe in cui il numero del dipartimento è lo stesso di ALLEN.

Il predicato IN può essere utilizzato in senso negativo facendolo precedere dal predicato NOT.

Per esempio, per cercare l'EMPNO di quei DEPT che non lavorano nel dipartimento di 'DALLAS', dobbiamo digitare:

```
SELECT DISTINCT empno
FROM emp
WHERE deptno NOT IN
(SELECT deptno
FROM dept
WHERE loc = 'dallas');
```

Tale richiesta può essere ramificata in livelli multipli.

Se si desidera cercare i nomi accompagnati dal numero di DIPENDENTI elencati dal precedente comando, potrebbe essere data la seguente istruzione:

```
SELECT ename
FROM emp
WHERE empno IN
(SELECT empno FROM emp
WHERE deptno IN
(SELECT deptno
FROM dept
WHERE loc = 'dallas'));
```

Questa richiesta potrebbe anche essere scritta con un operazione JOIN,

per esempio:

```
SELECT DISTINCT empno
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND loc = 'dallas';
```

Qualche richiesta JOIN può essere riscritta come richiesta ramificata (ma non tutte le richieste ramificate possono essere riscritte utilizzando un operazione JOIN). Entrambe le richieste sono corrette e la scelta di utilizzare l'una o l'altra è puramente soggettiva.

4.9.1 Le subquery con predicati

Nell'interrogare le tabelle, spesso si va incontro a richieste particolari che possono essere risolte con l'uso dei cosiddetti predicati delle subquery. Questi restituiscono soltanto un determinato range di valori. Qui di seguito vengono elencati:

- ANY - i valori della colonna specificata vengono confrontati con ciascuno dei valori restituiti dalla subquery; la condizione risulta vera solo se almeno uno dei confronti ha avuto successo. Esempio:

```
SELECT ename, sal
FROM emp
WHERE sal>ANY
(SELECT sal
FROM emp WHERE deptno = 10);
```

- EXISTS - per suo tramite si possono effettuare controlli su l'esistenza di almeno una riga che soddisfi specifiche condizioni, ad esempio:

```
SELECT ename, job
FROM emp
WHERE EXISTS
(SELECT deptno
FROM dept WHERE deptno = 40
AND deptno = 20);
```

- **ALL** - i valori della colonna specificati vengono confrontati con ciascuno degli elementi restituiti dalla subquery; la condizione risulta vera solo se tutti i confronti hanno avuto successo. Esempio:

```
SELECT ename,  sal
FROM emp
WHERE sal>ALL
(SELECT sal
FROM emp WHERE deptno = 30);
```

5 Appendice A (SQL*PLUS)

SQL*Plus è un tool di ORACLE che permette di eseguire in maniera interattiva comandi e programmi SQL, comandi e programmi PL/SQL, e direttive interne di SQL*Plus. Scopo di questo paragrafo, è di dare al lettore gli strumenti necessari per lo svolgimento delle esercitazioni di SQL*Plus.

Per poter lavorare con SQL*Plus è necessario che l'istanza di Oracle sia avviata e inoltre si possegga un nome di utente con relativa password che permetta il collegamento al database.

Avviando SQL*Plus dalla relativa icona e dopo aver dato l'identificativo dell'utente (nome/password) SQL*Plus è pronto a ricevere i primi comandi:

```
SQL>_
```

Ora si possono dare i seguenti comandi:

- Comandi SQL
- Istruzioni PL/SQL
- Direttive SQL*Plus ausiliarie (START, EXIT, HOST, QUIT ...)

Quando SQL*Plus viene avviato, ricerca nella directory utente il file *login.sql* e, se esiste, esegue i comandi contenuti in esso. E' importante sapere che i comandi possono essere scritti indifferentemente in maiuscolo o in minuscolo. Nel caso in cui i comandi SQL*Plus vadano su due o più righe, possono essere immessi andando a capo dopo il carattere trattino (-).

5.1 DIRETTIVE DI SQL*PLUS.

Come già detto anche SQL*Plus ha i suoi comandi o meglio le sue direttive. Qui di seguito sono elencate e corredate da alcuni esempi.

Per terminare ed eseguire una istruzione immessa si usa il punto e virgola (;), oppure sulla riga successiva dare il comando RUN o alternativamente / ; l'immissione di una riga vuota (la *4 dell'esempio sottostante)* termina l'input del comando e restituisce il prompt SQL*Plus.

```
SQL> SELECT <alfa>, <beta>
2    FROM <gamma>
3    WHERE <alfa> = 10
4
SQL> /
```

Per eseguire un file contenente istruzioni PL/SQL dare il comando START nome_file oppure il nome del file preceduto dalla @.

```
SQL> START <nome_file>;
```

```
SQL> @<nome_file>;
```

SQL*PLUS permette la memorizzazione dell'ultimo comando tramite una memoria buffer, che è possibile editare per modificarne il contenuto; si possono utilizzare le seguenti direttive:

ABBREVIAZIONE	COMANDO	DESCRIZIONE
L	List	Per listare una o più linee del buffer
L n m	List condizionata	Lista dalla linea n alla linea m
C	Change	Per modificare la linea corrente C/stringa_vecchia/stringa_nuova
A	Append	Per aggiungere il testo alla linea corrente
I	Input	Per inserire le righe dopo la linea corrente
DEL	Delete	Per cancellare la linea corrente
`DESC`	`Nome Tabella`	Mostra la struttura di una tabella

E' possibile salvare il suddetto buffer su un file esterno, con il comando `SAVE` <nome del file>, se non viene specificato alcuna estensione al file viene dato di default ".SQL", la direttiva ha diverse opzioni.

```
SQL> SAVE <nome_file> CREATE;
SQL> SAVE < nome_file > REPLACE;
SQL> SAVE < nome_file > APPEND;
```

Nel primo caso il file viene creato ex-novo; nel secondo viene sostituito se già esistente; nell'ultimo il contenuto del buffer viene appeso al file esistente.

Altro comando che utilizza il buffer è il `GET` che permette di caricare il contenuto del file nel buffer.

```
SQL> GET <nome_file> LIST;
SQL> GET <nome_file> NOLIST;
```

La prima riga permette di avere anche il listato del file, mentre la seconda non lo permette.

E' importante inoltre ricordare che se non viene dato alcun suffisso viene preso quello di default ".SQL".

Per abbandonare il programma basta digitare EXIT o QUIT.

```
SQL> EXIT;
SQL> QUIT;
```

Durante la sessione SQL*Plus, può tornare utile dare un comando del sistema operativo su cui si lavora, come una "dir" per verificare la presenza di un file: ciò è possibile grazie al comando `HOST`, oppure utilizzando i caratteri `!` e `$`.

```
SQL> HOST-<nome-comando>;
```

```
SQL> !<nome-comando>;
SQL> $<nome-comando>;
```

Anche se si è collegati con il nome di un determinato utente, durante la fase di lavoro è possibile cambiare la connessione con il comando CONNECT seguito dall'account.

```
SQL> CONNECT username/password;
```

Se durante l'inizializzazione dell'istanza sono state caricate le apposite tabelle, l'utente può chiedere un supporto con il comando `HELP` seguito dalla direttiva, o comando SQL.

```
SQL> HELP <verbo-SQL | argomento>;
```

In molti casi si ha l'esigenza di dover ripetere una serie di comandi: è possibile farlo con l'utilizzo di un editor, che permette la creazione di file richiamabili con il comando `START`. L'editor di SQL*Plus è richiamabile con il comando `ED` nella finestra di editor vengono scritti i comandi e alla fine salvati o nel file di default che si chiama *"Afiedt.buf"* oppure in un file con un qualsiasi altro nome mnemonico.

```
SQL> ED <nome-file>;
```

SQL*Plus permette di rendere parametriche le istruzioni SQL, con la semplice aggiunta di un carattere al nome della colonna, della tabella o della costante in esame. Il carattere da utilizzare è la &, che messa davanti al parametro permette di richiede all'utente durante l'esecuzione del comando, l'immissione interattiva del dato. L'azione parametrica può essere controllata durante l'esecuzione con la clausola `SET VERIFY ON` oppure `SET VERIFY OFF`.

```
SQL> SET VERIFY ON
SQL> SELECT deptno, ename, job, sal
2    FROM EMP
3    WHERE deptno = &deptnum AND job = '&jobn';
```

Viene richiesto il valore numerico di deptnum e la stringa di jobn

```
Enter value for deptnum: 20
Enter value for jobn: CLERK
old 3:     WHERE deptno = &deptnum AND job = '&jobn';
new 3:     WHERE deptno = 20 AND job = 'CLERK';
```

Viene visualizzato quindi il risultato

Nel caso in cui non è attiva la verifica:

```
SQL> SET VERIFY OFF
SQL> SELECT deptno, ename, job, sal
```

```
2    FROM EMP
3    WHERE deptno = &deptnum AND job = '&jobn';
```

Viene richiesto il valore numerico di deptnum e la stringa di jobn

```
Enter value for deptnum: 20
Enter value for jobn: CLERK
```

e quindi prodotto direttamente il risultato dell'interrogazione.

Per controllare il valore delle variabili allocate in un buffer di SQL*Plus, si usa il comando `DEF`.

```
SQL> DEF <nome della variabile>;
```

E' possibile realizzare un altro tipo di istruzione parametrica con il comando `ACCEPT` che permette all'utente di chiedere, prima dell' istruzione, il valore della variabile.

```
SQL> ACCEPT <nome variabile> tipo variabile(NUMBER o CHAR) PROMPT
<'messagio'>;
```

6 IL Linguaggio PL/SQL

PL/SQL è l'estensione procedurale del linguaggio SQL di ORACLE, esso lo arricchisce dei vantaggi dei linguaggi procedurali. PL/SQL è una componente integrata in molti prodotti Oracle. Per chi sviluppa applicazioni. Client/Server e per chi amministra Oracle la conoscenza di questo linguaggio è obbligatoria perché il suo uso consente di ridurre in modo significativo il traffico tra Client e server e di eseguire programmi compilati in maniera più veloce degli statement SQL non compilati.

6.1 STRUTTURA A BLOCCHI

PL/SQL è un linguaggio strutturato a blocchi. Cioè le unità base (procedure, funzioni, ecc...) che costituiscono un programma PL/SQL sono blocchi logici.

Tipicamente, ogni blocco logico corrisponde a un problema o a un sottoproblema che deve essere risolto.

Un blocco (o sottoblocco) è collegato al relativo gruppo logico tramite dichiarazioni di variabili. Le dichiarazioni sono locali al blocco e cessano di esistere quando il blocco termina, come si vede successivamente:

```
 [DECLARE
  ..........]
BEGIN
  .....
    [DECLARE
       ............
                BEGIN
       ..........
                END]
...........
[EXCEPTION
END
```

La prima viene detta dichiarativa, in cui gli oggetti possono essere dichiarati. Una volta dichiarati, gli oggetti possono essere manipolati nella parte Begin-End. L'Exception (ovvero quella che in PL/SQL gestisce le eccezioni o condizioni di errore). L'unica parte necessaria è la parte Begin-End.

Si può definire una sottoprocedura locale nella parte dichiarativa di ogni blocco. Comunque si può chiamare una sottoprocedura locale solo nel blocco in cui è stata definita.

6.2 ARCHITETTURA

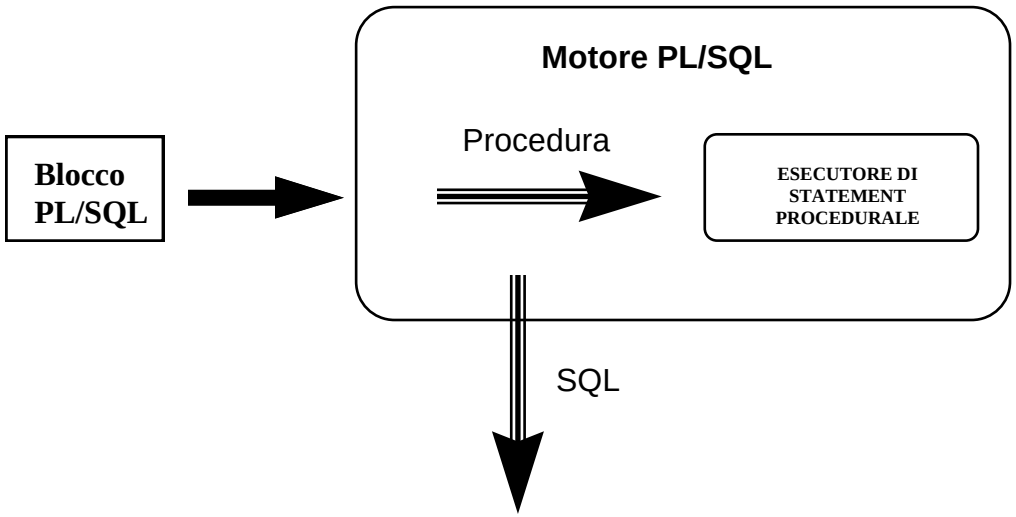
Il runtine del PL/SQL è un modulo incluso nei Tools o nel Server Oracle.

Quindi PL/SQL può risiedere in:

- Server ORACLE (Se è stata attivata una Procedural Option)
- ORACLE tools (Es. Forms 4.5, Report 2.5, . .)

PL/SQL potrebbe essere disponibile nel Server e nei Tools o solo in uno dei due.

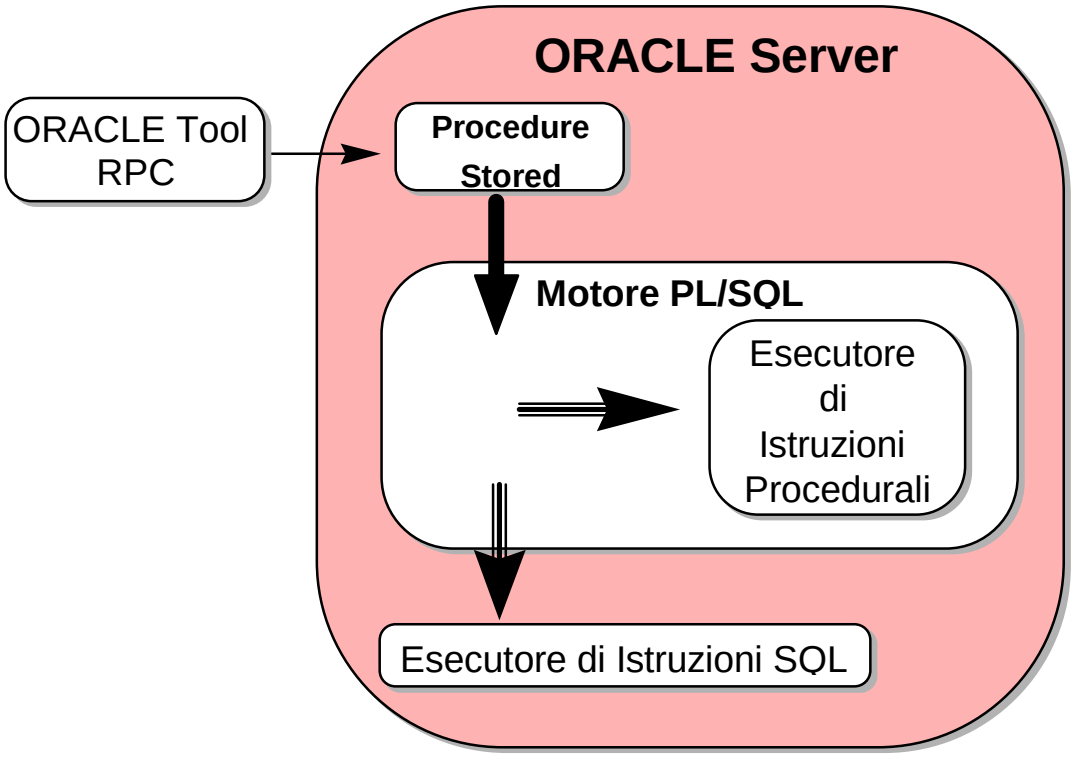
Quando è disponibile nei Tools il motore PL/SQL accetta come input ogni blocco o sottoprogramma e lo elabora, come mostrato nella successiva figura.



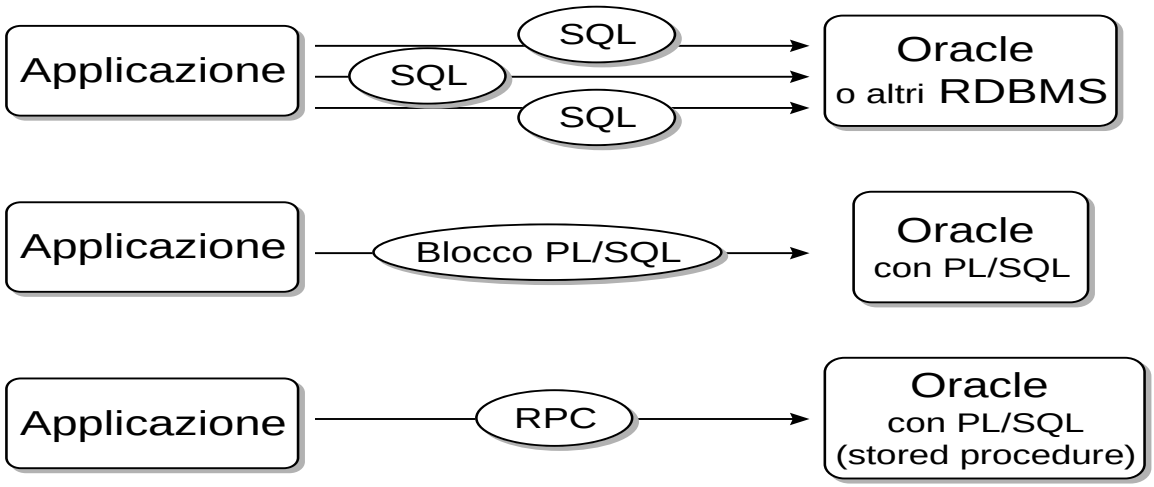
Il "motore" PL/SQL, dopo aver suddiviso le istruzioni dei blocchi esegue la parte procedurale e manda la parte SQL al Server ORACLE o ad un altro RDBMS.

Quando un Server ORACLE contiene il motore PL/SQL (Procedural Option), può eseguire blocchi e sottoprogrammi PL/SQL.

- Package, procedure o funzioni possono essere memorizzate dentro l'applicazione dove sono richieste (locali). Questi non si possono utilizzare da un'altra applicazione, ma esistono solo all'interno del blocco entro il quale vengono definiti.
- Packege, procedure o funzioni possono essere memorizzate direttamente sul Server Oracle (Tablespace System), queste diventano utilizzabili, dopo la connessione, da più applicazioni (stored procedure)



Il linguaggio PL/SQL è stato concepito sia per completare con i costrutti procedurali il linguaggio SQL che per migliorare le prestazioni in ambiente Client/Server.



6.3 BLOCCO ANONYMOUS

Un blocco Anonymous è un programma senza nome. Per esempio, si può inviare la procedura DELETE CUSTOMER come blocco senza nome al server quando si usa SQL*Plus, come illustriamo qui sotto:

```
SQL>DECLARE
SQL> errnum      NUMBER := -20000;
SQL> errmess     VARCHAR2(2000) := 'Errore Standard';
SQL>BEGIN
SQL> DELETE FROM customer WHERE id = 12;
SQL> IF SQL%NOTFOUND THEN    -- se non trova il cliente errore
SQL> SELECT errornumber, errormessage INTO errnum, errmess
SQL> FROM  userserrors
SQL> WHERE errormessage LIKE ('Invalid cust%');
SQL> raise_application_error (errnum, errmess);
SQL> END IF;
SQL>EXCEPTION
SQL> WHEN NO_DATA_FOUND THEN -- se non trova un messaggio
     -- d'errore usa quello standard
SQL> raise_application_error (errnum, errmess);
SQL>END delete customer;
SQL>/
```

Bisogna osservare che usando SQL*Plus terminare la scrittura del blocco anonymous, bisogna inserire un punto (.) ad inizio riga dopo l'ultimo statement END.

Un'applicazione client invia un blocco anonymous al server di database con una singola operazione di I/O, riducendo di molto il traffico che si avrebbe inviando uno statement alla volta.

D'altra parte, ogni volta che il server riceve un nuovo blocco anonymous deve compilarlo prima di eseguirlo.

I blocchi anonymous sono utili nel caso di una procedura nuova che si vuole eseguire solo una volta.

6.4 VARIABILI E COSTANTI

6.4.1 Dichiarazioni di variabili

I tipi di variabili comprendono sia quelli di SQL: CHAR, DATE, e NUMBER sia quelli più specifici del PL/SQL come BOOLEAN e BINARY_INTEGER.

Supponiamo di voler definire la variabile part_no di quattro caratteri di tipo numerico e, una variabile chiamata in_stock che assume i valori booleani TRUE e FALSE.

Abbiamo:

```
part_no    NUMBER(4);
in_stock   BOOLEAN;
```

Si possono anche dichiarare record e tavole PL/SQL usando RECORD e TABLE che sono tipi di dati composti.

6.4.2 Assegnare un valore ad una variabile

Si può assegnare un valore ad una variabile in due modi.

Il primo, utilizzando l'operatore di assegnazione (:=), come nei seguenti esempi:

```
tassa := prezzo *  valore_rata;
bonus := salario_corrente * 0.10;
validità := FALSE;
```

Il secondo modo, quello di assegnargli il completo risultato di una SELECT o una FETCH fatte su una tabella.

```
SELECT sal * 0.10 INTO bonus FROM emp WHERE empno = emp_id;
```

6.4.3 Tipi di dati PL/SQl

TIPI	SOTTOTIPI	DESCRIZIONE
NUMBER(Precisione, Scala)	DEC, DECIMAL, DOUBLE_PRECISION, FLOAT, INT, INTEGER, NUMERIC, REAL, SMALLEST	Numeri interi o floating-point. La precisione di default è 38, ma sono validi tutti i valori da 0 a 38. La scala di default è 0 e sono validi tutti i valori da -84 a 127. Se il valore di scala è negativo, Oracle arrotonda tutti i valori a sinistra della virgola.
BINARY_INTEGER	NATURAL, POSITIVE	Interi con segno da -2^31-1 a 2^31-1. L'uso di questi tipi è vantaggioso perché evita le conversioni per i calcoli.
CHAR (dimensione)	CHARACTER, STRING	Stringhe a lunghezza fissa, al massimo di 32767 byte
VARCHAR2 (dimensione)	VARCHAR	Stringhe a lunghezza variabile, al massimo di 32767 byte
DATE		Date, ore, minuti o secondi
BOOLEAN		Valori logici TRUE o FALSE
RECORD		Tipi di record definiti dall'utente
TABLE		Tabelle PL/SQL

6.4.3.1 Tipi di dati composti

Il PL/SQL ha due tipi di dati composti:

- TABLE
- RECORD

- **TABLE**

Oggetti di tipo Table sono chiamate tabelle PL/SQL. Le tabelle PL/SQL sono ad una sola colonna e usano una chiave primaria di tipo BINARY INTEGER per accedere ai dati.

Viene definita come segue:

```
DECLARE
  TYPE tavola IS TABLE OF CHAR(10)
       INDEX BY BINARY INTEGER;
  tav1 tavola;
BEGIN
  ...
END;
```

Possono utilizzare gli attributi, per riprendere definizioni di una tabella già esistente (vedi dopo).

- **RECORD**

Un Record può contenere tipi di dati diversi, ma logicamente correlati tra loro.

In PL/SQL un tipo Record viene definito come segue:

```
DECLARE
   TYPE tipo_rec IS RECORD
            (deptno NUMBER(2),
             dname CHAR(14),
             loc CHAR(13));
   var1     tipo_rec;
BEGIN
    ...
END;
```

E' inoltre possibile definire un tipo di dati Record utilizzando gli attributi (vedi dopo).

6.4.4 Dichiarazione delle Costanti

Dichiarare una costante è come dichiarare una variabile basta aggiungere la parola CONSTANT e immediatamente assegnare ad essa un valore.

Come nell'esempio dove assegniamo un valore alla variabile minimo:

```
minimo CONSTANT  REAL := 10.00;
```

6.4.5 Cursori

ORACLE utilizza aree di lavoro dette *aree SQL private* per eseguire comandi SQL e avere informazioni su queste.

Un costrutto PL/SQL chiamato *cursore* permette di dare un nome ad un'area privata SQL e accedere alle informazioni prelevate.

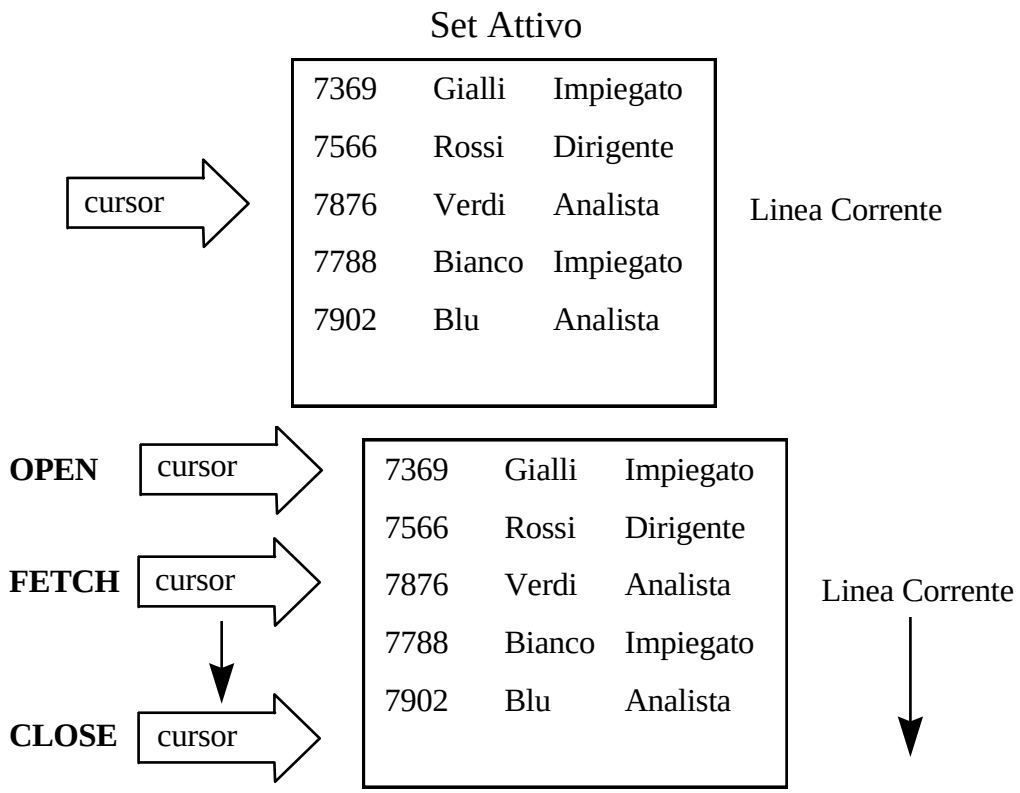
Ci sono due tipi di cursori: *impliciti* ed *espliciti*. PL/SQL implicitamente dichiara un cursore per tutti i comandi di manipolazione dati, incluso le query che danno come risposta una sola riga. Per query che hanno come risposta più di una riga si deve esplicitamente dichiarare un cursore che indica le righe individualmente.

```
DECLARE
CURSOR c1 IS SELECT num_imp, nome, lavoro FROM impiegati
     WHERE dipno = 20;
```

L'insieme di righe che ritorna come risultato della query è chiamata set attivo (*Active Set*). Facciamo un esempio di cursore esplicito che "punta" alla riga corrente del set attivo

Query:

```
SELECT imp_num, nome, lavoro FROM impiegati WHERE dipno = 20
```



La figura mostra l'uso dei comandi OPEN, FETCH, e CLOSE per controllare un cursore.

- OPEN esegue la query associata con il cursore.
- FETCH ricerca la riga corrente e sposta il cursore alla riga successiva.
- CLOSE disabilita il cursore.

Un cursore che viene utilizzato in un ciclo FOR LOOP dichiara implicitamente un suo indice di loop come un record %ROWTYPE, apre un cursore, successivamente esegue i fetch e chiude il cursore quando tutte le righe sono state analizzate.

```
DECLARE
  CURSOR c1 IS SELECT nome, sal, dipno FROM impiegati;
BEGIN
  FOR impiegati_rec IN c1 LOOP
    salari_total := salari_total + impiegati_rec.sal;
  END LOOP;
END;
```

6.4.6 Attributi

Quando si deve assegnare il tipo ad una variabile o costante lo stesso tipo di una colonna di una tabella, possiamo utilizzare gli attributi.

6.4.6.1 %TYPE

Questo tipo di attributo è usato nel caso di dichiarazione di variabile riferita a una colonna di una tabella.

Per esempio, se esiste un campo di nome titoli nella tabella libri, per dichiarare una variabile chiamata mio_titolo con lo stesso tipo_dati della colonna titoli, bisogna usare la seguente notazione:

```
mio_titolo libri.titolo%TYPE;
```

Dichiarare mio_titolo con %TYPE ha due vantaggi.

- Non bisogna conoscere l'esatto tipo della variabile
- Nel database si cambia la definizione di titolo (per esempio in una stringa maggiore), il tipo-dati mio_libro cambia automaticamente.

6.4.6.2 %ROWTYPE

In PL/SQL è possibile definire variabili composte di tipo record con l'attributo %ROWTYPE usando il tracciato Record di una tabella o di un cursore (come vedremo dopo).

Supponiamo di avere una tabella di nome dept e vogliamo dichiarare la variabile composta dept_rec:

```
DECLARE
dept_rec   dept%ROWTYPE;
........
```

Per accedere ai campi del record si usa:

```
my_deptno := dept_rec.deptno;
```

Se si definisce un cursore che contiene nome, salario, e qualifica di un impiegato si può usare l'attributo %ROWTYPE per dichiarare una variabile record che contiene le stesse informazioni:

```
DECLARE
CURSOR c1 IS SELECT nome, sal, qualifica FROM impiegati;
impiegati_rec    c1%ROWTYPE;
...
```

Quando si fa eseguire l'istruzione

```
FETCH c1 INTO impiegati_rec;
```

il valore nella colonna nome della tavola impiegati viene assegnato al campo nome di impiegati_rec, il valore della colonna sal viene assegnato al campo sal, e così via.

```
impiegati_rec.nome      contiene      Giovanni
impiegati_rec.sal       contiene      1500000
impiegati_rec.qualifica      contiene      Dirigente
```

6.4.6.3 %FOUND

Esistono due tipi di cursori: impliciti ed espliciti. PL/SQL dichiara implicitamente un cursore per tutte le manipolazioni di dati fatti con statement SQL, incluse le query di una singola riga.

Finché lo statement SQL non viene eseguito %FOUND è NULL, e ritornerà TRUE non appena ci sarà un INSERT, UPDATE, o DELETE su una o più righe, oppure non appena da una SELECT INTO ritornano una o più righe, altrimenti %FOUND ritorna FALSE.

Il seguente esempio mostra l'utilizzo dell'attributo %FOUND. Considerano due tabelle, tab_num1 e tab_num2, per ognuna di esse viene restituito un

numero, la cui somma verrà inserita in una terza tabella tab_somm, il ciclo finisce non appena le righe di una delle due tabelle sono state "fetched".

```
DECLARE
      CURSOR cur_num1 IS SELECT num FROM tab_num1
                         ORDER BY sequenza;
      CURSOR cur_num2 IS SELECT num FROM tab_num2
                         ORDER BY sequenza;
      num1               tab_num1.num%TYPE;
      num2               tab_num2.num%TYPE;
      num3               NUMBER := 0;
BEGIN
     OPEN cur_num1;
     OPEN cur_num2;
     LOOP                -- si crea il ciclo attraverso le due
                         -- tabelle e si prende la coppia di numeri
                         FETCH cur_num1 INTO num1;
                         FETCH cur_num2 INTO num2;
                         IF (cur_num1%FOUND) AND (cur_num2%FOUND) THEN
                          num3 := num3 + 1;
                          INSERT INTO tab_somm VALUES (num3, num1 + num2);
                         ELSE
                          EXIT;
                         END IF;
     END LOOP;
     CLOSE cur_num1;
     CLOSE cur_num2;
  END;
```

6.4.6.4 %ISOPEN

Oracle chiude automaticamente il cursore dopo l'esecuzione associata agli statement SQL, in questo caso %ISOPEN è valutato FALSE.

Per query che ritornano più righe si deve esplicitamente dichiarare un cursore; per ogni differente processo sulle righe. %ISOPEN vale TRUE se il cursore è aperto e vale FALSE in caso contrario.

```
IF NOT (cur_mio%ISOPEN) THEN
OPEN cur_mio;
```

```
    END IF;
    FETCH cur_mio INTO . . .
```

6.5 STRUTTURE DI CONTROLLO

Le strutture di controllo sono la parte più importante del PL/SQL come estensione del linguaggio SQL.

Non solo ci permettono di manipolare i dati di ORACLE, ma di usare controlli condizionali, iterativi come IF-THEN-ELSE, FOR-LOOP, WHILE-LOOP, EXIT-WHEN, and GOTO.

6.5.1 Controllo condizionale

Spesso è necessario fare delle scelte a secondo delle circostanze che si verificano. L'IF-THEN-ELSE ci permette di fare ciò.

Così come per molti altri linguaggi di programmazione procedurali dopo l'IF abbiamo la condizione; il THEN si esegue se la condizione dopo l'IF è vera; l'ELSE se la condizione dopo l'IF è falsa o nulla. Consideriamo il seguente esempio che gestisce una transazione di una banca:

```
DECLARE
    acct_balance NUMBER(11,2);
    acct         CONSTANT NUMBER(4) := 3;
    debit_amt    CONSTANT NUMBER(5,2) := 500000;
BEGIN
    SELECT bal INTO acct_balance FROM accounts
              WHERE account_id = acct
    FOR UPDATE OF bal;
        IF acct_balance >= debit_amt THEN
                 UPDATE accounts SET bal = bal - debit_amt
                 WHERE account_id = acct;
        ELSE
                 INSERT INTO temp VALUES
                 (acct, acct_balance, 'Fondi Insufficienti');
                 -- inserisce account, current balance, e message
                 END IF;
    COMMIT;
END;
```

6.5.2 Controllo Iterativo

Il comando LOOP permette di eseguire più volte una serie di comandi. Basta far precedere la lista di comandi dalla parola LOOP e chiudere il ciclo con END LOOP:

```
LOOP
     -- sequenza di comandi
END LOOP;
```

Ad esempio per utilizzare le righe di un cursore:

```
LOOP
FETCH ordercursor INTO linetotal;
EXIT WHEN ordercursor%NOTFOUND;
ordertotal := ordertotal + linetotal;
END LOOP;
```

Il comando FOR-LOOP ci permette di definire un range di interi, e di eseguire una serie di comandi una volta per ogni intero.

Esempio:

```
--Aumento del 10% dei prezzi degli articoli 1,2,3,4,5
FOR i IN 1..5 LOOP
     UPDATE prodotti SET prezzo = prezzo * 1,10
     WHERE articolo = i
END LOOP;
```

Ci sono diversi modi per uscire da un LOOP; abbiamo già visto che si può usare il comando EXIT WHEN, si può anche utilizzare un IF e un EXIT, come nell'esempio seguente in cui si esce dopo aver prelevato l'ultima riga di ORDERCURSOR.

```
LOOP
     FETCH ordercursor INTO linetotal;
     IF ordercursor%NOTFOUND THEN
           EXIT;
     END IF;
     ordertotal := ordertotal + linetotal;
END  LOOP;
```

Oppure come nel seguente esempio, dove il loop termina non appena il valore di total ha superato i 10000000:

```
LOOP
     ...
```

```
            total := total + salario;
            EXIT WHEN total > 10000000;
        -- si esce dal loop se la condizione è vera
        END LOOP;
```

Il comando WHILE-LOOP associa una condizione ad una sequenza di comandi.

Prima di ogni iterazione del loop la condizione viene valutata, se la condizione è vera la sequenza di comandi viene eseguita se è falsa o nulla il loop viene saltato e il controllo passa al comando successivo.

Nel successivo esempio si cerca gerarchicamente il primo impiegato che ha un salario maggiore di 4000000.

```
DECLARE
   salario       impiegati.sal%TYPE;
   capo_num      impiegati.mgr%TYPE;
   ultimo_nome   impiegati.nome%TYPE;
   num_imp_inizio      CONSTANT NUMBER(4) := 7902;
                           -- impiegato di più basso livello
BEGIN
   SELECT sal, mgr INTO salario, capo_num FROM impiegati
                   WHERE num_imp = num_imp_inizio;
   WHILE salario > 4000000 LOOP
        SELECT sal, mgr, nome INTO salario, capo_num, ultimo_nome
                     FROM impiegati WHERE num_imp = capo_num;
   END LOOP;
   INSERT INTO temp VALUES (NULL, salario, ultimo_nome);
   COMMIT;
END;
```

6.5.3 Goto

Consente di proseguire l'esecuzione del programma da un punto indicato da una label, che è un identificatore tra doppie parentesi angolari, come mostrano i seguenti esempi:

```
-- esempio 1
IF rating > 90 THEN
  GOTO calc_raise;
END IF;
<<calc_raise>>
IF job_title = 'SALESMAN' THEN
  raise := commission * 0.25;
ELSE
  raise := salary * 0.10;
END IF;
-- esempio 2
<<orderitemloop>>
LOOP
  FETCH ordercursor INTO linetotal;
  IF ordercursor%NOTFOUND
    THEN EXIT;
  END IF;
  ordertotal := ordertotal + linetotal;
END LOOP orderitemloop;
altri comandi
IF ...THEN
  GOTO orderitemloop;
END IF;
```

6.6 La Gestione Degli Errori

E' opportuno prevedere le possibili condizioni di errore e pianificare il programma in modo che reagisca correttamente.

PL/SQL come altri linguaggi di programmazione ha un meccanismo specifico che gestisce le condizioni di errore separatamente dalla normale elaborazione del programma.

Il gestore delle eccezioni o condizioni d'errore sono definiti nel blocco EXCEPTION di un programma PL/SQL.

Ci sono due tipi di exception: quelle predefinite e quelle definite dall'utente.

Le exception predefinite sono automaticamente mandate al gestore del sistema runtime. Per esempio se cerchiamo di dividere un numero per zero l'exception predefinita ZERO_DIVIDE va a creare automaticamente una condizione di errore richiamando il gestore.

Le exception definite da un utente devono essere mandate al gestore esplicitamente usando il comando di RAISE.

6.6.1 Exception predefinite

Le exception predefinite gestiscono le comuni condizioni di errore, ad esempio il tentativo di eseguire una statement SQL senza prima essersi collegati ad un database.

Elenchiamo ora le exception predefinite.

Tabella:

EXCEPTION	DESCRIZIONE
CURSOR_ALREADY_OPEN	Apertura di un cursore già aperto
DUP_VAL_ON_INDEX	Valore duplicato in una colonna a valori unici
INVALID_CURSOR	Riferimento a un cursore non valido o operazione di cursore non valida
INVALID_NUMBER	Valore non numerico dove è richiesto un valore numerico
LOGIN_DENIED	Collegamento non attivato
NO_DATA_FOUND	Nessun dato come risultato di un SELECT INTO
NOT_LOGGED_ON	Assenza di collegamento a Oracle
PROGRAM_ERROR	Errore interno di PL/SQL
STORAGE_ERROR	Errore di memoria PL/SQL
TIMEOUT_ON_RESOURCE	Timeout nell'attesa di una risorsa
TOO_MANY_ROWS	Più righe restituite da uno statement SELECT INTO
TRANSACTION_BACKED_OUT	Un server remoto ha annullato la transazione
VALUE_ERROR	Errore aritmetico, di conversione, di troncamento o di vincolo
ZERO_DIVIDE	Divisione per 0

Nel seguente programma si vede come usare le exception NO_DATA_FOUND e TOO_MANY_ROWS.

```
CREATE FUNCTION customerid (last IN VARCHAR2, first IN VARCHAR2)
  RETURN INTEGER AS
       --  Questa  function  restituisce  l'ID  di  un  cliente  dato
       -- nome e cognome.
  custumerid    INTEGER;
  errnum        INTEGER := -20000;
  errmess       VARCHAR2 (2000) := ' Errore Standard ';
BEGIN
  SELECT id INTO customerid FROM  customer
          WHERE lastname = last AND firstname = first;
  RETURN customerid;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
         -- se non trova il cliente restituisce errore
    SELECT errornumber, errormessage INTO errnum, errmess
      FROM usererrors WHERE errormessage LIKE ('Invalid cust%');
    raise_application_error (errnum, errmess);
  WHEN TOO_MANY_ROWS THEN   -- se ci sono più clienti allora ...
    SELECT errornumber, errormessage INTO errnum, errmess
      FROM usererrors WHERE errormessage LIKE ('Multiple cust%');
    raise_application_error (errnum, errmess);
  WHEN OTHERS THEN
         -- per errori non definiti messaggio standard
    raise _application_error (errnum, errmess);
END customerid;
```

Quando il programma incontra una condizione di errore, si interrompe e trasferisce il controllo al *gestore della exception* corrispondente nel blocco EXCEPTION ed esegue il relativo codice.

Nell'esempio precedente della CUSTOMERID, se lo statement SELECT non trova un record cliente con il nome e il cognome indicati, il programma lancia l'exception NO_DATA_FOUND, che recupera un numero di errore.

Nel blocco EXCEPTION c'è un gestore OTHERS per tutte le condizioni di errore che non sono esplicitamente previste dagli altri gestori.

6.6.2 Exception definite dall'utente

Il programmatore può definire delle exception per gestire eventi particolari come errrori.

Esempio: calcolo di un bonus di un venditore. Il bonus è basato sul salario e sulle commissioni di questo venditore. Se la commissione è nulla allora si aziona l'exception comm_persa.

```
DECLARE
   salario       NUMBER(7,2);
   commissione   NUMBER(7,2);
   comm_persa    EXCEPTION;  -- dichiarazione di exception
BEGIN
   SELECT sal, comm INTO salario, commissione FROM impiegati
                   WHERE imp_num := imp_id;
   IF commissione IS NULL THEN
      RAISE comm_persa;   -- raise exception
   ELSE
      bonus := (salario * 0.05) + (commissione * 0.15);
   END IF;
EXCEPTION    -- inizio exception utilizzata
   WHEN comm_persa THEN
   -- process error
END;
```

6.6.2.1 La Procedura RAISE_APLICATION_ERROR

Con l'utility RAISE_APPLICATION_ERROR si possono creare numeri e messaggi di errori personali.

Ci possono essere fino a 1000 numeri di errore definiti dall'utente, da -20000 a - 20999.

Se si pensa di usare questa utility estensivamente, conviene centralizzare i numeri e i corrispondenti messaggi in una tabella del database per mantenere la coerenza di tutte le applicazioni. Vediamone un esempio:

```
CREATE TABLE usererrors
                (errornumber NUMBER (5,0) PRIMARY KEY,
                errormessage VARCHAR2 (2000));
INSERT INTO usererrors VALUES (-20000, 'Errore Standard');
INSERT INTO usererrors VALUES (-20001, 'Numero identificativo
                                cliente non valido');
```

```
CREATE PROCEDURE deletecustomer (cust_id IN NUMBER) AS
    errnum       NUMBER := -20000;
    errmess      VARCHAR2 (2000) := 'Errore Standard';
BEGIN
    DELETE FROM customer WHERE id = cust_id;
    IF SQL%NOTFOUND THEN
                -- se non trova il cliente restituisce errore
      SELECT errornum, errormessage INTO errnum, errmess
                    FROM usererrors WHERE errormessage LIKE
                    ('Invalida cust%');
      raise_application_error (errnum, errmess);
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
      -- se non trova un messaggio di errore usa quello standard
      raise_application_error (errnum, errmess);
END deletecustomer;
```

Quando un programma chiama la RAISE_APPLICATION_ERROR si lancia una exception, e restituisce un numero e un messaggio all'ambiente chiamante.

6.6.3 Exception non gestite

Se capita un errore non previsto esplicitamente dai gestori del blocco EXCEPTION, e non si usa OTHERS, PL/SQL annulla le operazioni eseguite nel blocco in cui si è verificato l'errore e restituisce la *exception non gestita* all'ambiente chiamante.

6.7 MODULARITÀ

Attraverso opportuni rifinimenti, si può risolvere un problema complesso in un insieme di problemi semplici che hanno una soluzione facilmente implementabile.

PL/SQL permette la scomposizione di un programma complesso in sottoprogrammi che possono essere correlati fra loro.

6.7.1 Sottoprogrammi

Perché utilizzare i sottoprogrammi:

- La produttività di chi sviluppa viene aumentata poiché i codici comuni vengono scritti, provati e corretti una sola volta e conservati in modo centralizzato, e tutte le applicazioni possono usarli.
- La performance delle applicazioni migliora perché un'applicazione chiama ed esegue un blocco di codice compilato e conservato nel database, invece che inviare un blocco non compilato di statement attraverso la rete.
- L'overhead di rete viene ridotto perché le applicazioni client inviano una chiamata a una procedura e un set di parametri invece che singoli statement SQL o un intero blocco anonymous.

6.7.1.1 Procedure

Per scrivere una procedura si utilizza la seguente sintassi:

```
PROCEDURE nome [ (parametro [ , parametro, . . . ] ) ] IS
                [ dichiarazione locale ]
BEGIN
  comandi eseguibili
[ EXCEPTION
  corpo dell' exception ]
END [nome];
```

Con i parametri si passano alle procedure e alle function i valori su cui lavorare.

I parametri devono avere un tipo che può essere un qualsiasi tipo PL/SQL.

I parametri possono essere IN, OUT, o INOUT.

- Un *parametro* IN ha un valore che viene passato alla procedura e che non può essere modificato dalla procedura.
- Un *parametro* OUT, invece non ha un valore quando la procedura viene chiamata, mentre gli viene assegnato un valore all'interno della procedura.
- Un *parametro* INOUT può avere un valore quando viene passato alla procedura e può essere modificato dalla procedura.

```
nome_var [IN | OUT | INOUT ] tipodati [ { := | DEFAULT } valore
]
```

Abbiamo già visto nello schema, che una procedura è composta da due parti: la specification e il corpo.

La parte della specification comincia con la parola chiave PROCEDURE e termina con il nome della procedura o con una lista di parametri.

La parte della dichiarazione dei parametri è opzionale, ovviamente le procedure che non hanno parametri sono scritte senza parentesi.

Il corpo della procedura comincia con la parola chiave IS o AS e termina con END seguita o meno dal nome della procedura (è opzionale); esso si divide in tre parti: parte dichiarativa, parte eseguibile, e una parte opzionale che contiene le exception.

Possiamo notare che una procedura è un vero e proprio programma in miniatura:

```
PROCEDURE prog_bonus (imp_id NUMBER) IS
  bonus         REAL;
  comm_persa    EXCEPTION;
BEGIN
  SELECT comm * 0.25 INTO bonus FROM impiegati
              WHERE impno = imp_id;
  IF bonus IS NULL THEN
      RAISE comm_persa;
  ELSE
      UPDATE stipendio SET paga = paga + bonus
           WHERE impno = imp_id;
  END IF;
EXCEPTION
  WHEN comm_persa THEN
                . . .
END prog_bonus;
```

Quando viene chiamata questa procedura, accetta un numero di impiegato, usa il numero per selezionare la commissione dell' impiegato da una tabella del database e, allo stesso tempo, calcola il 25% di bonus, dopo di che calcola l'ammontare del bonus, se il bonus è nullo viene lanciata una raise, altrimenti viene aggiornato il dato relativo allo stipendio del commesso stesso.

Facciamo un esempio di blocco anonimo PL/SQL che chiama la function CUSTOMERID e la procedure DELETECUSTOMER per cercare ed eliminare un cliente dalla tabella CUSTOMER:

```
DECLARE
  custid        NUMBER(5,0);
BEGIN
```

```
  -- assegna il valore restituito da CUSTOMERID a custid
  -- e lo usa per cancellare il cliente.
  custid := customerid ( ' Hamilton ', ' Alexander ' );
  deletecustomer (custid);
END;
```

6.7.1.2 Funzioni

Una funzione è un sottoprogramma che restituisce un valore. La sintassi della funzione è la seguente:

```
FUNCTION nome [(argomento[ , argomento,...])] RETURN tipodati IS
  [ dichiarazioni locali ]
BEGIN
  comandi eseguibili
[  EXCEPTION
  corpo dell'exception ]
END  [nome];
```

dove gli argomenti della funzione hanno la seguente sintassi:

```
nome_var [ IN | OUT | IN OUT ] tipodati [ { := | DEFAULT } valore]
```

Consideriamo la seguente funzione che ci restituisce se un impiegato riceve un giusto salario in funzione della sua qualifica.

```
FUNCTION sal_ok ( salario REAL, titolo REAL ) RETURN BOOLEAN IS
  sal_min     REAL;
  sal_max     REAL;
BEGIN
  SELECT losal, hisal INTO sal_min, sal_max FROM sal
          WHERE lavoro = titolo;
  RETURN ( salario >= sal_min ) AND ( salario <= sal_max );
END sal_ok;
```

Quando viene chiamata, questa funzione ha bisogno del salario di un impiegato e del titolo di quest'ultimo.

Questa funzione usa il titolo dell'impiegato per selezionare dalla tabella sal del database il range entro il quale dovrà stare lo stipendio della categoria a cui appartiene l'impiegato. Se il salario è fuori dal range, allora ritorna il valore booleano FALSE, altrimenti ritorna il valore TRUE.

Diversamente dalle procedure, le funzioni devono essere richiamate come parte di una espressione, come nel seguente esempio:

```
IF sal_ok (nuovo_sal, nuovo_titolo) THEN
. . .
END IF;
. . .
promozione := sal_ok (nuovo_sal, nuovo_titolo) AND (categoria> 3);
```

In entrambe i casi la funzione è in una espressione e dà come risultato una valore.

6.7.1.3 Return Statement

Il comando RETURN completa immediatamente l'esecuzione di un sottoprogramma e ridà il controllo al programma chiamante.

L'esecuzione riparte con il comando successivo alla chiamata del sottoprogramma.

L'importante è non confondere il comando RETURN con la clausola RETURN che abbiamo visto nelle funzioni, la quale specifica il tipo di dati del valore del risultato in una specification di una funzione.

Un sottoprogramma può contenere più RETURN statement, nessuno dei quali ha bisogno dell'ultimo statement lessicale, eseguendo ognuno di essi si può completare il sottoprogramma immediatamente.

Consideriamo la seguente funzione saldo:

```
FUNCTION saldo (acc_id INTEGER) RETURN REAL IS
  acc_sal       REAL;
BEGIN
  SELECT sal INTO acc_sal FROM acc WHERE accnu = acc_id;
  RETURN acc_sal;
END saldo;
```

6.7.1.4 Variabili e loro visibilità

I problemi di utilizzo di una variabile sono risolti una volta stabiliti il campo di azione (*scope*) o la visibilità.

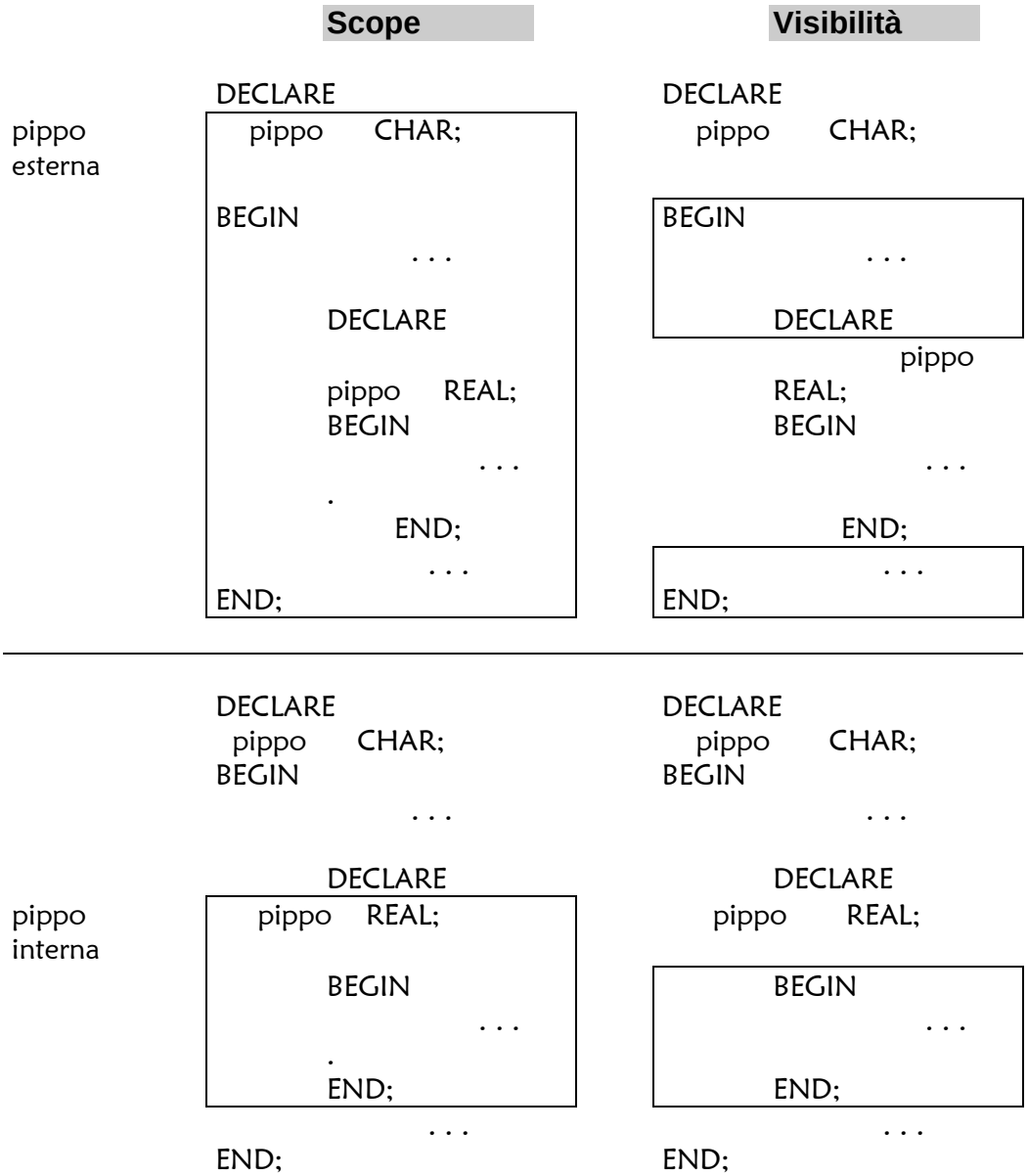
Il campo di azione di una variabile è la regione di programma (blocco, sottoprocedura, o package) da cui si può richiamare la data variabile.

Una variabile inoltre è visibile solo nelle regioni da cui si può fare riferimento ad essa usando un nome inqualificato.

Sebbene non si possa dichiarare due volte lo stesso identificatore nello stesso blocco, si può dichiarare due volte uno stesso identificatore in due blocchi differenti.

I due oggetti rappresentati dalla variabile sono distinti e un cambiamento nell'uno non influenza l'altro.

Comunque un blocco non può far riferimento a due variabili dichiarate in altri blocchi perché queste variabili sono locali e non globali rispetto al blocco.



Il seguente esempio illustra lo scope:

```
DECLARE
   A      CHAR;
```

```
  B     REAL;
BEGIN
-- variabili disponibili qui: A (CHAR), B
  DECLARE
     A     INTEGER;
     C     REAL;
  BEGIN
               -- variabili disponibili qui: A (INTEGER), B, C
  END;
  DECLARE
    D          REAL;
  BEGIN
               -- variabili disponibili qui: A (CHAR), B, D
  END;
-- variabili disponibili qui: A (CHAR), B
END;
```

Una variabile globale può essere ridichiarata in locale utilizzando lo stesso nome, in questo caso la dichiarazione locale prevale sulla globale come abbiamo appena visto, e non si può fare riferimento alla variabile locale a meno di non usare un nome qualificato, così come avviene nel seguente esempio.

```
<<esterno>>
DECLARE
  compleanno    DATE;
BEGIN
  DECLARE
    compleanno   DATE;
  BEGIN
               . . .
               IF compleanno = esterno.compleanno THEN
               END IF;
  END;
END esterno;
```

Oppure come nel successivo esempio, dove come qualificatore si prende proprio il nome della procedura:

```
PROCEDURE controllo_credito(...)  IS
   rate          NUMBER;
FUNCTION validità(...) RETURN BOOLEAN IS
   rate          NUMBER;
BEGIN
   IF controllo_credito.rate < 4 THEN
   END IF;
END validità;
BEGIN.....
END controllo_credito;
```

Una variabileviene comunque inizializzata di default una volta definita in un programma o sottoprocedura, a NULL.

6.7.1.5 Parametri

Riassumiamo nella seguente tabella quali sono le caratteristiche per i parametri IN, OUT, e INOUT.

IN	OUT	IN OUT
è di default	deve essere specificato	deve essere specificato
si passa un valore a un sottoprogramma	ridà valori al chiamante	passa un valore iniziale ad un sottoprogramma; ritorna un valore modificato al chiamante
parametro formale agisce come una costante	parametro formale agisce come una variabile non inizializzata	parametro formale agisce come una variabile inizializzata
parametro formale non gli può assegnare un valore	parametro formale non può essere usato in una espressione; gli si deve assegnare un valore	parametro formale gli si dovrebbe assegnare un valore.
parametro attuale può	parametro attuale	parametro attuale

essere una costante, una variabile inizializzata, letterale, o una espressione	deve essere una variabile	deve essere una variabile

6.7.1.5.1 Valori dei Parametri di Default

Come mostra il seguente esempio è possibile inizializzare dei parametri IN a valori di default. In questo modo si possono passare dei numeri differenti al sottoprogramma, usare o meno il parametro di default.

```
PROCEDURE creare_dip(nuovo_nomed CHAR DEFAULT ' TEMP ',
nuova_loc CHAR DEFAULT 'TEMP') IS
BEGIN
   INSERT INTO dip
   VALUES (dipnu_seq.NEXTVAL, nuovo_nomed, nuova_loc);
END creare_dip;
```

Se non viene passato nessun parametro attuale, viene usato il corrispondente parametro formale. Come si può osservare dalla seguente chiamata della procedura creare_dip.

```
BEGIN
   . . .
   creare_dip;
   creare_dip('MARKETING');
   creare_dip('MARKETING', 'ROMA');
END;
```

Nella prima chiamata alla procedura creare_dip non viene passato alcun parametro attuale, in tal caso vengono usati tutti e due i parametri di default; nel secondo caso viene usato solo il parametro di default nuova_loc, visto che l'altro è stato passato.

Nel terzo caso non viene usato alcun parametro di default.

La cosa importante è che bisogna ricordare che i parametri sono posizionali, come vedremo nei seguenti esempi.

```
creare_dip('ROMA'); -- chiamata di procedura errata
```

Non si può risolvere il problema utilizzando una semplice virgola prima del nome così come si vede nel seguente esempio.

```
creare_dip( ,'ROMA'); -- chiamata alla procedura illegale
```

In tal caso è necessario utilizzare la notazione nominale, come segue:

```
creare_dip(nuova_loc => 'ROMA');
```

6.7.1.5.2 Overloading

In PL/SQL è possibile chiamare più sottoprogrammi con lo stesso nome (*Overloading*). Supponiamo di voler inizializzare le prime n righe in due tavole PL/SQL, vengono dichiarate come segue:

```
DECLARE
  TYPE tipo_tav_dati IS TABLE OF DATE INDEX BY BINARY_INTEGER;
  TYPE tipo_tav_reali IS TABLE OF REALINDEX BY BINARY_INTEGER;
  dataass_tab   tipo_tav_dati;
  sal_tab       tipo_tav_reali;
  . . .
```

Bisogna scrivere la seguente procedura per inizializzare la tabella PL/SQL chiamata dataass_tab:

```
PROCEDURE inizializzazione(tab OUT tipo_tav_dati, n INTEGER) IS
BEGIN
  FOR i IN 1 . . n LOOP
               tab (i) := SYSDATE;
  END LOOP;
END inizializzazione;
```

Bisogna scrivere la stessa procedura per inizializzare la tabella PL/SQL chiamata sal_tab:

```
PROCEDURE inizializzazione( tab OUT tipo_tav_reali, n INTEGER) IS
BEGIN
  FOR i IN 1 . . n LOOP
               tab ( i ) := 0.0;
  END LOOP;
END inizializzazione;
```

Poiché il processo è lo stesso nelle due inizializzazioni è logico dargli lo stesso nome.

Si possono utilizzare le due procedure all'interno dello stesso blocco senza dover specificare quale delle due è, sarà poi il PL/SQL stesso che in base al tipo di parametri passati nella chiamata saprà quale delle due utilizzare, come si può osservare nell'esempio successivo.

```
DECLARE
  TYPE tipo_tav_dati IS TABLE OF DATE
     INDEX BY BINARY_INTEGER;
```

```
   TYPE tipo_tav_reali IS TABLE OF REAL
       INDEX BY BINARY_INTEGER;
   dataass_tab   tipo_tav_dati;
   comm_tab      tipo_tav_reali;
   ind           BINARY_INTEGER;
   . . .
BEGIN
   ind := 50;
   inizializzazione(dataass_tab, ind);
                 -- chiamata alla prima inizializzazione
   inizializzazione(comm_tab, ind);
                 -- chiamata alla seconda inizializzazione
   . . .
END;
```

Non è possibile definire due sottoprocedure con lo stesso nome cambiando solo il modo di passaggi dei parametri. Esempio non è possibile la seguente condizione:

```
PROCEDURE prova(n IN INTEGER) IS
BEGIN
   . . .
END prova;
PROCEDURE prova(n OUT INTEGER) IS
BEGIN
   . . .
END prova;
```

Inoltre non è possibile scrivere due sottoprogrammi con lo stesso nome se la definizione formale differisce solo per il tipo di dati appartenenti alla stessa famiglia. Per esempio:

```
PROCEDURE prova1(somma INTEGER) IS
BEGIN
   . . .
END prova;
PROCEDURE prova1(somma REAL) IS
BEGIN
   . . .
END prova1;
```

Infatti sia gli INTEGER sia i REAL appartengono alla stessa famiglia di tipi di dati.
Infine non si possono avere due funzioni che differiscono solo per il tipo di dati del RETURN, perfino se i dati sono in differenti famiglie, come segue:

```
FUNCTION acconto_ok(num_acc INTEGER) RETURN BOOLEAN IS
BEGIN . . .
END acconto_ok;
FUNCTION acconto_ok(num_acc INTEGER) RETURN INTEGER IS
BEGIN . . .
END acconto_ok;
```

6.7.1.5.3 Ricorsività

La ricorsività è una tecnica potente per la semplificazione della struttura di un algoritmo. Matematicamente la ricorsività non è niente altro che l'applicazione di una formula ai termini precedenti della successione di dati.

La successione di Fibonacci (1, 1, 2, 3, 5, 8, 13, 21 ...), che servì per illustrare la crescita di una colonia di conigli, non è niente altro che una successione di numeri basata sulla somma dei due termini che precedono il successivo.

Consideriamo la definizione del fattoriale (n!) (ovvero il prodotto di numeri interi da 1 a n):

```
n!=n*(n - 1)!
```

Un sottoprogramma ricorsivo non è niente altro che un programma che richiama più volte se stesso.

Ogni chiamata ricorsiva crea una nuova istanza di un oggetto dichiarato in un sottoprogramma, inclusi i parametri, le variabili, i cursori, e le exception, perciò deve essere strutturato in modo che abbia anche una uscita, altrimenti nel caso del PL/SQL interviene con una exception predefinita STORAGE_ERROR.

Facciamone un esempio, per calcolare il valore di 3! possiamo ragionare nel seguente modo:

```
0!= 1
1!=1*0!=1*1=1
2!=2*1!=2*1=2
3!=3*2!=3*2*1=6
```

Per implementare il semplice calcolo sopra illustrato bisogna scrivere la seguente funzione ricorsiva, che ci dà come risultato un intero positivo.

```
FUNCTION fattoriale(n POSITIVE) RETURN INTEGER IS -- ridà n!
BEGIN
  IF n = 1 THEN -- condizione per uscire
    RETURN 1;
```

```
  ELSE
    RETURN n * fattoriale(n - 1); -- chiamata ricorsiva
  END IF;
END fattoriale;
```

Ad ogni chiamata ricorsiva, n si decrementa e la chiamata ricorsiva si interrompe non appena n diventa 1.

Esistono anche le procedure mutuamente ricorsive, ovvero le procedure ricorsive che richiamano altre procedure ricorsive. Nell'esempio successivo le funzioni dispari e pari, che determinano se un numero è pari o dispari vengono chiamate direttamente.

```
FUNCTION dispari(n NATURAL) RETURN BOOLEAN IS
      -- la dichiarazione è avanti
FUNCTION pari(n NATURAL) RETURN BOOLEAN IS
BEGIN
  IF n = 0 THEN
    RETURN TRUE;
  ELSE
    RETURN dispari(n - 1);  -- chiamata mutuamente ricorsiva
  END IF;
END pari;

FUNCTION dispari(n NATURAL) RETURN BOOLEAN IS
BEGIN
  IF n = 0 THEN
    RETURN FALSE;
  ELSE
    RETURN pari(n - 1); -- chiamata mutuamente ricorsiva
  END IF;
END dispari;
```

Quando viene passato alla procedura un intero n positivo, allora vengono chiamate le funzioni pari e dispari alternativamente e ad ogni chiamata n viene decrementato. Alla fine n diventa zero e come uscita si ha TRUE o FALSE. Se per esempio passiamo il numero 4 al dispari il risultato è:

```
dispari(4)
pari(3)
dispari(2)
pari(1)
dispari(0)       -- il risultato è FALSE
```

D'altra parte se passiamo 4 alla funzione pari otteniamo:

```
pari(4)
dispari(3)
pari(2)
dispari(1)
pari(0)          -- il risultato è TRUE
```

Ogni funzione ricorsiva è possibile implementarla iterativamente. Confrontiamo le due versioni di uno stesso programma, il calcolo della successione di Fibonacci: una implementata iterativamente, l'altra ricorsivamente.

```
-- versione ricorsiva

FUNCTION Fibonacci(n POSITIVE) RETURN INTEGER IS
BEGIN
  IF (n = 1) OR (n = 2) THEN
    RETURN 1;
  ELSE
    RETURN Fibonacci(n - 1) + Fibonacci(n - 2);
  END IF;
END Fibonacci;

-- versione iterativa
FUNCTION Fibonacci(n POSITIVE) RETURN INTEGER IS
  pos1           INTEGER := 1;
  pos2           INTEGER := 0;
  somma          INTEGER;
BEGIN
  IF (n = 1) OR (n = 2) THEN
    RETURN 1;
  ELSE
```

```
      somma := pos1 + pos2;
      FOR i IN 3 . . n LOOP
        pos2 := pos1;
        pos1 := somma;
        somma := pos1 + pos2;
      END LOOP;
      RETURN somma;
  END IF;
END Fibonacci;
```

6.7.1.61 Trigger

Un trigger di database è una procedura PL/SQL associata ad una tabella.

Quando uno statement SQL manipola una tabella incontra una condizione di trigger, ORACLE esegue automaticamente il corpo del trigger.

I trigger si usano per programmare il server in modo che reagisca a specifiche situazioni, per esempio per applicare regole di integrità particolarmente complesse o regole aziendali.

Per creare un trigger si usa un comando di CREATE TRIGGER. Esempio: calcolo automatico del campo totale di inserimento o aggiornamento usando il prezzo unitario dalla tabella prezzi:

```
CREATE TRIGGER lineetotali
  BEFORE INSERT OR UPDATE OF quantità ON nome_tab FOR EACH ROW
DECLARE
  prezzo REAL;
BEGIN
  SELECT prezzounitario INTO prezzo
      FROM prezzi WHERE id := :new.stockid;
  :new.totale := :new.quantità * prezzo;
END lineetotali;
```

Si può limitare l'uso del trigger con una restrizione, cioè una condizione booleana specificata in una clausola WHEN del comando CREATE TRIGGER.

```
CREATE TRIGGER reorder
  AFTER UPDATE OF onhand, reorder ON stock
  FOR EACH ROW WHEN (new.onhand <= new.reorder)
BEGIN
  INSERT INTO reorder VALUES (:new.id, :new.onhand, :new.reorder);
```

```
END;
```

Quando si crea un trigger si possono specificare diverse opzioni che determinano le modalità con cui il corpo del trigger viene eseguito.

Si può indicare il numero di volte che si vuole eseguire il trigger per ogni statement:

- il trigger di statement
- il trigger di riga.

Il primo viene lanciato una sola volta, indipendentemente da quante righe lo statement tocca; il secondo una volta per ogni riga.

Si può anche specificare se il trigger va eseguito prima o dopo lo statement. Per esempio, il trigger lineetotali è un trigger BEFORE, il che è corretto in quanto si vuole calcolare il totale prima di inserire o modificare una riga. Invece, il trigger stockchanges è un trigger AFTER, in modo che Oracle non debba eseguire il rollback sulla tabella stockchangelog se lo statement di trigger fallisce.

- BEFORE
- AFTER

Il trigger STOCKCHANGES dell'esempio seguente inserisce una riga di log nella tabella stockchanges ogni volta che viene manipolata la tabella stock.

```
CREATE TRIGGER stockchanges
  AFTER INSERT OR UPDATE OR DELETE NO stock
DECLARE
  dmltype VARCHAR2(6);
BEGIN
              --Assegna a dmltype il tipo di statement SQL
              --(INSET, UPDATE, DELETE) eseguito sulla tabella
              --STOCK
IF INSERTING THEN
       dmltype := 'INSERT' ;
    ELSIF UPDATING THEN
       dmltype := 'UPDATE' ;
    ELSE
```

```
        dmltype := 'DELETE' ;
     END IF;
-- Inserisce username e dmltype nella tabella di log.
INSERT INTO stockchangelog VALUES (USER, dmltype);
END stockchanges;
```

- NEW
- OLD

Se si vuole una registrazione più estensiva dei cambiamenti della tabella stock, occorre scrivere stockchanges come trigger di riga:

Ci sono diverse estensioni per potenziare i trigger: i predicati condizionali e i valori di correlazione.

I predicati INSERTING, UPDATING e DELETING consentono di eseguire diversi blocchi di statement che ha lanciato il trigger.

```
CREATE TRIGGER stockchanges
  AFTER INSERT OR UPDATE OR DELETE ON stock
  FOR EACH ROW
DECLARE
  dmltype VARCHAR2(6);
BEGIN
--Assegna a dmltype il tipo di comando eseguito sulla tabella
--STOCK.
    IF INSERTING THEN
      dmltype := 'INSERT' ;
    ELSIF UPDATING THEN
      dmltype := 'UPDATE' ;
    ELSE
      dmltype := 'DELETE' ;
    END IF;
--Per ogni riga che è cambiata nella tabella STOCK, inserisce i
--valori vecchi e nuovi.
```

```
INSERT INTO stockchangelog VALUES
(dmltype,:old.id,:old.unitprice,:old.onhand,:old.reorder,:old.descript
ion,:new.id,:new.unitprice,:new.onhand,:new.reorder,:new.description);
END stockchanges;
```

I comandi di Commit e Rollback non sono ammessi in un Trigger.

6.7.1.7 Package

Procedure funzioni e altri costrutti PL/SQL possono essere creati e conservati insieme in un *package*, con un miglioramento dell'organizzazione dei programmi, della sicurezza e delle performance. Consentono la creazione di variabili globali. Un package è costituito da due parti:

- Specification
- Body

La parte *specification* definisce: l'interfaccia alle nostre applicazioni (parte pubblica); vi si dichiarano i tipi, le costanti, le variabili, funzioni, procedure disponibili per l'utilizzo.

Il parte *body* definisce: funzioni e procedure che costituiscono la specification e variabili private.

Esempio:

```
PACKAGE azioni_imp IS  -- specification del package
   PROCEDURE imp_ass(impno NUMBER, nome CHAR, ...);
   PROCEDURE imp_lic(imp_id NUMBER);
END azioni_imp;

PACKAGE BODY azioni_imp IS   -- corpo del package
   PROCEDURE imp_ass (impno NUMBER, nome CHAR, . . .) IS
   BEGIN
      INSERT INTO impiegati VALUES (impno, nome, . . .);
   END imp_ass;
   PROCEDURE imp_lic (imp_id  NUMBER) IS
   BEGIN
      DELETE FROM impiegati WHERE impno = imp_id;
   END imp_lic;
END azioni_imp;
```

Solo le dichiarazioni nella parte specification del package sono visibili e accessibili alle applicazioni, cioè sono *pubbliche*.

I dettagli nella parte del corpo (Package Body) sono nascosti e inaccessibili.

Ciò che è definito nel corpo del package ma non è dichiarato nelle specifiche è *privato* e può essere usato solo dai costrutti del package.

La prima volta che viene chiamato un package, questo viene caricato in memoria, perciò per le successive chiamate non c'è un accesso diretto a disco ovvero non è richiesto un vero e proprio I/O su disco, quindi l'utilizzo delle sue, aumenta le performance del sistema.

6.7.1.8 Le Sequenze

ORACLE ha la possibilità di definire una sequenza di numeri, che può essere usata per generare chiavi primarie, per inserire una nuova riga in una tabella. Esempio:

```
CREATE SEQUENCE nome_seq
START WITH 1
INCREMENT BY 1
```

- CURRVAL
- NEXTVAL

Esempio:

```
INSERT INTO tabella1 VALUES (nome_seq.NEXTVAL, nome, ...);
INSERT INTO tabella2 VALUES (nome_seq.CURRVAL, salario, ...);
```

6.7.1.9 Informazioni Nascoste

Con le informazioni nascoste si vedono solo i dettagli che sono rilevanti ad un determinato livello dell'algoritmo.

Le informazioni nascoste tengono le decisioni ad alto livello separate da quelle di basso livello.

6.7.1.9.1 Algoritmi

Se si implementano le informazioni nascoste attraverso gli algoritmi, una volta definito lo scopo e la parte delle specification delle procedure a basso livello, si possono ignorare i dettagli dell'implementazione.

Per esempio, l'implementazione di una procedura chiamata aum_stipendi è nascosta. Infatti ogni cambiamento della definizione di aum_stipendi è trasparente alla chiamata delle applicazioni.

7 Tipi di Oggetti

7.1 INTRODUZIONE AGLI OGGETTI

I tipi oggetto sono un particolare datatype. È possibile usarli nello stesso modo in cui si usano i datatype più comuni quali `NUMBER` o `VARCHAR2`. Per esempio, si può specificare un tipo oggetto come il datatype di una colonna in una tabella relazionale e si possono dichiarare le variabili del tipo oggetto stesso. I tipi oggetto inoltre hanno alcune differenze importanti rispetto ai datatype più comuni ad esempio:

- Non esistono tipi oggetto predefiniti, ma bisogna definire i tipi di oggetto che si desidera.
- I tipi oggetto sono costituiti da *attributi* e *metodi*.
    - Gli attributi contengono i dati circa le caratteristiche dell'oggetto di interesse. Per esempio, il tipo oggetto *soldato* può avere gli attributi: `nome`, grado e `numero di serie`. Un attributo viene dichiarato come datatype che a sua volta può essere un altro tipo oggetto. Presi insieme, gli attributi dell'oggetto contengono i dati di quell'oggetto.
    - I metodi sono procedure o funzioni fornite per permettere agli di modificare i propri attributi. I metodi sono un elemento opzionale del tipo oggetto. Definiscono il comportamento degli oggetti di quel tipo e determinano che cosa quel tipo di oggetto può fare o meno.

- I tipi oggetto sono meno generici dei datatype, infatti, questa è una delle loro qualità principali: utilizzando tipi oggetto è possibile modellare la struttura di entità del mondo reale (come ad esempio i clienti e gli ordini d'acquisto). Ciò può rendere più facile e più intuitivo controllare i dati di queste entità. I tipi oggetto sono l'equivalente delle classi in C++ e Java.

Potete pensare ad un tipo oggetto come un modello e ad un oggetto come una cosa reale sviluppata secondo il modello stesso.

I tipi oggetto definiscono relazioni strutturali tra gli oggetti ed i loro attributi, invece di appiattire questa struttura in tabelle e colonne.

Con le espressioni CREATE TYPE e CREATE TYPE BODY è possibile creare un tipo oggetto, un tipo oggetto SQLJ, un tipo array variabile (varray), una tipo tabella annidata o un tipo oggetto incompleto. L'espressione CREATE TYPE BODY contiene il codice per i metodi del tipo.

Se viene creato un tipo oggetto nel quale vengono dichiarati solo attributi e non metodi, non c'è bisogno di specificare il type body. Se viene creato un tipo oggetto SQLJ, non è possibile specificare il type body, l'implementazione del tipo è specificata da una classe java.

7.2 TIPO OGGETTO INCOMPLETO

Un tipo oggetto incompleto è un tipo creato prima della definizione del tipo. Viene detto incompleto perché pur avendo un nome non presenta né attributi né metodi. Può essere utilizzato da altri tipi per definire tipi collegati. Comunque bisogna definire il tipo prima di utilizzarlo per creare tabelle, tabelle annidate, etc..

La sua nuova sintassi è la seguente:

```
CREATE [OR REPLACE] TYPE [schema .] type_name;
```

7.3 TIPO OGGETTO GENERALE

L'utilizzo di questo struttura serve per creare un tipo oggetto definito dall'utente. Le variabile associate all'oggetto sono chiamate attributi, mentre i sottoprogrammi che definiscono il comportamento dell'oggetto sono chiamatii metodi. La parola chiave AS OBJECT è necessaria quando si sta creando un oggetto di questo tipo. La sua nuova sintassi è la seguente:

```
CREATE [OR REPLACE] TYPE [schema .] type_name
[invoker_rights_clause]
{ { IS | AS } OBJECT
| UNDER [schema .] supertype
[ sqlj_object_type ]
}
[( { attribute datatype [sqlj_object_type_attr]
   | element_spec
   }
   [, attribute datatype [sqlj_object_type_attr]
   | element_spec
```

```
    ]...
)]
[ [ NOT ] FINAL ] [ [ NOT ] INSTANTIABLE ];
```

Ecco alcuni esempi per la creazione di oggetti generici:

```
CREATE TYPE data_typ AS OBJECT
   ( year NUMBER,
     MEMBER FUNCTION prod(invent NUMBER) RETURN NUMBER
   );

CREATE TYPE BODY data_typ IS
    MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
    BEGIN
        RETURN (year + invent);
    END;
END;

CREATE TYPE address_t AS OBJECT
  EXTERNAL NAME 'Examples.Address' LANGUAGE JAVA
  USING SQLData(
    street_attr varchar(250) EXTERNAL NAME 'street',
    city_attr varchar(50) EXTERNAL NAME 'city',
    state varchar(50) EXTERNAL NAME 'state',
    zip_code_attr number EXTERNAL NAME 'zipCode',
    STATIC FUNCTION recom_width RETURN NUMBER
      EXTERNAL VARIABLE NAME 'recommendedWidth',
    STATIC FUNCTION create_address RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION construct RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION create_address (street VARCHAR, city
            VARCHAR,state VARCHAR, zip NUMBER)
            RETURN address_t
      EXTERNAL NAME 'create (java.lang.String,
java.lang.String, java.lang.String, int) return
Examples.Address',
    STATIC FUNCTION construct (street VARCHAR, city VARCHAR,
state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME
```

```
        'create (java.lang.String, java.lang.String,
java.lang.String, int) return Examples.Address',
    MEMBER FUNCTION to_string RETURN VARCHAR
      EXTERNAL NAME 'tojava.lang.String() return
java.lang.String',
    MEMBER FUNCTION strip RETURN SELF AS RESULT
      EXTERNAL NAME 'removeLeadingBlanks () return
Examples.Address'
  ) NOT FINAL;
```

7.4 TIPO OGGETTO VARRAY

Con questo oggetto è possibile creare un set ordinato di elementi dello stesso tipo. Si devono specificare il nome e il valore massimo dell'array stesso (0 o più). Non si possono creare tipi varray di tipo LOB (Large OBject); questa restrizione vale anche per tipi XMLType poiché vengono memorizzati come CLOB.

La sua nuova sintassi è la seguente:

```
CREATE [OR REPLACE] TYPE [schema. .] type_name
{ IS | AS } { VARRAY | VARYING ARRAY } ( limit ) OF datatype;
```

Un esempio di oggetto varray è il seguente:

```
CREATE TYPE phone_list_typ_demo
  AS VARRAY(5) OF VARCHAR2(25);
```

7.5 TIPO OGGETTO TABELLA ANNIDATA

Utilizzando questo oggetto è possibile annidare una tabella all'interno di un campo di un'altra tabella. Non è possibile utilizzare all'interno di tabelle annidate tipi NCLOB, mentre sono consentiti i tipi CLOB e BLOB.

La sua nuova sintassi è la seguente:

```
CREATE [OR REPLACE] TYPE [schema. .] type_name
{ IS | AS } TABLE OF datatype;
```

Un esempio di tabella annidata è il seguente:

```
CREATE TYPE phone_list_typ AS OBJECT
```

```
        (location           VARCHAR2(10),
         prefix             NUMBER(5),
         phone              NUMBER(10)
        );

CREATE TYPE cust_address_typ AS OBJECT
        (street_address     VARCHAR2(40),
         postal_code        VARCHAR2(10),
         city               VARCHAR2(30),
         state_province     VARCHAR2(10),
         country_id         CHAR(2),
         phone              phone_list_typ
        );

CREATE TYPE cust_nt_address_typ
    AS TABLE OF cust_address_typ;
```

8 Tabelle Oggetto

8.1 TABELLE OGGETTO

Una tabella oggetto è un tipo speciale di tabella in cui ogni riga rappresenta un oggetto.
Per esempio, il codice seguenti genera un tipo oggetto `persona` e definisce una tabella oggetto costituito da oggetti persona:

```
CREATE TYPE person AS OBJECT (
  name         VARCHAR2(30),
  phone        VARCHAR2(20));


CREATE TABLE person_table OF person;
```

Si può interpretare questa tabella in due modi:

- Come tabella single-column in cui ogni riga è un oggetto persona.
- Come una tabella multi-column in cui ogni attributo del tipo persona, vale a dire nome e telefono, occupa una colonna.

Per esempio, si possono eseguire le seguenti istruzioni:

```
INSERT INTO person_table VALUES (
       'John Smith',
       '1-800-555-1212' );


SELECT VALUE(p) FROM person_table p
        WHERE p.name = 'John Smith';
```

La prima query considera la tabella come una tabella multi-column, mentre la seconda considera la tabella come una tabella single-column.

8.2 VINCOLI PER LE TABELLE OGGETTO

È possibile definire vincoli su una tabella oggetto così come per le altre tabelle e anche vincoli sugli attributi scalari di un oggetto della colonna.

I seguenti esempi illustrano le due possibilità. Questo esempio pone un vincolo chiave primaria sulla colonna di SSNO della tabella `PERSON_EXTENT`:

```
CREATE TYPE location AS OBJECT(
  building_no NUMBER,
  city        VARCHAR2(40));


CREATE TYPE person AS OBJECT (
  ssno      NUMBER,
  name      VARCHAR2(100),
  address   VARCHAR2(100),
  office    location);


CREATE TABLE person_extent OF person (
  ssno        PRIMARY KEY );
```

La tabella DEPARTMENT nell'esempio seguente ha una colonna di tipo LOCATION, definita nell'esempio precedente. L'esempio definisce i vincoli sugli attributi scalari degli oggetti LOCATION che compaiono nella colonna di DEPT_LOC della tabella seguente:

```
CREATE TABLE department (
  deptno      CHAR(5) PRIMARY KEY,
  dept_name   CHAR(20),
  dept_mgr    person,
  dept_loc    location,
  CONSTRAINT  dept_loc_cons1
      UNIQUE (dept_loc.building_no, dept_loc.city),
  CONSTRAINT  dept_loc_cons2
       CHECK (dept_loc.city IS NOT NULL));
```

8.3 INDICI PER LE TABELLE OGGETTO E LE TABELLE ANNIDATE

Si possono definire indici su una tabella oggetto come se fosse una normale tabella.

Si possono definire indici sugli attributi scalari degli oggetti colonna, come nel seguente esempio; qui, DEPT_ADDR è un oggetto colonna e CITY è un attributo scalare di DEPT_ADDR :

```
CREATE TABLE department (
  deptno      CHAR(5) PRIMARY KEY,
  dept_name   CHAR(20),
  dept_addr   location);


CREATE INDEX  i_dept_addr1
          ON  department (dept_addr.city);
```

Dove Oracle si aspetta un nome di colonna in una definizione di indice, si può anche specificare un attributo scalare di una colonna oggetto.

8.4 TRIGGER PER LE TABELLE OGGETTO

I trigger possono essere definiti su una tabella oggetto come per le altre tabelle. Il seguente esempio definisce un trigger sulla tabella PERSON_EXTENT definita precedentemente:

```
CREATE TABLE movement (
     ssno        NUMBER,
     old_office  location,
     new_office  location );

CREATE TRIGGER trig1
  BEFORE UPDATE
            OF  office
            ON  person_extent
   FOR EACH ROW
          WHEN  (new.office.city = 'REDWOOD SHORES')
   BEGIN
     IF :new.office.building_no = 600 THEN
      INSERT INTO movement (ssno, old_office, new_office)
       VALUES (:old.ssno, :old.office, :new.office);
     END IF;
```

```
END;
```

8.5 FUNZIONI TABELLA

Questo metodo usa una funzione tabella per restituire gli elementi della tabella in_list come righe. In primo luogo, si genera una tabella oggetto come valore di ritorno per la funzione tabella:

```
CREATE OR REPLACE TYPE t_in_list_tab AS TABLE OF VARCHAR2
(4000);
```

Dopo si crea la funzione tabella. Questa funzione accetta una stringa con valori separati da virgola che vengono poi traformati in righe di un'altra tabella che viene poi restituita come parametro di ritorno:

```
CREATE OR REPLACE FUNCTION in_list (p_in_list  IN  VARCHAR2)
  RETURN t_in_list_tab
AS
  l_tab   t_in_list_tab := t_in_list_tab();
  l_text  VARCHAR2(32767) := p_in_list || ',';
  l_idx   NUMBER;
BEGIN
  LOOP
    l_idx := INSTR(l_text, ',');
    EXIT WHEN NVL(l_idx, 0) = 0;
    l_tab.extend;
    l_tab(l_tab.last) := TRIM(SUBSTR(l_text, 1, l_idx - 1));
    l_text := SUBSTR(l_text, l_idx + 1);
  END LOOP;
  RETURN l_tab;
END;
```

La seguente query mostra la funzione in azione :

```
SELECT *
FROM   emp
WHERE      job   IN   (SELECT   *   FROM   TABLE(in_list('SALESMAN,
MANAGER')))
```

```
ORDER BY ename;
```

In output avrò :

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL
7499	ALLEN	SALESMAN	7698	20-FEB-1981 00:00:00	1600
7698	BLAKE	MANAGER	7839	01-MAY-1981 00:00:00	2850
7782	CLARK	MANAGER	7839	09-JUN-1981 00:00:00	2450
7566	JONES	MANAGER	7839	02-APR-1981 00:00:00	2975
7654	MARTIN	SALESMAN	7698	28-SEP-1981 00:00:00	1250

9 Codice SQL Dinamico Nativo

L'introduzione del codice SQL dinamico nativo ha semplificato l'uso del codice SQL dinamico. Questo rappresenta un notevole risparmio in termini di tempo, impegno di programmazione e complessità.

Le operazioni necessarie per utilizzare codice SQL dinamico sono molto semplici:

- definire l'istruzione SQL
- analizzare l'istruzione, caricare le variabili ed eseguire il programma in un unico passo utilizzando il comando EXECUTE IMMEDIATE.

Dunque è tutto molto facile. Ecco l'aspetto del programma:

```
CREATE OR REPLACE PROCEDURE inserisci_fatture
(in_data_vendita date,
 in_codice       number,
 in_quantita     number)
IS
 l_sql_stmt       VARCHAR2(200);
BEGIN
  l_sql_stmt:= 'INSERT INTO fatture
                VALUES (:data_vendita, :codice, :quantita)';
  EXECUTE IMMEDIATE l_sql_stmt
  USING in_data_vendita, in_codice, in_quantita;
END;
```

Questo metodo molto semplice per creare istruzioni SQL dinamiche in Oracle dovrebbe entrare a far parte del proprio bagaglio di conoscenze. Questo consentirà di utilizzare il linguaggio SQL per costruire programmi generici che possono trasformarsi in qualcosa di molto più specifico.

10 Incapsulamento e Overloading

L'incapsulamento è un meccanismo mediante il quale i dati e il codice che manipola i dati vengono inseriti in un unico oggetto.

L'overloading è la capacità di creare più versione di una procedura che vengono richiamate a seconda dei parametri inviati alla procedura stessa.

Un esempio di overloading è il seguente:

```
DECLARE
  TYPE DateTabTyp IS TABLE OF DATE INDEX BY BINARY_INTEGER;
  TYPE RealTabTyp IS TABLE OF REAL INDEX BY BINARY_INTEGER;
  PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
    BEGIN
       ...
    END;
  PROCEDURE initialize (tab OUT RealTabTyp, n INTEGER) IS
    BEGIN
       ...
    END;
BEGIN
 ...
END;
```

Si può notare come nelle due procedure si inizializza la stessa variabile tab assume una volta il tipo DateTabTyp e l'altra volta il tipo RealTabTyp.

11 Transazioni Autonome

Grazie a questa funzionalità è possibile rendere indipendente il programma in modo che quando si esegue il commit o il rollback dei dati di un programma questo non influenzi gli altri dati che possono essere stati creati all'esterno del programma stesso.

In Oracle 9i è possibile definire delle librerie per i nostri programmi; queste librerie sono chiamate pragma e sono direttive per il compilatore che gli chiedono di comportarsi in un determinato modo. Ecco un esempio d'uso di una pragma chiamata AUTONOMOUS_TRANSACTION utile al nostro scopo. Innanzitutto si creerà una procedura che inserisce dati nella tabella impiegato:

```
CREATE PROCEDURE insert_impiegato
(in_cod  IN   number,
 in_nome IN   varchar2,
 in_sal  IN   number,
 in_prov IN   varchar2)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO impiegato VALUES (in_cod,in_nome,in_sal,in_prov);
  COMMIT;
END;
```

```
Una volta creata la procedura ecco come può essere utilizzata:
SQL>BEGIN
  2 INSERT INTO impiegato
  3 VALUES (100, 'Harrison Astroff', 10000, 'Ontario');
  4 insert_impiegato (101, 'Jillian Abramson',
  5 10000, 'California');
  6 ROLLBACK;
  7 END;
  8 /
PL/SQL procedure successfully completed.
```

In questo breve programma è stato utilizzato un comando insert standard che consente di aggiungere un dipendente (Harrison Astroff). Nel comando successivo si richiama la transazione autonoma insert_impiegato. Si ricordi che questo programma esegue una commit. Prima delle transazioni autonome

entrambi i record devono avere eseguito il commit nel database. In questo caso si vuole annullare il primo inserimento. Ecco dunque il contenuto del database:

```
SQL>RUN
  2 SELECT * FROM impiegato
  3 WHERE cod_imp >= 100

COD_IMP          IMP_NOME SALARIO   PROVINCIA
------- ----------------- ------- -----------
    101  Jillian Abramson   10000  California
```

Come si può vedere solo i dati che sono passati attraverso la transazione autonoma sono stati riportati sul database; gli altri record sono stati invece annullati dal programma.

12 Query di testo dal Database

12.1 QUERY CONTEXT

A partire da ORACLE 7.3 è possible utilizzare l'opzione ConText per effettuare ricerche di testo come parte di query sul database. Le ricerche di testo sono implementate in ORACLE tramite processi server che operano in parallelo ai normali processi in background.

Se il database è impostato in modo da utilizzare l'opzione ConText è possibile eseguire la seguente select:

```
select Nome
  from Tab1
 where CONTAINS (Coll, 'ingegneria')>0;
```

Se nella colonna, Coll, viene individuata la parola 'ingegneria' viene restituito un *punteggio* maggiore di zero e il valore Nome corrispondente.

12.2 ESPRESSIONI DI QUERY CONTEXT DISPONIBILI

Gli operatori all'interno della funzione CONTAINS consentono l'esecuzione delle ricerche di testo elencate di seguito:

- Esatta coincidenza di una parola o frase.
- Esatta coincidenza di più parole, utilizzando la logica booleana per combinare le ricerche.
- Ricerche basate sulla prossimità delle parole alle altre parole nel testo.
- Ricerche di parole che condividano le stessa etimologia.
- Coincidenza "fuzzy" delle parole.
- Ricerche di parole che abbiano lo stesso suono di altre.

13 Concetti orientati agli oggetti avanzati in oracle

Un *oggetto annidato* è completamente contenuto all'interno di un altro oggetto. Anche se I dati di una tabella annidata vengono memorizzati separatamente dalla tabella principale, è possibile accedervi solo attraverso la tabella principale stessa.

Per avvalersi delle caratteristiche orientate agli oggetti un database deve supportare anche *oggetti riga*. Gli oggetti riga non sono oggetti annidati, ma *oggetti referenziali*, accessibili per mezzo di riferimenti da altri oggetti.

13.1 TABELLE OGGETTO E OID

In una tabella oggetto, ciascuna riga è un oggetto riga. Una tabella oggetto differisce in molti modi da una normale tabella relazionale. Ciascuna riga all'interno della tabella oggetto ha un valore di identificazione oggetto (OID, Object Identifier) assegnato da ORACLE quando viene creata. Alle righe di una tabella oggetto, possono fare riferimento agli oggetti all'interno del database.

Per creare una tabella oggetto:

```
create or replace type ANIMALE_TY as object
(Famiglia  VARCHAR2 (25),
Nome            VARCHAR2 (25),
DataNascita     DATE);
```

13.2 INSERIMENTO DI RIGHE NELLE TABELLE OGGETTO

Per inserire righe nella tabella oggetto ANIMALE:

```
insert into ANIMALE values
     (ANIMALE_TY ('MULO','FRANCES',
                  TO_DATE('01-APR-1997','DD-MON YYYY')));
```

13.3 UPDATE E DELETE DA TABELLE OGGETTO

Nell'esempio seguente viene effettuato l'update delle righe di ANIMALE.

```
update ANIMALE
   set DataNascita = TO_DATE ('01-MAY-1997','DD-MON-YYYY')
 where Nome = 'LYLE'
```

Durante un delete è possibile utilizzare:

```
delete from ANIMALE
 where Nome = 'LYLE';
```

13.4 VISTE OGGETTO CON REF

Le viste oggetto consentono di sovrapporre strutture orientate agli oggetti a tabelle relazionali esistenti. È possibile creare tipi di dati e utilizzarli all'interno della vista oggetto di una tabella esistente.

Viene creata una tabella CLIENTE, con la colonna Cliente_ID designata come chiave primaria.

```
Create table CLIENTE
(Cliente_ID NUMBER primary Key,
Nome        VARCHAR2 (25),
Via         VARCHAR2 (50),
Citta       VARCHAR2 (25),
Prov        CHAR (2),
Cap         NUMBER);
```

In seguito vengono creati due tipi di dati astratti.

```
create or replace type INDIRIZZO_TY as object
(Via         VARCHAR2 (50),
Citta        VARCHAR2 (25),
Prov         VARCHAR2 (2),
Cap          NUMBER);
create or replace type PERSONA_TY as object
Nome         VARCHAR2 (25)
Indirizzo INDIRIZZO_TY;
```

E' possibile creare una vista oggetto che specifichi i tipi di dati astratti che si applicano alla tabella CLIENTE.

```
create view CLIENTE_OV (Client_ID, Persona) as
Select Cliente_ID,
       PERSONA_TY (Nome,
       INDIRIZZO_TY(Via, Citta, Prov, Cap))
    From CLIENTE
```