

BASI DI DATI

Prof. Vincenzo Moscato – A.A. 2025/26

Luca Maria Incarnato

INDICE DEGLI ARGOMENTI

SQL E IL MODELLO RELAZIONALE

1. SISTEMI INFORMATIVI E BASI DI DATI (p. 3)
2. IL MODELLO RELAZIONALE (p. 10)
3. OPERAZIONI DI MODIFICA E RECUPERO DELLE INFORMAZIONI (p. 17)
4. LA NORMALIZZAZIONE (p. 25)
5. IL MODELLO ER (p. 29)
6. IL MODELLO ER AVANZATO (p. 36)

PROGETTAZIONE FISICA E LIVELLO APPLICATIVO

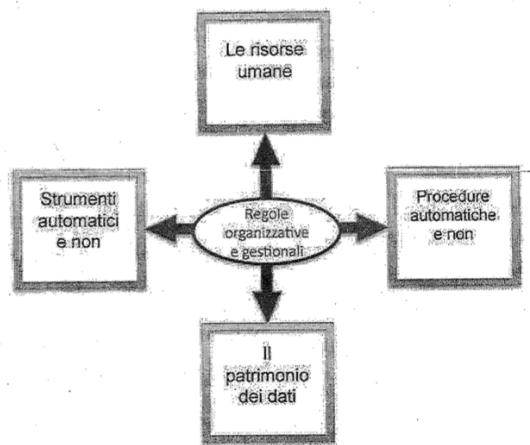
7. ORACLE DBMS (p. 43)
8. PL/SQL (p. 52)
9. I TRIGGER E LE BASI DI DATI ATTIVE (p. 63)
10. BASI DI DATI DIREZIONALI (p. 69)
11. TECNOLOGIE PER I SISTEMI DI BASI DI DATI (p. 77)
12. BASI DI DATI DISTRIBUITE (p. 98)

SQL E IL MODELLO RELAZIONALE

SISTEMI INFORMATIVI E BASI DI DATI

Dal punto di vista pratico e lavorativo, **l'informazione** (e quindi i dati da cui tali informazioni sono estratte) **rappresenta uno dei beni più preziosi**, a partire dalla quale vengono **prese decisioni**, **organizzati i flussi di lavoro e pianificata e/o controllata l'operatività**; quindi, si può facilmente intuire come **sistemi per la gestione delle informazioni siano più che necessari per l'operatività di un'azienda o di un servizio**.

L'informazione è una **risorsa particolare** su cui operano tutte le organizzazioni, **non viene mai consumata** e, anzi, **tende ad accrescere nel tempo con l'aumento della conoscenza** (i sistemi per la gestione delle informazioni sono sistemi che gestiscono una mole di dati strettamente crescente). Un “**processo aziendale**” può identificarsi con la **sequenza di attività svolte all'interno di un'azienda opportunamente correlate alla realizzazione di un risultato definito e misurabile** e tale da coinvolgere un insieme di risorse materiali, informative e organizzative. L'insieme delle **informazioni gestite dai processi aziendali** costituisce l'ossatura del cosiddetto **sistema informativo**, un sistema costituito da più elementi interagenti ai fini di soddisfare l'interesse dell'azienda (appena illustrato).

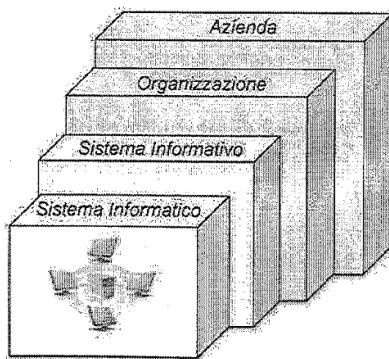


Quando in forma esplicita, **un sistema informativo è composto dalle seguenti componenti**:

- Un **patrimonio di dati**, la materia prima con cui si producono informazioni;
- Un **insieme di procedure per l'acquisizione e la produzione di informazioni**;
- Un **insieme di procedure per la gestione dei dati**;
- Un **insieme di persone** (risorse umane) che **sovraintendono a tali procedure**;
- Un **insieme di mezzi** e di strumenti necessari al **trattamento, trasferimento e archiviazione di dati e di informazioni**;
- Un **insieme di regole organizzative e gestionali** che **caratterizzano il sistema** e ne **determinano il comportamento**.

Le **interazioni fra le varie componenti** di base di un sistema informativo **generano un insieme di flussi informativi** che attraversano i processi di una data organizzazione, condizionandone l'efficienza e l'efficacia. In un tale scenario, **si è soliti operare una netta distinzione tra sistema informativo e sistema informatico**: il primo esiste sicuramente da molto più tempo del secondo e spesso **include molti aspetti che un sistema informatico non può ancora implementare**.

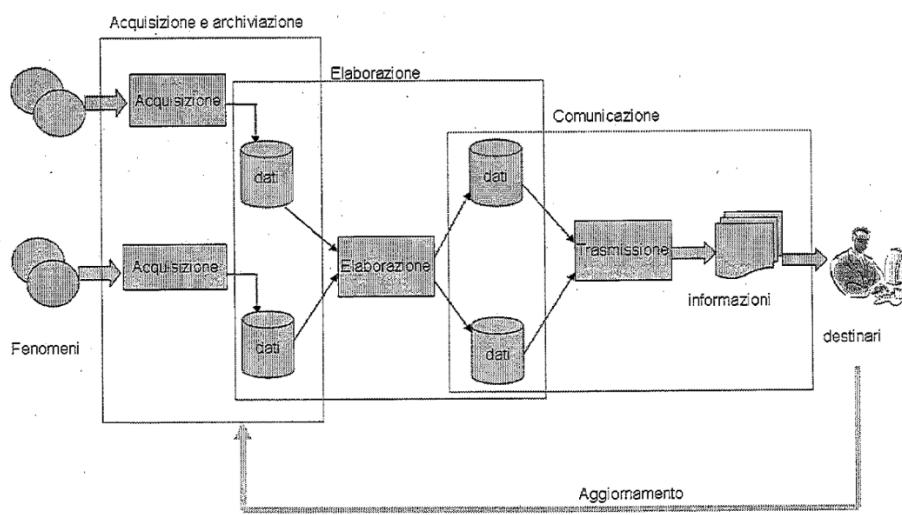
Pertanto, è utile dare una definizione formale di sistema informativo, definito come l'insieme delle componenti di un'organizzazione dedita all'acquisizione, elaborazione, memorizzazione, recupero, condivisione e trasmissione dell'informazione. In altre parole, un sistema informativo definisce le modalità con cui le informazioni sono trattate lungo tutto il loro ciclo vitale; di contro, un sistema informatico ha a che fare con gli aspetti tecnologici dei sistemi di elaborazione dell'informazione e, formalmente, si definisce come la tecnologia di supporto del sistema informativo (in termini di supporto hardware, software e reti di comunicazione).



In conclusione, un sistema informativo deve consentire:

- L'acquisizione delle informazioni ed il loro aggiornamento;
- L'archiviazione delle informazioni tale da garantirne un reperimento efficace;
- L'elaborazione delle informazioni per produrre aggregazioni e correlazioni utili per produrre statistiche e sintesi;
- La comunicazione delle informazioni alle persone che, a vari livelli, devono poterle utilizzare.

Di seguito è proposto lo schema generale di produzione e aggiornamento delle informazioni:



Abitualmente, in un'azienda il sistema informativo è solo in parte automatizzato; infatti, anche sulla base di quanto detto precedentemente in merito ai sistemi informatici, permangono, per difficoltà tecniche o per convenienza economica, alcune aree in cui le informazioni sono prodotte senza l'uso di tecnologie informatiche.

Alcuni esempi di sistemi informativi sono:

- **Sistemi di supporto operativo**, gestiscono scambi informativi all'interno di processi operativi tra diverse aziende (**B2B**, Business to Business), tra processi della stessa azienda o tra un utente e un'azienda (**B2C**, Business to Customer);
- **Sistemi di monitoraggio e controllo;**
- **Sistemi informativi nella Pubblica Amministrazione;**
- **Sistemi informativi ospedalieri;**
- **Sistemi informativi per i trasporti.**

Esiste una sostanziale **differenza tra informazioni e dati**; in primis, **il dato è una rappresentazione dell'informazione utile alla sua memorizzazione e gestione** mentre, più in sostanza, **il concetto di informazione è legato a quello di scelta**, un'informazione cerca di **eliminare un'incertezza presente** identificando, all'interno di un dominio di valori, un determinato elemento associandogli un preciso significato. **L'informazione viene identificata**, in informatica, **da una tripla: (valore, tipo, attributo)**, dove valore è un particolare elemento scelto, tipo è l'insieme degli elementi entro cui si compie una scelta e attributo l'identificativo che attribuisce un significato al dato. Semplificando, **la differenza tra dato e informazione è nel contesto: il primo non lo possiede, il secondo sì**.

Attributo	Tipo	Valore
Cliente	Stringa di Caratteri	Paolo Rossi
Data di Nascita	Data	08/12/1964
Titolo Libro	Stringa di Caratteri	Così parlò Zarathustra
x	Reale	1.0

Poiché dipende fortemente dal contesto, **è possibile che un'informazione si comporti come dato per un'altra informazione**, ottenuta per combinazione di diverse altre informazioni e, quindi, "complessa". Ad esempio:



Una **base di dati (BD)** è l'insieme di informazioni associato a collezioni di dati tra di loro correlati e dotati di un'opportuna descrizione. In altre parole, una base di dati è una raccolta di dati che possono essere usati simultaneamente da utenti differenti all'interno di una stessa organizzazione; l'idea di condividere tale collezione permette di lavorare su uno stato consistente dei dati, evitando ridondanze. Da ciò, **è possibile dire che una base di dati è**:

- **Di grandi dimensioni;**
- **Condivisa** da più applicazioni/utenti;

- **Persistente** (i dati e le relative informazioni devono avere un tempo di vita nettamente maggiore del tempo di vita più lungo delle procedure pensate per la loro gestione).

In una base di dati, **i dati contenuti devono essere sempre aggiornati per riflettere i cambiamenti che si verificano continuamente nel sistema informativo dell'organizzazione**, ma non sono solo i dati che vanno mantenuti aggiornati, anche le loro descrizioni, in modo da garantire **l'aggiornamento anche delle relative informazioni**; si parla di **collezione di dati autodescrittiva** l'insieme di dati che descrivono i dati (detti anche **metadati** o **catalogo**). L'insieme dei dati e del relativo catalogo viene oggi gestito in modo integrato da uno strato software che prende il nome di **Data Base Management System (DBMS)** ed è l'ultimo tassello che serve per definire con chiarezza la struttura di un'organizzazione e del proprio sistema di gestione di una base di dati:

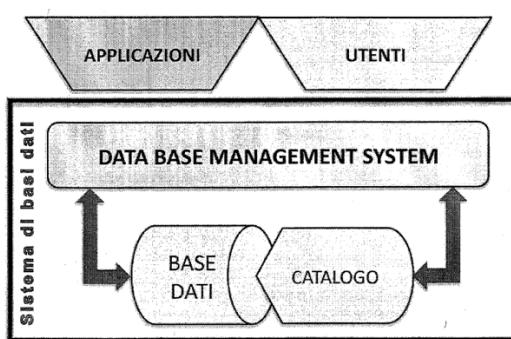


Formalmente, **un sistema di gestione di una base di dati (DBMS) è un insieme di programmi che permette di definire, manipolare e controllare una base di dati**. Un DBMS, in altre parole, permette di:

- **Definire le informazioni;**
- **Manipolare i dati;**
- **Accedere in modo controllato alla base di dati** (ovvero condividere le informazioni tra più utenti, garantendo allo stesso tempo la protezione dei dati e la manutenzione evolutiva del sistema).

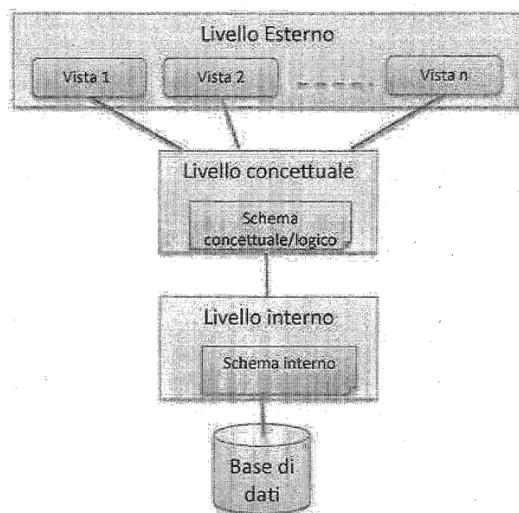
Un DBMS facilita gli utenti nell'utilizzo della propria banca dati; prima dell'avvento dei DBMS, **un'applicazione doveva gestire tutto l'interfacciamento con l'archivio** (mediante i meccanismi di gestione dei file forniti dai vari linguaggi di programmazione), **la concorrenza degli accessi e l'allineamento dei dati in caso di archivio condiviso da più utenti.**

Per **sistema di basi di dati** si intende l'insieme composto da una base di dati e da un DBMS:



DBTG (Data Base Task Group) fu una delle prime proposte di architettura generale per i sistemi di basi di dati e fu introdotta nel 1971; tuttavia, non passò molto tempo prima che l'**ANSI** (American National Standard Institute) propose il proprio modello analogo, detta architettura **ANSI-SPARC** (da SPARC, la commissione che l'ha prodotto), il cui scopo fondamentale era quello di garantire uno stretto isolamento tra i dati e i programmi/applicazioni che su di esse operano. ANSI-SPARC non divenne mai uno standard ma la sua comprensione è una buona base di partenza per capire il funzionamento di un DBMS. In accordo con tale modello, i dati vengono descritti secondo tre livelli:

- **Livello interno**, specifica i dettagli della memorizzazione fisica dei dati e come le relative informazioni sono memorizzate su dispositivi fisici di memorizzazione di massa (descrive il modo in cui il DBMS e il Sistema Operativo gestiscono i dati);
- **Livello concettuale**, descrive dal punto di vista logico come i dati sono organizzati in informazioni e come queste ultime sono correlate tra loro (può essere visto come un insieme di record con le loro descrizioni);
- **Livello esterno**, personalizza l'accesso alle informazioni relativamente ad un singolo utente o ad una classe di utenti, sulla base della struttura logica del livello concettuale (descrive il modo in cui gli utenti e le applicazioni “vedono” i dati).



La **descrizione complessiva di una base di dati** è anche detta **schema di dati**. Si noti che mentre **una base di dati ha sempre un singolo schema concettuale ed un singolo schema fisico**, essa può possedere molteplici **schemi esterni** che costituiscono, in effetti, viste differenti delle informazioni. Il vantaggio fondamentale di uno schema a livelli come quello illustrato è la cosiddetta **data-application independence**; infatti, **considerando programmi applicativi che interagiscono con la base di dati attraverso un DBMS, è possibile dimostrare che**:

- Attraverso il meccanismo delle “viste”, le **applicazioni possono essere rese ignare dello schema logico della base di dati**, visto solo dal DBMS;
- Attraverso lo schema logico del livello concettuale, si è garantiti dalla **differente implementazione fisica dei dati** (un utente può operare sulla base di dati anche se nel tempo viene cambiato il modo in cui i dati sottostanti sono rappresentati).

Queste due proprietà fanno risalire alla memoria le proprietà di **information hiding** della Programmazione Orientata agli Oggetti.

Storicamente, i modelli dei dati più diffusi nel mondo delle basi di dati sono i seguenti:

- **Modello gerarchico**, per il quale i dati sono organizzati in gerarchie attraverso l'uso di strutture dati ad albero;
- **Modello reticolare**, per il quale i dati sono organizzati in un reticolo attraverso l'uso di strutture dati a grafo;
- **Modello relazionale**, per il quale i dati sono organizzati in relazioni, ovvero insiemi di record aventi la stessa struttura logica;
- **Modello object oriented**, per il quale i dati sono organizzati sotto forma di “oggetti”;
- **Modelli object relational**, la struttura rimane relazionale ma la tabella è strutturalmente generalizzata al fine di contenere oggetti oltre che valori elementari;
- **Modelli NoSQL**, alternative al modello relazionale.

Il DBTG ha definito i seguenti compiti principali di un DBMS:

- **Definire** come i dati sono organizzati in informazioni attraverso un opportuno linguaggio di definizione dei dati, detto DDL;
- **Manipolare e gestire** i dati attraverso un opportuno linguaggio di gestione, detto DML;

Le istruzioni DML si dividono in **istruzioni interrogative** (query) e **istruzioni che modificano lo stato della base di dati**, come:

- **INSERT**, permette di inserire nuove informazioni all'interno di una base di dati;
- **UPDATE**, permette di modificare informazioni persistenti all'interno di una base di dati;
- **DELETE**, permette di cancellare informazioni da una base di dati.

Un DBMS esegue dei particolari programmi, detti **transazioni**, che costituiscono **unità atomiche di modifiche fatte allo stato di una base di dati; una transazione o termina in uno stato finale (commit) o porta il sistema ad uno stato precedente all'esecuzione (abort)**. I sistemi di basi di dati basati su transazione sono detti anche **On Line Transaction Processing (OLTP)**.

Una transazione può contenere anche solo interrogazioni, pur continuando ad assicurare l'**atomicità**, dal momento in cui **un'interrogazione non modifica lo stato di una base di dati**. Le transazioni devono possedere necessariamente delle proprietà fondamentali, riassunte dall'acronimo ACID:

- **Atomicità**, una transazione è atomica se è eseguita nella sua interezza o se non è eseguita proprio;
- **Consistenza**, una transazione è una trasformazione di stato consistente della base di dati in un altro stato consistente (si tratta di verificare che tutti i vincoli definiti sulla base di dati siano soddisfatti);
- **Isolamento**, le transazioni devono essere eseguite in modo indipendente l'una dalle altre (gli effetti parziali di transazioni incomplete non devono essere visibili alle altre transazioni);
- **Durability** (o Persistenza), gli effetti di una transazione che è terminata con un commit devono essere registrati in maniera permanente nella base di dati e non devono essere persi per alcun motivo.

Si noti che **per preservare le proprietà di atomicità e persistenza**, qualunque cosa accada, sia a livello di crash fisico che di sistema, deve poter essere garantito che solo gli effetti di ogni transazione andata a buon fine siano permanentemente memorizzati nella base di dati. Per quanto riguarda le query, invece, un utente che cerca informazioni in una base di dati si aspetta di ricevere dati coerenti e consistenti, ovvero efficaci, nel minor tempo possibile, ovvero

efficiente; un DBMS, pertanto, deve poter gestire l'esecuzione efficiente di una query attraverso una ricerca di informazioni su una collezione di dati per lo più sempre molto grande.

Riassumendo, le funzioni fondamentali che un DMBS deve possedere, come descritto da Codd (ideatore del modello relazionale), sono:

- Permettere agli utenti la possibilità di memorizzare, modificare e, ovviamente, recuperare i dati presenti nella base di dati;
- Mettere a disposizione un catalogo di sistema, accessibile all'utente, che contenga la descrizione dei dati;
- Fornire un meccanismo che assicuri che tutte le modifiche dei dati relative ad una transazione siano fatte in modo permanente o che, in caso contrario, non ne venga fatta nessuna;
- Fornire un meccanismo che assicuri che la base di dati sia correttamente modificata quando utenti multipli effettuano modifiche concorrenti sulla base di dati stessa (controllo della concorrenza);
- Disporre di meccanismi atti a recuperare i dati in caso di danni (recovery);
- Fornire meccanismi che permettano l'accesso ai dati ai soli utenti autorizzati;
- Potersi facilmente integrare con software applicativi;
- Fornire dei meccanismi per assicurare che i dati presenti in una base di dati soddisfino certe regole, dette regole di integrità.

I principali componenti di cui possono essere dotati i DBMS sono:

- Gestore degli accessi, modulo che effettua il controllo degli accessi alla base di dati garantendo che solo utenti ed applicazioni autorizzati abbiano accesso alle informazioni della base di dati;
- Gestore delle query, modulo che si occupa della gestione delle richieste utente in termini di operazioni DDL e DML;
- Gestore della memoria, componente che si occupa di definire le strategie di accesso alle informazioni presenti in memoria di massa e del relativo trasferimento di queste ultime in memoria centrale e viceversa;
- Gestore dei file, modulo che si occupa di gestire i file in memoria di massa contenenti la base di dati;
- Gestore della concorrenza, modulo che si occupa di gestire l'accesso di transazioni concorrenti a risorse condivise della base di dati;
- Gestore di integrità, modulo che si occupa di verificare il rispetto delle regole di integrità all'interno della base di dati;
- Gestore dell'affidabilità, modulo che si occupa del salvataggio delle operazioni sulla base di dati nei file di log e dell'avvio delle procedure di ripristino della base di dati a valle di malfunzionamenti.

Un importante vantaggio nell'uso di DBMS consiste nella riduzione della Ridondanza dei dati, essendo cercata l'eliminazione di duplicazioni inutili o pericolose di dati su vari file; allo stesso tempo, è garantita la consistenza, se un'informazione è presente una sola volta in una base di dati una sua modifica rende immediatamente disponibile a tutti il nuovo valore. Pertanto, è possibile dire che un DBMS offre elevata Affidabilità e Sicurezza.

I DBMS relazionali mettono a disposizione degli utenti e dei programmati un linguaggio per accedere, gestire e manipolare le informazioni presenti in una base di dati, noto come SQL (Structured Query Language). Si tratta di un linguaggio dichiarativo, nel senso che permette all'utente di specificare quale deve essere il risultato dell'azione elaborativa (**cosa si vuole**) lasciando

al sistema il compito di eseguire ed ottimizzare le varie operazioni. Sotto questo punto di vista, SQL si distingue di gran lunga dai linguaggi di programmazione procedurali, in cui un utente deve specificare come una certa azione deve essere svolta e l'ordine in cui le istruzioni necessarie devono essere eseguite. Un vantaggio notevole nell'utilizzo di SQL consiste nella **possibilità di interagire con una base di dati qualsiasi sia il DBMS che la gestisca**, svincolando le applicazioni che operano su una base di dati dal particolare DBMS commerciale o open-source sottostante.

IL MODELLO RELAZIONALE

Il modello relazionale si basa interamente sul concetto di **relazione matematica**, come definita nella teoria degli insiemi; in particolare, **siano considerati n insiemi di valori non necessariamente distinti**, detti **domini D_1, \dots, D_n** , allora si definisce **relazione r un sottoinsieme, anche vuoto, del prodotto cartesiano degli n domini**:

$$r \subseteq D_1 \times \dots \times D_n$$

Nel modello relazionale, è fatta l'ipotesi ulteriore di **atomicità dei domini**. Dalla definizione appena enunciata, è possibile concludere che una relazione sia formata come un **insieme di n -uple** (o ennupla) **ordinate di valori**:

$$t = (v_1, \dots, v_n) : v_1 \in D_1 \wedge \dots \wedge v_n \in D_n$$

Nel gergo tecnico, però, il nome ennupla è sostituito con il nome tupla, dall'inglese tuple. Una **relazione**, dunque, è **definita da un numero n di domini elementari**, detto anche **grado della relazione**, e **dal numero di tuple che la compongono**, detto anche **cardinalità della relazione**. Secondo il **modello relazionale di Codd**, devono valere le seguenti proprietà:

1. **Proprietà di ordinamento esterno**, per la quale non è definito alcun ordinamento tra le tuple di una relazione (ovvero, in termini matematici, non è definito sull'insieme una funzione di ordinamento);
2. **Proprietà di distinzione**, per la quale ogni tupla è distinta da tutte le altre (deriva dalle proprietà dell'insiemistica matematica);
3. **Proprietà di corrispondenza posizionale**, per la quale esiste una corrispondenza posizionale tra i valori interni ad una tupla ed i relativi domini.

La proprietà di corrispondenza posizionale viene a decadere nel momento in cui ad ogni valore di un dominio è associato un attributo per identificare e qualificare il ruolo del dominio stesso; il tutto avviene tramite **una funzione del tipo**:

$$\text{dom}: A \rightarrow D$$

Con **A insieme di tutti gli attributi** e **D insieme di domini di una relazione**. Con quanto appena detto, è necessario **aggiornare la definizione di tupla**:

$$t = \{\langle v_1, A_1 \rangle, \dots, \langle v_n, A_n \rangle\}$$

Dal risultato appena ottenuto, è possibile **rimuovere la proprietà di corrispondenza posizionale e definire la proprietà di ordinamento interno**, per la quale **non esiste alcun ordinamento all'interno di una tupla**; in altri termini, con l'introduzione degli attributi, **la posizione di un valore all'interno di una tupla perde la sua importanza**:

$$t = \{\langle v_1, A_1 \rangle, \dots, \langle v_n, A_n \rangle\} = \{\langle v_n, A_n \rangle, \dots, \langle v_1, A_1 \rangle\}$$

Seguendo lo stesso filone argomentativo intrapreso finora, è possibile, grazie all'introduzione degli attributi e alla rimozione dell'ordinamento interno, rappresentare una relazione anche attraverso tabelle in cui le righe sono tuple (o record) e le colonne i valori attribuiti ad un dato attributo lungo tutte le tuple. Ovviamente, nonostante sia stata sostituita la terza proprietà del modello relazionale di Codd, una tabella per essere una relazione deve necessariamente rispettare anche le altre due proprietà precedentemente enunciate; ad esempio, una relazione in forma tabellare è la seguente:

Codice	Nome	Cognome	Nazionalità
001	Pedro	Almodovar	Spagna
005	Federico	Fellini	Italia

L'operazione appena eseguita permette di dare ad una struttura matematica la forma di una struttura dati più simile a quelle a cui si è abituati nell'ambito informatico. Con l'introduzione degli attributi, si introduce una nuova definizione di relazione, basata sul concetto di schema di relazione (definito come uno scheletro per la relazione); dato un insieme di nomi di attributi, $X = \{A_1, \dots, A_n\}$, si definisce schema di relazione un nome R , seguito dall'insieme di nomi di attributi X :

$$R(A_1, \dots, A_n) = R(X)$$

Ritornando all'esempio precedente, lo schema di relazione associato alla tabella è AUTORE (Codice, Nome, Cognome, Nazionalità); inoltre, si definisce relazione r su uno schema $R(X)$ un'istanza di $R(X)$.



In altre parole, per costruire una relazione su uno schema di relazione è sufficiente associare a quest'ultimo un insieme di tuple; tuttavia, mentre lo schema non varia nel tempo, l'insieme delle tuple subisce modifiche in funzione della realtà che la tabella rappresenta. Per questo motivo, si dice che lo schema è la componente intensionale della base di dati, mentre una sua istanza è la componente estensionale. Il rapporto tra uno schema di relazione e la relazione sullo stesso schema è lo stesso che sussiste tra una classe e un oggetto istanza di quella classe.

Sia t_i una delle n tuple definite su un insieme di attributi X e sia $A \in X$ un attributo del relativo schema di relazione. Con la notazione $t_i[A]$ si indica il valore della i -esima tupla rispetto all'attributo A ; è possibile, inoltre, estendere l'operazione, con abuso di notazione, anche ad un sottoinsieme $Y \subseteq X$ di attributi, indicando con $t_i[Y]$ i valori della i -esima tupla rispetto agli attributi contenuti in Y . Ad esempio, per la relazione:

NomeStudente	NomeEsame	Voto	Lode
Pippo	TSI	30	SI
Paolo	TM	28	NO
Marta	EI	18	NO
Maria	A1	30	NO

Si ha $t_2[\text{NomeStudente}] = \text{Paolo}$ e $t_2[\text{NomeStudente}, \text{Voto}] = (\text{Paolo}, 28)$. Si noti che, per questo secondo tipo di operazione, il risultato è una tupla, non necessariamente di una relazione.

Nel modello relazionale, un ruolo molto importante è assunto dall'assenza di informazione. Un modo per codificare un valore per rappresentare l'assenza di valore consiste nell'estendere i domini alla possibilità di assumere un valore particolare, **NULL**, che assolve proprio al compito specificato; sotto queste ipotesi, è possibile aggiornare la definizione di dominio:

$$D_i \equiv D_i \cup \{\text{NULL}\}$$

Tendenzialmente l'utilizzo di valori **NULL** induce ad una maggiore probabilità di incontrare errori (soprattutto quando si vanno ad effettuare analisi statistiche); pertanto, ci si chiede quali possano essere i motivi per cui sia necessario specificare un valore per codificare l'assenza di informazione e la risposta risiede in diverse casistiche reali; ad esempio:

- Il dato esiste ma non è stato fornito (**NULL come valore sconosciuto**);
- Il dato non esiste, in quanto non è applicabile alla specifica tupla (**NULL come valore inesistente**);
- Il dato non c'è ma non si sa se è sconosciuto oppure inesistente (**NULL come valore senza informazione**).

Si consideri che l'impiego del valore **NULL** è giustificato quando si va a parlare di gestione di basi di dati ma non ha senso dal punto di vista matematico.

Per vincolo di integrità (Integrity Constraint, IC) si intende una regola (di schema) che ogni istanza di uno schema di relazione deve rispettare affinché i propri dati siano corrispondenti al modello della realtà che esso cattura.

Si definisce schema di base di dati B_D un nome associato alla base di dati (B_D) seguito dall'insieme di schemi di relazione ($R_1(X_1), \dots, R_n(X_n)$) più un insieme IC di regole e costituisce un metodo di controllo di ciò che viene inserito all'interno di una base di dati, mentre per base di dati relazionale si intende un'istanza di uno schema di base di dati che soddisfa le regole contenute in IC. Se un'istanza soddisfa tutti i vincoli di integrità specificati nel suo schema, si parla di istanza legale della base di dati; il controllo del rispetto dei vincoli imposti ad uno schema di base di dati è uno dei compiti fondamentali a cui un DBMS deve assolvere.

Esistono sostanzialmente due tipologie di vincoli, entrambi sono definiti sullo schema di basi di dati ma applicati alle istanze (riguardano la parte intenzionale):

- Vincoli intra-relazionali

Sono vincoli espressi attraverso condizioni logiche che devono essere soddisfatte all'interno di una singola relazione; a loro volta, i vincoli intra-relazionali possono essere intesi come vincoli di dominio o vincoli di tupla: i primi interessano i singoli valori dei campi di una tupla, i secondi si applicano a più campi della tupla o all'intera tupla. Quindi, un vincolo di dominio è una regola

che deve essere soddisfatta relativamente ai valori di un fissato attributo di relazione (ad esempio, una LODE può essere una stringa “SI” o “NO” ma non “FORZA NAPOLI”); di contro, **un vincolo di tupla è una regola che deve essere soddisfatta relativamente ai valori di un insieme di domini** (ad esempio, una LODE può essere assegnata a “SI” solo quando il VOTO è 30).

Il vincolo di integrità intra-relazionale più importante è il **vincolo di chiave**; tuttavia, per chiarirne la definizione, è utile **definire dapprima la superchiave**. Sia dato uno schema di relazione $R(X)$ e sia $SK \subseteq X$ un sottoinsieme di attributi di X , si può dire che SK è una superchiave di una relazione $r \in R(X)$ se $\forall t_i, t_j \in r : i \neq j \Rightarrow t_i[SK] \neq t_j[SK]$. Detto ciò, un sottoinsieme $K \subseteq X$ di attributi di X è chiave per r se è una superchiave minimale di r (ovvero se, togliendo qualsiasi attributo da K , K non è più superchiave).

È possibile notare che, in una relazione, esiste sempre almeno una chiave e che è possibile individuare chiavi differenti. Tra tutte le possibili chiavi di una relazione, si sceglie sempre una ed una sola chiave primaria (o Primary Key, PK) sulla quale deve valere il vincolo di impossibilità del valore nullo, ovvero il vincolo per il quale nessun valore di chiave primaria può essere nullo. Inoltre, è sempre possibile definire una superchiave per una relazione, $SK = X$; infatti, ogni tupla deve essere distinta e può essere lei stessa una superchiave (teoricamente ciò è possibile, dal punto di vista matematico ha senso perché una relazione è un insieme e in quanto tale prevede che ogni elemento sia diverso, ma nell’ottica della gestione di una base di dati perde di senso).

L’importanza della scelta della chiave primaria risiede nel suo **stretto legame con l’implementazione corretta ed efficiente di una base di dati**, dal momento in cui non sempre gli attributi di una relazione sono delle chiavi che ottimizzano l’accesso ai dati (spesso si usano codici numerici, come ISBN per i libri, perché ottimizzano in maniera efficace l’accesso ai dati, un ISBN è un numero facile da utilizzare per interrogare una base di dati).

- **Vincoli inter-relazionali**

In una base di dati relazionale, le informazioni sono distribuite su relazioni differenti, per evitare ridondanze di dati inutili che possono portare ad inconsistenza in caso di modifica della base di dati; tuttavia, questa dispersione richiede inevitabilmente meccanismi di linking che permettano di associare i dati presenti in una tabella con i dati presenti in altre tabelle.

Siano considerate le seguenti relazioni:

<u>MatStudente</u>	<u>CodiceCorso</u>	<u>Data</u>	<u>Voto</u>
150	TSI	10/10/04	30
150	A1	10/9/05	28
151	TSI	10/10/05	28

CARRIERE:

<u>Matricola</u>	<u>Nome</u>	<u>Cognome</u>	<u>Indirizzo</u>
150	Alex	Del Piero	via dei Palloni, 30
151	Martina	Stellina	via del Cielo, 40
142	Giovanni	SenzaTerra	via delle Crociate, 30

STUDENTI:

L’attributo **MatStudente** della relazione **CARRIERE** è definito sullo stesso dominio di **Matricola** della relazione **STUDENTI**, di cui è chiave primaria; dal momento in cui **MatStudente** fa riferimento alla relazione **STUDENTI**, è detta chiave esterna (Foreign Key, FK) perché attributo “esterno” al concetto espresso dalla relazione **CARRIERE**. Poiché sussiste un legame tra le tuple delle due tabelle, deve accadere che il valore di una **MatStudente** in

CARRIERE deve essere anche presente come valore di **Matricola** in **STUDENTI**; infatti, se in CARRIERE esistesse una tupla $t = (160, \text{TSI}, 2/10/90, 30)$, si genererebbe un'inconsistenza, in quanto in STUDENTI non esiste un'omonima tupla con Matricola pari a 160.

Date due relazioni, r_1 e r_2 , non necessariamente differenti, con r_1 dotata di chiave esterna FK relativa alla chiave primaria PK della relazione r_2 , si dice che tra r_1 e r_2 sussiste un vincolo di integrità referenziale se ogni occorrenza di FK in $t_1 \in r_1$ è **NULL** oppure se esiste una tupla $t_2 \in r_2 : t_1[\text{FK}] = t_2[\text{PK}]$. In altri termini, con il vincolo di integrità inter-referenziale si stabilisce che per ogni occorrenza non nulla della chiave esterna nella tabella referente sia presente un uguale valore di chiave primaria nella tabella referenziata (o riferita). Il vincolo appena enunciato spiega perché il modello relazionale è un modello basato sui valori: l'informazione, distribuita su relazioni differenti, si ricava cercando sulla base di dati la presenza di un valore “comune”.

In SQL è utile specificare, al fianco dei nomi degli attributi che fungono da chiavi esterne, il nome delle relazioni referenziate separate da : o _, in modo da evidenziare i vincoli di integrità referenziale; ad esempio: **MatStudente : STUDENTI** indica che tra **MatStudente** e la chiave primaria di **STUDENTI** sussiste un vincolo di integrità referenziale.

SQL comprende sia istruzioni per la definizione di dati (DDL) che per la loro manipolazione (DML); tra i primi, **CREATE TABLE** permette di creare una nuova relazione specificandone anche un nome, i suoi attributi (in termini di nome e tipo) e i suoi vincoli, inter-relazionali e intra-relazionali. I tipi di dato disponibili per la definizione di attributi sono tutti considerati atomici e, come da modello, sono esclusi meccanismi di strutturazione. In particolare, i tipi fondamentali disponibili sono:

- **Numerico**, comprende sia numeri interi (INT, INTEGER, SMALLINT) che numeri reali a precisione differente, in virgola fissa e mobile (NUMBER, REAL, FLOAT, DOUBLE, PRECISION);
- **Stringa**, stringhe di caratteri a lunghezza fissa (CHAR(n) con n esatto numero di caratteri) o variabile (VARCHAR(n) con n numero massimo di caratteri);
- **Binario** (BOOLEAN);
- **Data e Ora**, in formato YYYY-MM-DD HH:MM:SS.

Il blueprint per la creazione di una relazione e la definizione dei relativi vincoli intra-relazionali e inter-relazionali è configurato come segue:

```
CREATE TABLE NOME TABELLA (
    NomeAttributo Dominio [ValoreDefault] [Vincoli_InterRelazionali]
    ...
    NomeAttributo Dominio [ValoreDefault] [Vincoli_InterRelazionali]
    [AltriVincoli]
)
```

La creazione dei vincoli più importanti avviene usando le seguenti parole chiavi:

- **NOT NULL**, specifica il vincolo per il quale il valore dell'attributo deve essere diverso da NULL;
- **CHECK**, specifica una condizione che deve verificare il valore di un dato attributo;

- **UNIQUE**, specifica che uno o più attributi di una relazione sono a valore unico e serve a specificare un vincolo di chiave (infatti si parla anche di vincolo di superchiave);
- **PRIMARY KEY**, specifica che uno o più attributi sono chiave primaria di una relazione e, per default, è NOT NULL e UNIQUE;
- **FOREIGN KEY**, permette di specificare un vincolo di integrità referenziale:
 - La specifica, opzionale, delle politiche di violazione del vincolo di chiave esterna avviene attraverso le opzioni ON DELETE/UPDATE SET NULL, ON DELETE/UPDATE SET DEFAULT, ON DELETE/UPDATE CASCADE, ON DELETE/UPDATE NO ACTION (di default).

Affinché gli statement di creazione non generino errori, è necessario creare dapprima le tabelle referenziate e poi quelle referenzianti. Per basi di dati di grosse dimensioni, inoltre, è preferibile non includere negli statement di **CREATE TABLE** i vincoli di chiave esterna, possono essere aggiunti in un secondo luogo (a breve sarà chiaro come). Il linguaggio SQL fornisce apposite primitive per la manipolazione degli schemi delle basi di dati; in particolare, su schemi precedentemente creati risulta possibile modificare le definizioni di relazioni mediante l'uso del comando **DDL ALTER TABLE** (si sappia che <... | ... | ...> serve per dire che si può scegliere uno tra i termini separati da |):

```
ALTER TABLE NOME TABELLA <
    ADD COLUMN NomeAttributo Dominio [Default] [Vincoli] |
    DROP COLUMN NomeAttributo |
    ADD CONSTRAINT [NomeVincolo] Vincolo |
    DROP CONSTRAINT NomeVincolo
>
```

ALTER TABLE, quindi, permette di aggiungere o rimuovere vincoli (l'aggiunta è retroattiva) ed attributi sullo schema della relazione. È una buona prassi dare un nome ai vincoli implementati, dal momento in cui il DBMS comunica eventuali violazioni sui vincoli specificandone un ID che, se non inserito dal programmatore, sarà assegnato randomicamente dal DBMS stesso, rendendo difficile il debugging.

Il comando **DROP TABLE** permette di rimuovere dallo schema della base di dati una relazione precedentemente creata:

```
DROP TABLE NOMETABELLA <RESTRICT | CASCADE>
```

Nella cancellazione di una relazione, è possibile specificare due politiche di reazione differenti: **RESTRICT** fa sì che la rimozione della relazione ha successo se e solo se questa è vuota, mentre **CASCADE** provoca una reazione a catena e non solo viene rimossa dallo schema la relazione in questione ma anche tutte le relazioni che dipendono da essa.

A prescindere da dove sia utilizzata, la politica di **CASCADE** va ponderata cautamente: può capitare che, a causa di qualche dipendenza non considerata in fase di analisi, il comando provochi un comportamento diverso da quello atteso, causando la cancellazione non prevista di relazioni e, quindi, la perdita di informazioni.

Il linguaggio SQL fornisce anche la possibilità di includere tutte le definizioni di relazioni all'interno di un unico schema, facendo precedere le istruzioni di **CREATE TABLE** con un'istruzione di **CREATE/ALTER SCHEMA**:

```
<CREATE | ALTER > NOME SCHEMA  
CREATE TABLE NOME TABELLA (  
    NomeAttributo Dominio [ValoreDefault] [Vincoli_InterRelazionali]  
    ...
```

I moderni DBMS mettono a disposizione appositi meccanismi di sicurezza per proteggere i dati da accessi non autorizzati; uno dei compiti più importanti di un amministratore di dati (o Data Base Administrator, DBA) consiste proprio nel definire ed implementare opportune politiche di sicurezza e controllo degli accessi, supportate da SQL con apposite istruzioni. Quasi tutti i DBMS prevedono, innanzitutto, che ogni utente che deve accedere alla base di dati sia identificato in maniera univoca dal sistema ed autenticato prima di ogni accesso e che ci sia un utente amministratore attraverso il quale configurare il sistema; ciò implica che ad ogni utente siano associate delle apposite credenziali di accesso, espresse unicamente in termini di **username** e **password**. La creazione di un utente della base di dati avviene, in SQL, con la seguente sintassi:

```
CREATE USER username IDENTIFIED BY password
```

Una volta definite le credenziali di accesso, bisogna stabilire i privilegi che ogni utente deve avere sulle risorse (qualunque classe di oggetti, cioè tabelle, viste, schemi, ecc...) associate allo schema della base di dati; di norma, ogni privilegio *P* è caratterizzato da una quintupla di informazioni:

$$P = \langle R, U_1, U_2, A, T \rangle$$

Dove:

- *R* rappresenta la **risorsa** su cui è concesso il privilegio;
- *U₁* rappresenta l'**utente che concede il privilegio**;
- *U₂* rappresenta l'**utente che ottiene il privilegio**;
- *A* rappresenta l'insieme delle **azioni** che sono permesse sulla risorsa;
- *T* rappresenta l'**autorizzazione** concessa all'utente che riceve il privilegio di trasmettere lo stesso privilegio ad altri utenti.

Quando un utente crea una risorsa, un DBMS gli concede automaticamente tutti i privilegi possibili su di essa e ne fa diventare il proprietario. In genere, i principali privilegi disponibili per gli utenti sono:

- **Create**, permette di definire nuove istanze per una data risorsa;
- **Drop**, permette di rimuovere istanze di una data risorsa;
- **Update**, permette di modificare lo stato di un oggetto o istanza di una data risorsa (nel caso di tabelle, ne permette la modifica dei valori di una o più tuple);
- **Insert**, è applicabile solo a risorse di tipo tabelle e vista e permette di modificarne lo stato rimuovendo tuple da esse;
- **Select**, permette di leggere lo stato di un oggetto o istanza di una data risorsa (nel caso di tabelle, ne permette l'interrogazione e la visualizzazione dei valori di una o più tuple);
- **Resource**, permette di creare, modificare e rimuovere risorse da uno schema ed implica, quindi, anche i permessi discussi finora;
- **Connect**, permette ad un dato utente di connettersi al DBMS per poter accedere alle varie risorse;
- **All privileges**, permette ad un utente di ottenere tutti i privilegi possibili sulle risorse di una base di dati.

In SQL, la concessione di privilegi agli utenti è realizzata attraverso il comando **GRANT**:

```
GRANT priv1, ..., privN ON NomeRisorsa TO username [WITH GRANT OPTION]
```

L'opzione **WITH GRANT OPTION** corrisponde all'elemento T della quintupla del permesso. In maniera analoga, è possibile revocare uno o più permessi ad un utente mediante l'istruzione **REVOKE**:

```
REVOKE priv1, ..., privN ON NomeRisorsa FROM username [<RESTRICT | CASCADE>]
```

L'opzione **RESTRICT** impedisce che il comando sia eseguito se con tale privilegio l'utente ha creato nuove risorse oppure se è stato trasmesso ad altri utenti, mentre **CASCADE** forza l'esecuzione del comando con la rimozione di nuovi oggetti creati e la revoca dei privilegi trasmessi. Spesso, quando più utenti devono condividere gli stessi privilegi, è possibile creare un apposito ruolo o profilo:

```
CREATE ROLE NomeRuolo  
GRANT priv1, ..., privN ON NomeRisorsa1 TO NomeRuolo  
GRANT priv1, ..., privN ON NomeRisorsaN TO NomeRuolo  
GRANT NomeRuolo TO username
```

In generale, se le risorse sono associate a schemi e nel caso si vogliano concedere gli stessi privilegi su tutte le risorse di no schema, è possibile usare la sintassi:

```
GRANT priv1, ..., privN ON NomeSchema.* TO NomeRuolo [WITH GRANT OPTION]
```

In maniera analoga, la concessione di privilegi su una singola risorsa di uno schema avviene con la sintassi:

```
GRANT priv1, ..., privN ON NomeSchema.NomeRis TO NomeRuolo [WITH GRANT OPTION]
```

In un DBMS, il ruolo predefinito maggiormente utilizzato è quello di **DBA**, che include tutti i privilegi su tutte le risorse della base di dati.

OPERAZIONI DI MODIFICA E RECUPERO DELLE INFORMAZIONI

Sul modello relazionale è possibile fare, sostanzialmente, due tipi di operazioni: operazioni di modifica della base di dati, che vengono dette **updates**, e interrogazioni per il recupero delle informazioni, dette **query** o interrogazioni.

Per quanto riguarda le operazioni di modifica della base di dati sulle relazioni r_1 e r_2 (sapendo che possono essere fatte anche su più relazioni), si considerino in primis le operazioni insiemistiche: **l'unione, l'intersezione e la differenza**. Si definisce **unione tra r_1 e r_2** una nuova **relazione r** ottenuta considerando le tuple di r_1 e quelle di r_2 :

$$r = r_1 \cup r_2 = \{t : t \in r_1 \vee t \in r_2\}$$

Si definisce **intersezione tra r_1 e r_2** una nuova relazione r ottenuta considerando le tuple di r_1 che sono anche tuple di r_2 :

$$r = r_1 \cap r_2 = \{t : t \in r_1 \wedge t \in r_2\}$$

Si definisce **differenza tra r_1 e r_2** una nuova relazione r ottenuta considerando le tuple di r_1 che non sono anche tuple di r_2 :

$$r = r_1 - r_2 = \{t : t \in r_1 \wedge t \notin r_2\}$$

Nell'implementazione tecnica di una base di dati, le **operazioni insiemistiche vengono utilizzate per realizzare tre operazioni più complesse ma altrettanto fondamentali: inserimento, aggiornamento ed eliminazione.**

Un'operazione di inserimento permette di inserire una nuova tupla in una relazione elencandone i valori; se REL è il nome di una relazione definita sugli attributi $\{A_1, \dots, A_n\}$ con valori da aggiungere $\langle v_1, \dots, v_n \rangle : v_i \in \text{dom}(A_i)$, allora l'inserimento è definito come segue:

$$\text{INSERT} := \langle v_1, \dots, v_n \rangle \text{ UREL} \rightarrow \text{REL}$$

Un'operazione di inserimento può provocare una violazione dei vincoli di integrità definiti sulla **base di dati**; in particolare:

- **Violazione del vincolo di dominio**, accade se la `INSERT` inserisce valori che non appartengono al dominio corrispondente;
- **Violazione del vincolo di chiave primaria**, accade se il valore del nuovo record in corrispondenza della chiave primaria è `NULL` o se esiste già una tupla avente lo stesso valore di chiave;
- **Violazione del vincolo di integrità referenziale**, accade se il valore di una chiave esterna fa riferimento ad una chiave che non esiste nella relazione riferita.

In SQL l'inserimento è effettuato con la seguente sintassi:

```
INSERT INTO NomeTabella VALUES (ElencoValoriTupla)
```

Analogamente, un'operazione di cancellazione è definita come segue:

$$\text{DELETE} := \text{REL} - \langle v_1, \dots, v_n \rangle \rightarrow \text{REL}$$

Tuttavia, a differenza della `INSERT`, la `DELETE` rischia di violare solo il **vincolo di integrità referenziale** (nel caso in cui la tupla che si elimina contiene legami con chiavi esterne di altre relazioni); in tal caso, è possibile adottare una tra tre soluzioni:

- **Rifiuto della cancellazione**, l'azione viene ignorata (`NO ACTION`);
- **Propagazione in cascata**, vengono cancellate in cascata tutte le tuple della tabella referente che riferiscono la tupla cancellata nella tabella riferita (`CASCADE`);
- **Modifica del valore degli attributi riferiti**, ogni valore nella relazione referente è posto a `NULL` (`SET NULL`), se non esiste un vincolo di `NOT NULL`, o ad un valore di default (`SET DEFAULT`) nella tabella riferita.

In SQL:

```
DELETE FROM NomeTabella WHERE Condizione
```

Si rende necessario individuare la tupla che si vuole cancellare attraverso una condizione logica.

Analogamente, si definisce operazione di **UPDATE** un'operazione che cambia i valori di uno o più attributi di tuple presenti in una relazione REL:

$$\text{UPDATE} := \text{REL} - \langle v'_1, \dots, v'_n \rangle \rightarrow \text{REL} \wedge \langle v_1, \dots, v_n \rangle \cup \text{REL} \rightarrow \text{REL}$$

Con $\langle v'_1, \dots, v'_n \rangle$ valori da aggiornare e $\langle v_1, \dots, v_n \rangle$ valori aggiornati. In caso di inserimento, il DBMS deve:

1. Verificare la condizione $v_i \in \text{dom}(A_i)$;
2. Se è modificata una chiave primaria, analogamente alla INSERT, deve essere soddisfatto il relativo vincolo mentre, analogamente alla DELETE, vanno considerate le relative politiche di violazione dei vincoli;
3. Se la modifica coinvolge una chiave esterna, il DBMS deve verificare che il nuovo valore nella tabella referente sia presente nella tabella riferita, o sia NULL.

In SQL:

```
UPDATE NomeTabella SET {Attributo = Espressione, Attributo = Espressione, ...} [WHERE Condizione]
```

Nell'espressione si può porre il risultato di una query SQL o una costante, il valore di default o il valore NULL.

Per **operazioni relazionali** si intendono le **operazioni che sono specifiche del modello relazionale e non hanno un seguito nella teoria insiemistica**. Sia dato uno schema di relazione $R(X)$, sia Y un sottoinsieme di attributi dello schema, $Y \subseteq X$, e sia r un'istanza di R , si definisce **proiezione della relazione r rispetto a Y** l'insieme dei valori di Y di tutte le tuple t della relazione:

$$\pi_Y(r) = \{t[Y], t \in r\}$$

In SQL:

```
SELECT [DISTINCT] ElencoAttributi FROM NomeTabella
```

Estraendo dalla tabella specificata le colonne di cui sono indicati gli attributi. La clausola DISTINCT permette di eliminare le righe ripetute.

L'operazione di selezione permette di scegliere tra le tuple di una relazione quelle che soddisfano uno specificato predicato logico. Si introducano le seguenti definizioni:

- **Condizione atomica**, sia r una relazione su $R(X)$, siano $A \in X$ e $B \in X$ due attributi di r , sia $\theta \in \{=, \neq, <, >, \leq, \geq\}$ una qualsiasi relazione definita sul dominio di A e B , si definiscono condizioni atomiche $A\theta B$ e $A\theta c$ con c costante appartenente al dominio di A ;
- **Condizione di selezione**, ogni condizione atomica è una condizione di selezione e, inoltre, se vc_1 e vc_2 sono due condizioni di selezione, lo sono pure $vc_1 \wedge vc_2$, $vc_1 \vee vc_2$ e $\neg vc_1$;
 - Una condizione di selezione, quindi, è tale se è composta o da una condizione atomica o da una qualsiasi espressione logica ottenuta componendo più condizioni atomiche;

- **Soddisfacimento di una condizione di selezione**, sia $R(X)$ uno schema di relazione, r un’istanza di R , t una tupla di r , A e B attributi in X e c una costante, sia χ una qualsiasi condizione di selezione su A , B e c , il soddisfacimento di χ da t ($t \models \chi$) è definito per induzione sulla struttura di χ , come segue:
 - Paragone tra attributi: $t \models A\theta B \Leftrightarrow t[A]\theta t[B]$;
 - Paragone con costante: $t \models A\theta c \Leftrightarrow t[A]\theta c$;
 - Nel caso di espressione composta, essa deve essere soddisfatta tenendo conto degli operatori logici presenti nell’espressione stessa.

Con ciò a disposizione, è possibile dare la definizione di selezione. Sia data una relazione $r \in R(X)$ ed una condizione di selezione χ , è **definito selezione l’insieme**:

$$\sigma_\chi(r) = \{t \in r : t \models \chi\}$$

In SQL:

```
SELECT * FROM NomeTabella WHERE Condizione
```

Si noti che **un’operazione di selezione genera una decomposizione orizzontale della relazione su cui è applicata**. Una classica **query SQL** si ottiene **combinando una proiezione con una selezione**:

$$\pi_Y(\sigma_\chi(r))$$

Che in SQL:

```
SELECT Y FROM r WHERE \chi
```

Con Y detto anche elenco degli attributi target, se si usa $*$ allora la proiezione è eseguita su tutti gli attributi della relazione, con $Y = X$.

Per quanto riguarda i valori del tipo stringa, è uso far riferimento anche ad operatori differenti rispetto a quelli classici di confronto tra elementi di un insieme, precedentemente elencati; in particolare, si richiede spesso di effettuare **confronti solo tra parti di una stringa**, con l’operatore `LIKE` in SQL seguito da uno tra due caratteri speciali:

- `%`, per indicare una qualsiasi sequenza di caratteri;
- `_`, per indicare un qualsiasi carattere in una posizione specifica.

Ad esempio, `_e%a` indica le stringhe che hanno la `e` in seconda posizione e che finiscono con `a`.

Ad un valore o attributo appartenenti a domini di tipo numerico, invece, è **possibile applicare anche gli operatori aritmetici classici** (`+, -, *, /`). A volte può essere utile **richiedere un ordine di visualizzazione del risultato di una query, ordinando le tuple del risultato rispetto ad uno o più attributi, attraverso la clausola**:

```
ORDER BY { Attributo [<ASC, DESC>], Attributo [<ASC, DESC>], ...}
```

Dove **ASC** (di default) **indica una modalità di ordinamento ascendente** e **DESC** discendente.

La **join** è sicuramente una delle **operazioni fondamentali e più utili del modello relazionale**. Volendo generalizzare, è possibile dire che **la join è usata per mettere assieme informazioni che**

sono presenti in due o più relazioni, sebbene la sua definizione rigorosa richieda quella del prodotto cartesiano. Date due relazioni, r_1 e r_2 , definite rispettivamente sugli schemi $R_1(X_1)$ e $R_2(X_2)$ tali che $X_1 \cap X_2 = \emptyset$, il **prodotto cartesiano tra r_1 e r_2 restituisce un'istanza di una relazione r definita sullo schema $R(X_1 \cup X_2)$ e contiene le tuple t formate dalla concatenazione di ciascuna tupla di r_1 con ciascuna tupla di r_2** , ovvero:

$$r = r_1 \times r_2 = \{t = \langle t_1, t_2 \rangle : t_1 \in r_1 \wedge t_2 \in r_2\}$$

Si noti che, **in presenza di attributi uguali su tabelle diverse**, si può utilizzare la cosiddetta **dot notation**, che permette di **distinguere gli attributi comuni facendo esplicitamente riferimento al nome della tabella di partenza**. Da quanto appena detto, in particolare a partire dalla definizione di prodotto cartesiano, è possibile definire la theta join, una delle tre possibili tipologie di join: date due relazioni, r_1 su $R(X_1)$ e r_2 su $R(X_2)$, con $X_1 \cap X_2 = \emptyset$, **si definisce theta join l'operazione**:

$$\sigma_\chi(r_1 \times r_2) \equiv r_1 \bowtie_\chi r_2$$

Si noti che **si sta parlando di un'operazione di prodotto cartesiano applicato ad una selezione**. La condizione χ può essere composta, come al solito, dagli operatori previsti per la selezione; se χ è una relazione di uguaglianza tra un attributo di r_1 ed un attributo di r_2 , allora il theta join viene comunemente chiamato **equi join**, usato comunemente per **congiungere le tabelle** eventualmente legate da vincoli di integrità referenziali.

In SQL, l'operazione di join può essere eseguita in un modo implicito ed in un modo esplicito; in particolare, il join implicito si ottiene applicando la definizione data di equi join:

```
SELECT * FROM r1, r2 WHERE Condizione
```

Considerando che **la virgola agisce come operatore di prodotto cartesiano**. La join esplicita, invece, **richiede che venga indicato l'operatore di JOIN come di seguito mostrato**:

```
SELECT * FROM r1 JOIN r2 ON Condizione
```

La differenza sostanziale che intercorre tra i due metodi, considerando che il risultato è lo stesso, consiste nella proiezione, che avviene nel metodo implicito e non avviene nel metodo esplicito, evitando al DBMS un carico di memoria troppo oneroso. Inoltre, è possibile effettuare operazioni di join a più tabelle innestando più JOIN nella stessa istruzione, richiedendo però la forma esplicita.

Un caso speciale di equi join è il secondo tipo di join, la join naturale e occorre quando $X_1 \cap X_2 \neq \emptyset$, ovvero quando le due relazioni **hanno attributi in comune**. Il join naturale è anche indicato con $r_1 \bowtie r_2$ e, in tal caso, la condizione χ è implicitamente verificata su tutte le coppie di attributi delle due relazioni aventi lo stesso nome:

$$r_1 \bowtie r_2 \equiv r_1 \bowtie_{r_1.A_c=r_2.A_c} r_2$$

Si noti che **non tutte le tuple delle relazioni coinvolte concorrono a formare la relazione risultato di un join naturale**, quelle non coinvolte sono dette **dangling**. La presenza di questo tipo di tuple può essere, in certi casi, **dannosa, lasciando indietro alcune informazioni che potrebbero essere importanti**; per evitare questo problema, è introdotto il terzo tipo di join, il **join esterno**, diviso a sue volte in tre varianti sulla base delle due relazioni coinvolte r_s (o relazione sinsistra) e r_d (o relazione destra):

- Join sinistra, $r_s \bowtie_{LEFT} r_d$;
- Join destra, $r_s \bowtie_{RIGHT} r_d$;
- Join completa, $r_s \bowtie_{FULL} r_d$.

Il join esterno prevede che tutte le tuple della relazione r_s o che tutte le tuple della relazione r_d o che tutte le tuple di entrambe concorrono alla formazione della relazione risultato, ponendo NULL laddove l'operazione di join coinvolge tuple dangling; in particolare:

- La **join esterna sinistra** prevede che vengano considerate tutte le tuple della relazione sinistra;
- La **join esterna destra** prevede che vengano considerate tutte le tuple della relazione destra;
- La **join esterna completa** prevede che vengano considerate tutte le tuple di entrambe le relazioni.

Per come è stato definito il prodotto cartesiano, **se il theta join è fatto su relazioni che hanno nomi di attributi uguali, non è possibile effettuare l'operazione**; pertanto, **è necessario adottare misure per permettere questo tipo di operazione**, come la **definizione dell'operatore di ridenominazione**: sia r una relazione definita su uno schema $R(X)$ e sia Y un insieme di attributi aventi la stessa cardinalità di X , sia definito un ordinamento tra gli attributi di X , $A_1 \dots A_k$, e quelli di Y , $B_1 \dots B_k$, con la proprietà per la quale $dom(A_i) = dom(B_i)$, l'operatore di ridenominazione:

$$\rho_{B_1 \dots B_k \leftarrow A_1 \dots A_k}(r) = \{t' : \exists t \in r \wedge t[A_i] = t[B_i] \forall i = 1 \dots k\}$$

Tale definizione afferma che **le tuple della relazione ridenominata hanno lo stesso valore della tupla di partenza**, mentre **il nome degli attributi viene modificato**. In SQL la ridenominazione avviene tramite il meccanismo di alias, mediante l'operatore AS:

```
SELECT Attributo1 AS Alias1, ..., AttributoK AS AliasK FROM NomeTabella
```

Nel caso in cui **la clausola FROM contempla più tabelle, si deve ricorrere alla dot notation** (`NomeTabella.NomeAttributo`) e, a tal fine per semplificare l'indicazione delle tabelle, si può **estendere la ridenominazione ai loro nomi nella clausola FROM**:

```
SELECT T1.A1, T1.A2, T2.A3
FROM TabellaUno T1, TabellaDue T2
WHERE T1.Ak = T2.Ah
```

Si noti che **la ridenominazione nella clausola FROM corrisponde alla definizione di variabili di tipo tabella, consentendo così di utilizzare più volte la stessa tabella in un'interrogazione, mentre nella clausola SELECT permette di assegnare nomi ad attributi ed espressioni per rendere più leggibile il risultato di una query; inoltre, l'alias su variabili e su tabelle consente anche di usare in un'interrogazione una stessa relazione più volte**.

Per **funzioni aggregate** si intende un **insieme di funzioni che permettono di operare su collezioni di valori della base di dati** e, in SQL, sono: COUNT, SUM, MAX, MIN e AVG. Inoltre, **si può prevedere di raggruppare dapprima le tuple sulla base del valore di uno o più attributi**, mediante opportune **clausole**, dette **di raggruppamento**. Si consideri che **le operazioni di aggregazione e raggruppamento non sono definite nell'algebra relazionale e non hanno un seguito per quanto riguarda la teoria insiemistica** (come sarà solito da qui in avanti). Come già anticipato, **SQL supporta cinque diverse funzioni di aggregazione, divise in due gruppi**:

- **SUM()**, **MAX()**, **MIN()** e **AVG()**, sono funzioni che, **applicate all'insieme di valori numerici risultanti da un'interrogazione**, ne restituiscono, rispettivamente, la somma, il massimo, il minimo e la media aritmetica;
 - Possono essere **applicate a qualsiasi attributo della relazione o ad un'espressione avente come operandi attributi della relazione**, a patto che sia un **attributo di proiezione**;
- **COUNT(*)**, **STDEV** e **VARIANCE**, restituiscono, rispettivamente, il numero di tuple presenti nel risultato di un'interrogazione, la deviazione standard e la varianza;
 - **È possibile applicare questa funzione anche ad un attributo o ad un elenco di attributi**, considerando in questo caso **anche eventuali duplicazioni**.

Tutte queste funzioni, ad eccezione di COUNT, ignorano i valori NULL e, in generale, se tutti i valori sono NULL restituiscono NULL. Si noti che **gli operatori aggregati operano**, in genere, **su tutti i valori presenti nel risultato di una query**; quindi, se si vogliono escludere eventuali duplicazioni è necessario usare la clausola **DISTINCT** (che, però, non ha senso per le funzioni MIN e MAX). Inoltre, nello standard, **gli operatori aggregati non operano sui valori nulli**, se si vogliono includere le righe in questione (relativamente, ovviamente, agli attributi su cui operano gli operatori aggregati) occorre usare la clausola ALL.

Le operazioni di raggruppamento precedentemente menzionate sono ottenute attraverso le seguenti clausole:

- **GROUP BY ElencoAttributi**, permette di **raggruppare le tuple della relazione in sottoinsiemi caratterizzati dallo stesso valore degli attributi che compaiono come argomento della clausola stessa**;
 - Il funzionamento è del tutto **analogo a quello di una ORDER BY** ma, ad un occhio attento, non sfugge che **i risultati possono essere diversi** (soprattutto in termini di termini di ottimizzazione della memoria e di correttezza del risultato): **una GROUP BY ha senso in un contesto di aggregazione e/o di statistiche analitiche sui dati**, ottenendo **tanti valori quanti gruppi si vanno a formare**, mentre la **ORDER BY in casi complementari**;
- **HAVING (Condizione)**, permette di **specificare delle condizioni logiche (predicati) che devono essere verificate sul sottoinsieme di tuple precedentemente raggruppate con la GROUP BY** e, pertanto, può agire come filtro;
 - Il funzionamento è del tutto **analogo a quello di una WHERE** ma le due possono essere **distinte sulla base del seguente criterio**: se le condizioni devono essere verificate **a livello delle singole tuple**, si usa la clausola **WHERE**, se le condizioni sono **verificabili su un gruppo**, si usa la clausola **HAVING**.

Si noti che, **in un'interrogazione SQL contenente la clausola GROUP BY, un attributo può comparire come argomento della clausola SELECT se e solo se è presente nell'elenco degli attributi della clausola GROUP BY**; infatti, se un attributo appare nella clausola SELECT ma non nella GROUP BY, possono esserci diverse tuple all'interno del gruppo che su quegli attributi hanno valori diversi, non consentendo un raggruppamento adeguato. Inoltre, **si noti la possibilità di dare degli alias al fine di dare un nome, e quindi una semantica, al risultato di una funzione di aggregazione**, essenziale ai fini della leggibilità e dell'usabilità della tabella risultato della query.

SQL permette di usare, all'interno di query, **anche gli operatori insiemistici**, con parole chiavi UNION, INTERSECT e EXCEPT per, rispettivamente, l'unione, l'intersezione e la differenza (nonostante per quest'ultima Oracle utilizzi MINUS).

In alcuni casi, **può essere utile esprimere le query SQL con una struttura più complessa** in cui, nella clausola WHERE, la condizione è costituita dal confronto del valore di un'espressione con il risultato di un'altra query SQL (query nidificate). Tipicamente, **il confronto è tra il valore di un attributo dello schema e l'insieme dei valori risultanti dal calcolo dell'interrogazione** nidificata (quindi non ci saranno più di due nesting); ovviamente, **il tipo del valore dell'attributo deve coincidere con quello dei valori restituiti dalla query**. Nel caso in cui sia necessario esprimere su una relazione una condizione che deve essere a sua volta calcolata, **è necessario ampliare l'insieme di operatori di confronto con i seguenti**:

- **op ALL**, la condizione specificata da op è verificata se e solo se è verificata da tutti gli elementi restituiti dalla query nidificata;
- **op ANY**, la condizione specificata da op è verificata se e solo è verificata da almeno un elemento restituito dalla query nidificata;
- **IN e NOT IN**, sono del tutto identici, rispettivamente, a =ANY e ≠ALL.

Si noti che **nella clausola WHERE, può essere anche espresso l'operatore di confronto unario EXISTS** (e NOT EXISTS), **che verifica se un insieme di valori risultato di una query nidificata è non vuoto (oppure vuoto) e coincide con un confronto implicito con l'insieme vuoto**.

Spesso capita che **una query nidificata possa essere fatta anche attraverso un'operazione di JOIN**, sfruttando le caratteristiche dichiarative di SQL, perdendo, tuttavia, **leggibilità e guadagnando complessità**.

Per **vista** si intende **una tabella descritta**, ricorsivamente, in termini di altre tabelle, dette di base. In generale, **una vista è una relazione virtuale**, nel senso che le sue tuple non sono effettivamente memorizzate nella base di dati ma, piuttosto, ricavabili attraverso query presenti nella base di dati; quindi, **una vista costituisce un'interfaccia da mettere a disposizione di utenti o applicazioni per eventuali query**: si tratta di **dati frequentemente utilizzati** che possono anche **non esistere fisicamente**. In SQL:

```
CREATE [MATERIALIZED] VIEW NomeVista AS QuerySQL
```

Il problema dell'implementazione di una vista è molto complesso e non riguarda gli scopi di questa trattazione; tuttavia, **sono suggerite due modalità menzionate spesso nella letteratura di settore e utilizzate da molti DBMS commerciali**:

- **Modifica al momento dell'interrogazione**

La vista viene calcolata come risultato di una query ogni volta la si usa, consentendo di tenere allineata una vista ai dati sottostanti ma portando con sé una bassa efficienza su grosse moli di dati e su query molto complesse. Le operazioni di modifica sulla vista si ripercuotono sulle tabelle di base qualora la vista sia aggiornabile, ovvero derivata o da singole tabelle o da operazioni di join tra più tabelle in cui la condizione di join coinvolge solo chiavi primarie; altrimenti, il DBMS rifiuta l'operazione.

- **Viste materializzate**

Richiedono la creazione fisica di una tabella della vista quando viene eseguita la prima volta una query. In questo caso, la vista è una “fotografia” dei dati in quell’istante di tempo e vanno previste strategie di aggiornamento automatico della vista.

Le **operazioni di analisi**, soprattutto in confronto con le subquery, sono più efficienti ma continua a sussistere il problema dell'allineamento dei dati tra le tabelle di base e la vista, che deve essere risolto con l'adozione di opportune procedure di allineamento.

Si è, dunque, nella condizione di poter mostrare la **forma più generica di query SQL**:

```
SELECT ElencoAttributiProiezione + FunzioniAggregazione  
FROM ElencoTabelle  
[WHERE CondizioneLogica]  
[GROUP BY ElencoAttributi]  
[HAVING CondizioneRaggruppamento]  
[ORDER BY ElencoAttributi]
```

Se c'è la **GROUP BY**, tutti gli attributi specificati nella **SELECT** devono essere attributi di raggruppamento; infatti, in ordine, viene fatta dal DBMS prima la **GROUP BY** e poi la proiezione.

Volendo individuare un **criterio generale per realizzare una query**, si consideri la possibilità di **rispondere alle seguenti domande**, in ordine:

1. Dove si trovano le informazioni per risolvere la query? [FROM] (dove prendere i risultati)
2. Servono tutte le informazioni? [WHERE] (quali risultati prendere)
3. Bisogna effettuare raggruppamenti? [GROUP BY] (si hanno già i risultati)
4. Bisogna aggregare? [FunzioniAggregazione, dopo SELECT] (per eventuali statistiche)
5. Servono tutti i gruppi o si vuole escludere qualcosa? [HAVING]
6. Cosa si vuole visualizzare? [ElencoAttributiProiezione]
7. Bisogna ordinare? [ORDER BY]

Con l'introduzione dello statement **SELECT**, le **operazioni di insert, update e delete possono riferirsi anche ad insiemi di tuple**, consentendo così l'inserimento, l'aggiornamento e la cancellazione di informazioni all'interno della base di dati attraverso un unico costrutto:

```
INSERT INTO NomeTabella [ElencoAttributi]  
<VALUES ElencoValori | QuerySQL>  
  
DELETE FROM NomeTabella [WHERE Condizione]  
  
UPDATE NomeTabella  
SET {Attributo=<Espressione | QuerySQL | NULL | DEFAULT>, ... }  
WHERE Condizione
```

LA NORMALIZZAZIONE

Le forme normali sono state introdotte con l'intento di fornire un criterio di scelta tra i vari schemi relazionali che possono modellare una data realtà di interesse. Occorre subito precisare che la teoria della normalizzazione non è una tecnica di progettazione di una base di dati, fornisce solo uno strumento di verifica della sua ottimizzazione. In genere, la "bontà" di una relazione è verificata in assenza di ridondanza dei dati contenuti nella base di dati e in assenza di anomalie in caso di operazioni di modifica; tipicamente, la ridondanza è causata dalle anomalie a fronte di operazioni di modifica di una relazione, causando un'alterazione dello stato della base di dati. In genere, su una data relazione, possono verificarsi i seguenti tipi di anomalie:

- **Anomalia di aggiornamento**, si crea inconsistenza se l'aggiornamento di un dato non è esteso a tutte le tuple in cui esso compare;
- **Anomalia di cancellazione**, non è possibile cancellare certe informazioni senza perderne altre ad esse correlate;
- **Anomalia di inserimento**, l'inserimento di una nuova tupla in una relazione, avente (ad esempio) vincoli di NOT NULL sui suoi attributi, è impossibile se non si ha a disposizione l'informazione relativa a tutti gli attributi della tupla stessa.

Uno dei concetti più importanti nel modello relazionale è quello di **dipendenza funzionale**, **un vincolo tra due insiemi di attributi di una relazione**. Considerata una relazione r su $R(X)$ e due sottoinsiemi non vuoti, Y e Z , si dice che **esiste in r una dipendenza funzionale (FD) da Y a Z** (o viceversa) se, per ogni coppia di tuple t_1 e t_2 di r con gli stessi valori su Y , risulta che t_1 e t_2 hanno gli stessi valori anche su Z ; ovvero:

$$Y \rightarrow Z \Leftrightarrow \forall t_1, t_2 \in r : t_1[Y] = t_2[Y] \Rightarrow t_1[Z] = t_2[Z]$$

Ovvero, **si ha dipendenza funzionale quando ogni valore della componente Y determina univocamente il valore della componente Z** . Una **dipendenza funzionale $Y \rightarrow Z$** , con Y e Z due insiemi di attributi, **si dice banale** se esiste un sottoinsieme di Z incluso in Y , ovvero se:

$$\exists K \subseteq Z : K \subseteq Y$$

Nella pratica, si è interessati a dipendenze funzionali non banali. La dipendenza funzionale è legata alla semantica che il progettista di una base di dati assegna ad ogni attributo, è una caratteristica dello schema (dell'aspetto intenzionale) e non della particolare istanza dello schema. La semantica potrebbe anche cambiare nel tempo, essendo associata ad oggetti e proprietà del mondo reale modellato ed essendo questi potenzialmente mutabili nel tempo. **Un'istanza di uno schema che rispetti una data dipendenza funzionale è detta istanza legale dello schema rispetto alla data dipendenza funzionale**.

Il concetto di superchiave può essere espresso anche in funzione del concetto di dipendenza funzionale; infatti, è banalmente dimostrabile il seguente teorema.

ENUNCIATO TEOREMA DI SUPERCHIAVE COME DIPENDENZA FUNZIONALE

Ipotesi:

$\forall R(X)$ schema di relazione

Tesi:

$K \subseteq X$ è superchiave di $R(X) \Leftrightarrow K \rightarrow X$

Vale la seguente definizione: dato uno schema di relazione $R(X)$, **se un attributo partecipa almeno ad una chiave di $R(X)$ è detto attributo primo, altrimenti attributo non primo**; inoltre, tutti gli attributi non primi dipendono da una chiave. A partire da ciò, si enunci il teorema di generalizzazione del vincolo di chiave.

ENUNCIATO TEOREMA DI GENERALIZZAZIONE DEL VINCOLO DI CHIAVE

Ipotesi:

$\forall R(X)$ schema di relazione

$K \subseteq X$ è superchiave di $R(X)$

Tesi:

$\forall Z$ sottoinsieme di attributi non primi di $R(X)$, $K \rightarrow Z$ con $K \cup Z \subseteq X$

Il teorema in questione generalizza quello precedente. Dati gli insiemi di attributi Z e Y , **una dipendenza funzionale $Y \rightarrow Z$ è completa se Z non dipende da nessun altro sottoinsieme di Y .** Ovvero, **eliminando un qualsiasi attributo di Y , la dipendenza funzionale $Y \rightarrow Z$ non sussiste più;** inoltre, **se Y è chiave, la dipendenza funzionale è completa se Z dipende solo dalla chiave e non anche da una parte di essa.** Infine, una dipendenza funzionale $Y \rightarrow Z$ è transitiva se $\exists A \subseteq X : Y \rightarrow X \wedge A \rightarrow Z$.

Uno schema di relazione $R(X)$ è detto in **prima forma normale** se ogni attributo appartenente ad X è **un attributo semplice** (ovvero se per ogni attributo il relativo dominio è atomico). La **prima forma normale (1NF)** garantisce che ogni attributo non primo dipenda da **una chiave di $R(X)$** ; affinché una relazione sia in 1NF, non deve contenere nel suo schema attributi multivalore (attributi i cui valori sono insiemi di valori) né attributi strutturati (attributi i cui valori sono n-uple di valori). Volendo essere pragmatici, il **modello relazionale è già di per sé un modello in 1NF**, visto che la maggior parte dei DBMS relazionali mettono a disposizione, per la definizione dei domini, **tipi atomici, impedendo l'esistenza di meccanismi linguistici di strutturazione o di set;** nel caso in cui questo tipo di attributi sia possibile, per eliminarli (come sarà più chiaro a breve) è necessario creare delle tabelle aggiuntive per “decomporli”.

È possibile notare che **per normalizzare le relazioni non in 1NF è necessario procedere “sviluppando” gli attributi multivalore ed “estraendo” gli attributi strutturati dalle relazioni originarie.**

La 1NF non previene le anomalie di inserimento, aggiornamento e modifica, con la **seconda e la terza forma normale** che cercano di porvi rimedio usando la teoria delle dipendenze funzionali. Uno schema di relazione $R(X)$ è in **seconda forma normale** se è in prima forma normale e se **ogni attributo non primo di $R(X)$ è in dipendenza funzionale completa da ogni chiave di $R(X)$.** In altre parole, **non è possibile rimuovere da una chiave un attributo senza perdere qualche dipendenza funzionale.** Il test per la 2NF prevede l'esame delle dipendenze funzionali degli attributi non primi da ogni chiave della relazione; resta inteso che **se ogni chiave di una relazione è formata da un solo attributo ed inoltre essa è in 1NF, allora tale relazione è in 2NF.**

Se uno schema di relazione non è in 2NF, può facilmente essere normalizzato in 2NF scomponendo le relazioni di partenza in un certo numero di relazioni che soddisfano la 2NF; per riferirsi a questo tipo di procedimento, si parla di **decomposizione**. In altre parole, **gli attributi vanno assegnati a tabelle separate.** In generale, **occorre garantire che la decomposizione non alteri il contenuto informativo della base di dati**, ovvero che la decomposizione sia **lossless**: uno schema di relazione $R(X)$ si **decompone senza perdite negli schemi $R_1(X_1)$ e $R_2(X_2)$** se, per ogni istanza legale r su $R(X)$, si ha che $\pi_{X_1}(r) \bowtie \pi_{X_2}(r) = r$. Si può dimostrare che **la decomposizione**

senza perdita è garantita se gli attributi comuni contengono una chiave per almeno una delle relazioni decomposte.

Per quanto attiene le dipendenze funzionali, **vale la definizione di dependency preservation: le dipendenze funzionali presenti nella relazione di partenza devono essere mantenute anche dopo la decomposizione.** Lo scopo della decomposizione, quindi, è quello di ridurre la ridondanza dei dati mantenendo, però, le informazioni della relazione originale, senza perdere tuple o aggiungerne delle nuove senza significato (tuple spurie). Sebbene la proprietà di dependency preservation possa non essere rispettata, la proprietà di lossless decomposition è cruciale alla 2NF; infine, si noti che, in genere, la decomposizione di una relazione non è unica.

Si vuole adesso introdurre un **terzo tipo di forma normale, basata interamente sul concetto di dipendenza transitiva;** dato uno schema di relazione $R(X)$, un attributo $A \subset X$ dipende transitivamente dall'insieme di attributi $Y \subset X$ se esiste un altro insieme di attributi $Z \subset X$ tale che:

- $Y \rightarrow Z \wedge Z \not\rightarrow Y$;
- $Z \rightarrow A \wedge A \not\rightarrow Z$.

Con $A \notin Y \cup Z$. Con ciò a disposizione è possibile dire che **uno schema di relazione $R(X)$ è in terza forma normale** se:

- È in 2NF;
- Ogni attributo non primo di $R(X)$ non dipende transitivamente da ogni chiave di $R(X)$.

In altre parole, **ogni attributo non primo dipende dalla chiave (1NF) e da tutta la chiave (2NF) e solo dalla chiave (3NF).** Uno schema in 3NF impone, quindi, che un attributo non primo dipenda completamente e non transitivamente da ogni chiave di $R(X)$; affinché ciò sia soddisfatto, per ogni dipendenza funzionale del tipo $Y \rightarrow Z$ possono essere verificate due diverse condizioni:

- Y è superchiave, la dipendenza è non transitiva e completa;
- Y è un attributo non primo, l'unica possibilità affinché la 3NF sia soddisfatta è che Z non sia un attributo “non primo”, altrimenti la dipendenza è transitiva.

Da ciò, si deduce che **Z deve essere un attributo primo.** Con queste informazioni, è possibile definire alternativamente 3NF: uno schema di relazione $R(X)$ è in 3NF se e solo se per ogni FD (non banale) $Y \rightarrow Z$ definita su $R(X)$ si ha che o Y è superchiave di R o Z è attributo primo. Se uno schema di relazione non è in 3NF, può essere facilmente normalizzato decomponendo la relazione di partenza in un certo numero di relazioni che soddisfino la 3NF, senza perdere alcuna informazione dalla relazione originaria.

Con la terza forma normale si evitano le dipendenze transitive di attributi non primi della chiave; tuttavia, **possono verificarsi delle anomalie nel caso in cui esiste una dipendenza funzionale del tipo $Y \rightarrow Z$ con Y attributo non primo e Z attributo primo.** Per risolvere questo tipo di problemi è stata introdotta un’ulteriore forma normale, detta **forma normale di Boyce e Codd**; uno schema di relazione $R(X)$ è in BCNF se:

- È in 1NF;
- Per ogni dipendenza funzionale non banale $Y \rightarrow Z$, Y contiene una chiave per $R(X)$ (ovvero se Y è superchiave di $R(X)$).

BCNF garantisce, dunque, **che tutti gli attributi non primi dipendano dagli attributi primi e che tutti gli attributi primi non dipendano dagli attributi non primi**. È facilmente dimostrabile il seguente teorema.

ENUNCIATO TEOREMA DI RELAZIONE TRA 3NF E BCNF

Ipotesi:

$\forall R(X)$ schema di relazione

$R(X)$ è in BCNF

Tesi:

$R(X)$ è in 3NF

Il teorema in questione non ammette inversioni: una relazione in 3NF non necessariamente è in BCNF, anche se la maggior parte delle relazioni in 3NF lo sono. Se uno schema non è in BCNF, si cerca al solito di effettuare una decomposizione. **BCNF non rimuove tutte le ridondanze: alcune sembrano essere inevitabili**, specialmente quando si ha a che fare con attributi multivale (che, come già specificato, non sono possibili nella maggior parte dei DBMS commerciali); inoltre, uno dei motivi per cui BCNF può non essere preso in considerazione risiede nell'elevata necessità temporale per poter effettuare la normalizzazione.

La teoria della normalizzazione fornisce delle regole di progetto che cercano di prevenire anomalie di aggiornamento e inconsistenza dei dati. Si noti, tuttavia, che non sempre relazioni in forme normali sono le più utili ai fini del recupero delle informazioni; infatti, informazioni che in tabelle non normalizzate si trovano in un'unica tupla, nel caso di schemi fortemente normalizzati devono essere recuperate attraverso una o più operazioni relazionali (tipicamente di join). Nella pratica della progettazione, si cerca sempre un trade-off tra queste diverse esigenze (normalizzazione ed efficienza nel recupero delle informazioni), anche sulla base di considerazioni relative alla frequenza delle operazioni stesse; in alcuni casi, informazioni che necessitano di essere accedute frequentemente e che sono spalmate su più relazioni sono di solito memorizzate in viste non normalizzate della base di dati.

IL MODELLO ER

Quando si parla di **applicazione di basi di dati** ci si riferisce ad **una particolare base di dati ed ai programmi che operano su di essa**; ne consegue che **la progettazione di un'applicazione di basi di dati procede su due binari distinti** che, solitamente, evolvono in parallelo:

- **Progettazione della base di dati**

La progettazione della base di dati deve essere **indipendente dal particolare modello logico dei dati** e, pertanto, si procede con una **modellazione di alto livello** che consente di esprimere i concetti, le associazioni tra gli stessi e i vincoli che desumono dalla realtà. Per questo motivo, **la prima fase della progettazione di una base di dati è detta progettazione concettuale**, la cui traduzione in un

modello logico dei dati è assicurato dalla fase di **progettazione logica**; l'ultima fase del percorso di progettazione di una base di dati consiste nella **progettazione fisica**, nella quale **si scelgono le particolari strutture di memorizzazione interna**.

- **Progettazione delle procedure applicative**

La progettazione delle procedure applicative **assolve al compito di determinare le procedure atte alla gestione dei dati a livello applicativo nel rispetto dei vincoli prefissati** a livello della base di dati.

Fino a questo momento è stato mostrato un tipo particolare di progettazione logica, basata sull'impiego di SQL per l'interrogazione e l'operazione su una base di dati già determinata a livello fisico e concettuale; di seguito, invece, verrà proposto un particolare modello di progettazione concettuale, molto usato nell'ambito della progettazione di basi di dati, detto **Entity Relationship Model**, o **modello ER**. La versione che verrà illustrata di seguito è quella originale del 1976 ideata da Chen.

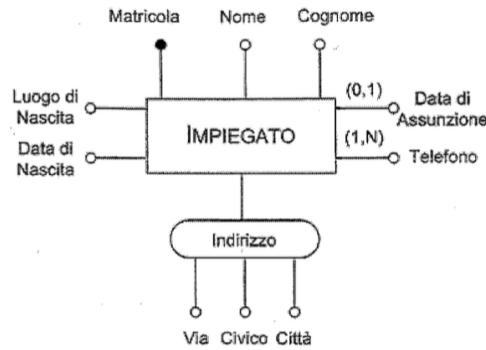
Il concetto chiave del modello ER è quello di entità, come suggerisce l'acronimo del nome. Un'occorrenza di entità è un concetto che esprime un oggetto realmente esistente nel mondo reale, con una sua esistenza autonoma, fisica o astratta, e dotato di proprietà significative per un dato dominio di interesse; da ciò, si definisce entità una collezione di occorrenze di entità che possiedono le stesse proprietà, le quali prendono il nome di attributi. Il rapporto tra entità e occorrenza di entità è lo stesso che intercorre nella Programmazione Orientata agli Oggetti tra classe e oggetto.

Un'entità è rappresentata nei diagrammi ER con un rettangolo contenente il suo nome, identificativo univoco nello schema ER (si è soliti porre come nomi di entità nomi significativi nel contesto del modello e al singolare, essendo rappresentazione di insiemi). Gli attributi, invece, sono individuati da linee terminate da cerchi con i relativi nomi e sono riferiti ad una singola entità; le linee in questione possono essere completate con la cardinalità dell'attributo, con la quale si esprime un numero minimo e un numero massimo di valori che un attributo può assumere, con riferimento alle occorrenze delle entità. I due numeri in questione vengono riportati tra parentesi (min, max) e deve necessariamente accadere che:

$$0 \leq min \leq max$$

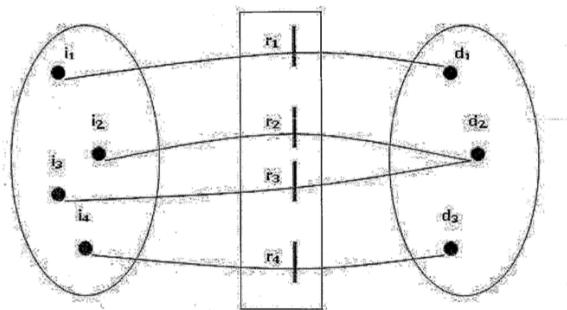
Il caso di **valore minimo della cardinalità pari a zero** indica che **l'attributo può assumere valore nullo**. Si noti nel diagramma seguente, la presenza di:

- **Attributi semplici**, descritti per mezzo di linee terminate da cerchi vuoti e da nomi;
- **Attributi identificatori delle entità**, usati come strumento per l'identificazione univoca delle occorrenze dell'entità e indicati come tutti gli altri attributi ma con il cerchio pieno per evidenziare il loro ruolo all'interno delle entità;
- **Attributi composti**, identificati da un nome caratterizzante la struttura a cui vengono collegati gli attributi componenti;
- **Attributi multivaleore**, formati da un insieme di valori ed indicati tramite due numeri (X,N) che esprimono la cardinalità dell'attributo (il minimo ed il massimo numero di valori che può avere);
- **Attributi opzionali**, indicati con una cardinalità minima pari a zero (essendo opzionali devono necessariamente poter contemplare il valore NULL), (0,X).

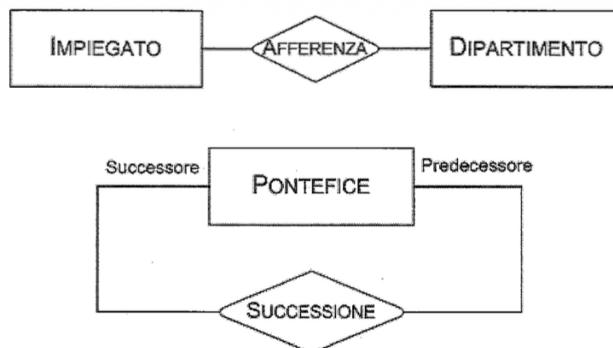


Si consideri che la realtà che si cerca di descrivere con una base di dati non è mai un'entità della base di dati stessa: se si cerca di descrivere uno zoo, non esisterà mai un'entità Zoo bensì l'insieme di entità e attributi costituirà la rappresentazione dello zoo.

Con il termine **associazione** si fa riferimento al legame concettuale fra due o più entità e, almeno da un punto di vista matematico, è possibile dire che ogni occorrenza di una associazione lega n occorrenze di entità. Pertanto, un'associazione \mathcal{R} tra n entità E_1, \dots, E_n è un sottoinsieme del prodotto cartesiano $E_1 \times \dots \times E_n$ dove ciascuna entità E_i partecipa all'associazione \mathcal{R} , così come le singole occorrenze di entità e_i partecipano alle occorrenze di associazioni $r_i = \langle e_1, \dots, e_n \rangle$. Con grado di un'associazione si indica il numero di entità che partecipano all'associazione stessa.



Le associazioni sono indicate, graficamente, da un rombo al cui centro è inserito il nome dell'associazione e da collegamenti verso le diverse entità che sono coinvolte nell'associazione:



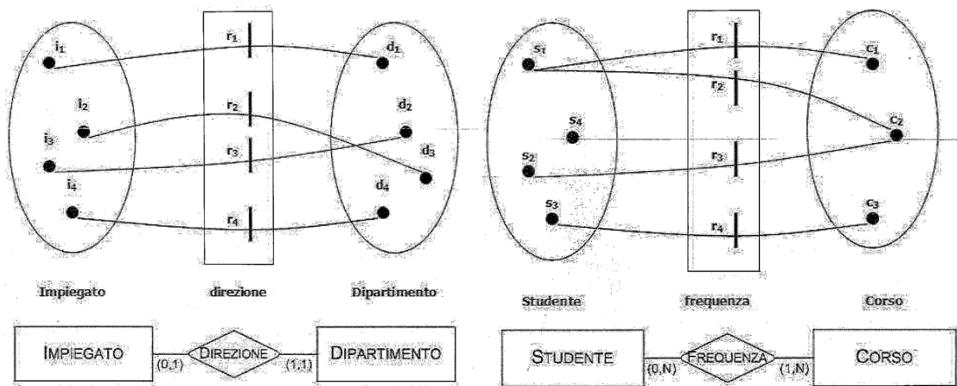
Quello appena riportato è un esempio particolare di associazione binaria, in cui si nota che la stessa entità PONTEFICE partecipa più volte allo stesso tipo di associazione; in queste condizioni, si dice che l'associazione è ricorsiva. Nelle associazioni ricorsive occorre dare un nome al legame,

detto **nome di ruolo** (**successore** e **predecessore** nell'esempio), per poter distinguere il ruolo assunto dall'entità nel legame associativo.

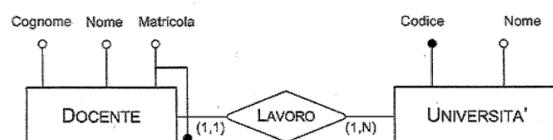
La cardinalità di un'associazione è una **coppia di valori**, associati ad ogni entità, che specificano il numero minimo e massimo delle occorrenze di associazioni a cui ciascuna occorrenza di entità può partecipare. Con riferimento alle cardinalità massime, si può definire anche il **rappporto di cardinalità**:

- **Uno ad uno**, con il quale si stabilisce che ad ogni occorrenza di un'entità corrisponde una e una sola occorrenza dell'altra entità;
- **Uno a molti**, con il quale si stabilisce che ad un'occorrenza di un'entità possono essere associate più occorrenze dell'altra entità;
- **Molti a molti**, con il quale si stabilisce che ad un'occorrenza di un'entità possono essere associate più occorrenze dell'altra entità e viceversa.

Esempi di associazioni uno ad uno e molti a molti sono, rispettivamente, le seguenti:



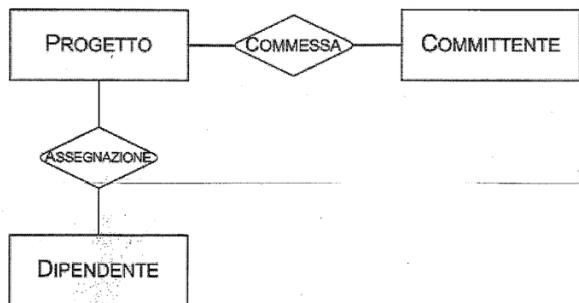
L'identificatore è il sottoinsieme di attributi che consente di identificare in modo univoco le occorrenze di un'entità e può essere costituito da attributi (uno o più) dell'entità stessa (in questo caso si parla di identificatore interno) o, in alcuni casi, facendo ricorso ad attributi di un'altra entità ad essa riferita attraverso la relativa associazione (in questo caso si parla di identificatore esterno). Si noti che è possibile introdurre un identificatore esterno solo se l'entità che comprende l'attributo (o l'insieme di attributi) facente parte della chiave partecipa con cardinalità uno ad uno alle associazioni i cui attributi contribuiscono alla chiave. Graficamente, l'identificatore esterno è individuato da una linea che attraversa l'attributo e il legame tra entità ed associazione. Ad esempio, Matricola è un identificatore esterno nell'esempio seguente:



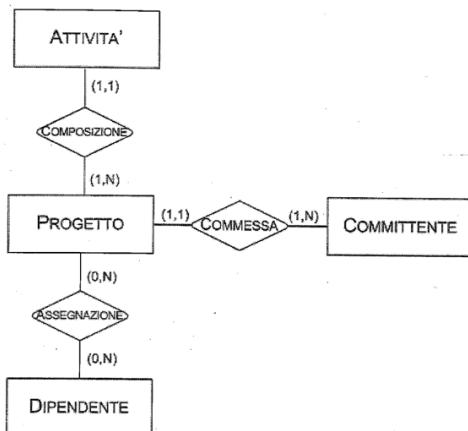
Si vuole realizzare una base di dati per una data società di consulenza che effettua lavori su commissione. La base di dati deve mantenere informazioni sui COMMITTENTI (aziende pubbliche o private), sui DIPENDENTI della società e sui PROGETTI commissionati (una commessa può riguardare un solo progetto), in lavorazione o già ultimati. Per i COMMITTENTI, identificati dalla partita I.V.A., si vuole memorizzare il nome, il titolare, il telefono, l'eventuale e-mail e l'indirizzo composto da città, provincia, via, numero civico e cap. Per i PROGETTI, identificati da un codice, si vuole memorizzare il nome, la descrizione, la tipologia, l'acronimo, il costo, le specifiche esecutive, la data di richiesta o di inizio, la data di consegna del lavoro ed il committente ed i dipendenti assegnati con il relativo ruolo (project-manager, analista programmatore, etc..). Un dipendente può essere assegnato a più progetti. Ogni progetto è poi composto da un serie di attività, identificate da un progressivo numerico e dal progetto a cui si riferisce, e, caratterizzate da un nome, una descrizione, una tipologia, una data di inizio, una data di fine, da informazioni relative alla terminazione, ed, infine, da un report tecnico. Per i DIPENDENTI, identificati da una matricola aziendale, memorizziamo il nome, il cognome, il numero o numeri di telefono, la data di nascita, il codice fiscale, l'eventuale e-mail, l'indirizzo, composto da città, provincia, via, numero civico e cap, e la qualifica professionale.

Per ricavare un modello ER a partire da una specifica (come quella appena riportata, ad esempio) è possibile usare, con grande comodità, un **approccio top down** mediante i seguenti passi:

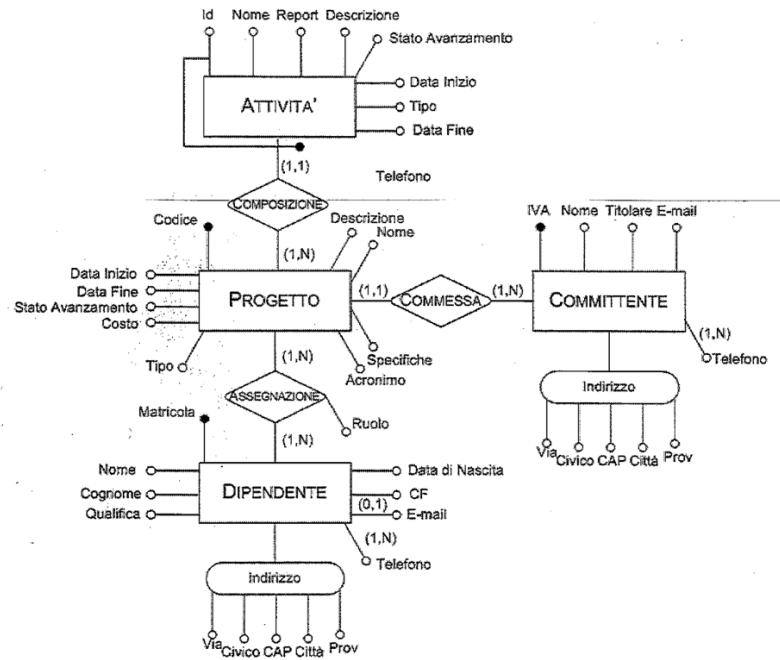
- **Individuazione**, in termini di entità e associazioni, dello schema portante della base di dati, ossia dello schema che racchiude i concetti fondamentali (definizione entità e associazioni);



- **Raffinamento ed estensione dello schema portante**, inserendo gradualmente in esso tutte le specifiche fino ad allora non ancora esaminate (definizione cardinalità delle associazioni);



- **Definizione delle proprietà relative alle varie entità e associazioni** (introduzione attributi).

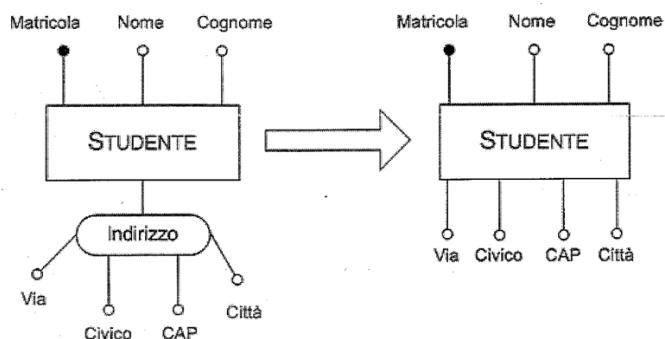


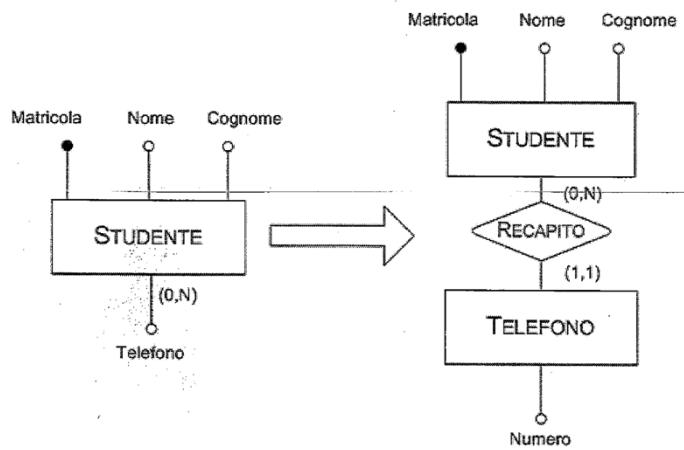
Il passaggio dal modello concettuale dei dati a quello logico consiste nel **costruire uno schema orientato al modello relazionale** ed in grado di rappresentare gli stessi concetti modellati dallo schema concettuale. La progettazione logica si articola in due fasi:

- **Trasformazione dello schema concettuale**, con lo scopo di semplificare alcuni costrutti del modello ER (come, ad esempio, gli attributi composti e multivalore) non direttamente traducibili in un modello logico relazionale;
- **Traduzione dello schema semplificato** in un insieme di relazioni e di vincoli, come stabilito dal modello relazionale.

Il risultato della progettazione logica costituisce, poi, il **punto di partenza per la progettazione fisica**, che ha lo scopo di rendere quanto più efficiente l'esecuzione delle operazioni previste sulla base di dati.

Gli attributi composti e multivalore non sono, come già anticipato, **immediatamente traducibili nel modello logico**; un **attributo composto** può essere semplificato sostituendolo con **tanti altri attributi semplici** quanti sono gli attributi componenti, mentre un **attributo multivale** deve essere **eliminato introducendo una nuova entità, corrispondente al concetto rappresentato dall'attributo multivale**, legata all'entità originale da un'associazione con rapporto di cardinalità uno a molti, che conserverà lo stesso nome e gli stessi attributi ad eccezione dell'attributo multivale.





Nella fase di traduzione, **a partire dallo schema ER semplificato, si costruisce lo schema logico relazionale equivalente**, ovvero l’insieme di relazioni e vincoli in grado di rappresentare gli stessi concetti descritti dallo schema concettuale. Si analizzi il processo di traduzione a compartimenti stagni:

- **Traduzione di entità**

Un’entità dello schema concettuale si traduce in una relazione dello schema logico avente lo stesso nome (ma al plurale, convenzionalmente) e gli stessi attributi dell’entità, con chiave primaria l’identificatore dell’entità stessa. A volte è opportuno scegliere come identificatore di un’entità un attributo non inizialmente previsto dalle specifiche di progetto (come un codice numerico, un contatore o un codice alfanumerico che rende facile l’accesso ai dati); inoltre, si noti come gli attributi della relazione corrispondenti ad attributi opzionali sono quelli che possono assumere valori nulli.

- **Traduzione di associazioni molti a molti**

Un’associazione molti a molti dello schema concettuale semplificato si traduce in una relazione dello schema logico avente lo stesso nome (ma al plurale, convenzionalmente) dell’associazione e per attributi gli attributi dell’associazione e gli identificatori delle entità coinvolte; l’insieme di tali identificatori costituisce la chiave primaria della relazione ed ognuno di essi ha un vincolo di integrità referenziale con il corrispondente attributo dell’entità da cui discende.

- **Traduzione di associazioni uno a molti**

Un’associazione uno a molti dello schema concettuale semplificato si traduce in una relazione dello schema logico avente lo stesso nome (ma al plurale, convenzionalmente) dell’associazione e per attributi gli attributi dell’associazione e gli identificatori delle entità coinvolte. L’identificatore dell’entità dal lato uno costituisce la chiave primaria della relazione ed ogni identificatore ha un vincolo di integrità referenziale con il corrispondente attributo dell’entità da cui discende.

Alternativamente, un’associazione uno a molti dello schema concettuale si traduce aggiungendo alla relazione che traduce l’entità dal lato uno gli attributi dell’associazione e l’identificatore dell’entità dal lato molti, eventualmente ridenominato, ed esiste un vincolo di integrità referenziale tra questo attributo ed il corrispondente attributo dell’entità dal lato molti.

Nel caso in cui la partecipazione all’associazione dell’entità dal lato uno sia opzionale, gli attributi introdotti nella relazione che la traduce possono assumere valori nulli; in questo caso, può essere

preferibile usare il primo metodo di traduzione introdotto. Nel caso in cui, invece, un'entità dell'associazione possiede un identificatore esterno, si hanno due possibili soluzioni:

- Usando la prima regola di traduzione, l'identificatore dell'entità dal lato molti, aggiunto alla relazione relativa all'associazione, entra a far parte della chiave primaria insieme all'identificatore dell'entità dal lato uno;
- Usando la seconda regola di traduzione, l'identificatore dell'entità dal lato molti, aggiunto all'entità dal lato uno, entra a far parte della chiave primaria della relativa relazione.

- **Traduzione di associazioni uno ad uno**

Un'associazione uno ad uno dello schema concettuale si traduce in una relazione dello schema logico avente lo stesso nome (ma al plurale, convenzionalmente) dell'associazione e per attributi gli attributi dell'associazione e gli identificatori delle entità coinvolte; in particolare, l'identificatore di una delle due entità costituisce la chiave primaria della relazione, mentre l'altro identificatore deve soddisfare il vincolo di unicità. Infine, ogni identificatore ha un vincolo di integrità referenziale con il corrispondente attributo dell'entità da cui discende.

Alternativamente, un'associazione uno ad uno dello schema concettuale si traduce aggiungendo alla relazione che traduce una delle due entità gli attributi dell'associazione e l'identificatore dell'altra entità, eventualmente ridenominato, ed esiste un vincolo di unicità sul nuovo attributo aggiunto ed un vincolo di integrità referenziale tra quest'ultimo e il corrispondente attributo dell'altra entità.

Per la traduzione dell'associazione uno ad uno usando la prima delle due regole introdotte, si potrebbe scegliere di inserire in modo ridondante due chiavi esterne in entrambe le relazioni. Inoltre, si noti che un'ulteriore strategia potrebbe essere quella di accorpate tutto in una singola entità, inserendo in quest'ultima gli attributi dell'associazione e quelli dell'altra entità; tuttavia, in questo modo si ottiene un appesantimento delle tuple della relazione, che potrebbe inficiare l'efficienza delle operazioni.

- **Traduzione di associazioni ricorsive**

Le regole di traduzione delle associazioni ricorsive sono le stesse usate nella traduzione delle associazioni binarie.

- **Traduzione di associazioni ternarie**

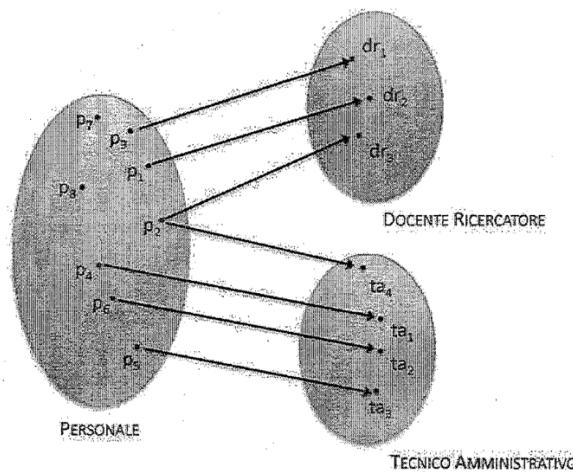
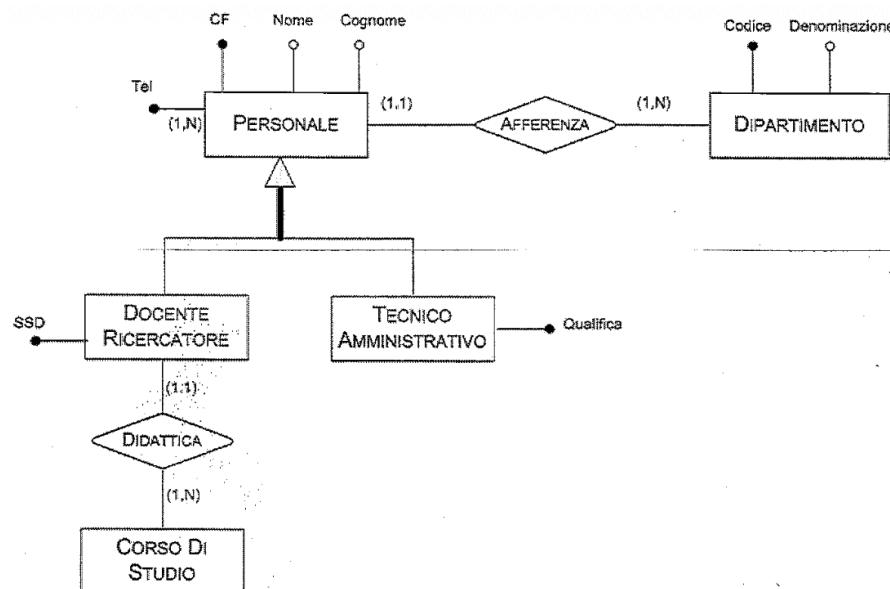
Un'associazione ternaria dello schema concettuale semplificato si traduce in una relazione dello schema logico avente lo stesso nome (ma al plurale, convenzionalmente) dell'associazione e per attributi gli attributi dell'associazione e gli identificatori di tutte le entità coinvolte. L'insieme degli identificatori, che partecipano all'associazione con cardinalità massima N , costituisce la chiave primaria della relazione ed ognuno di essi ha un vincolo di integrità referenziale con il corrispondente attributo dell'entità da cui discende.

IL MODELLO ER AVANZATO

Il **modello ER formulato da Chen** nel 1976 costituisce tutt'ora un **punto di riferimento per la progettazione concettuale di una base di dati**; tuttavia, nel tempo si è reso necessario gestire una mole di dati via via sempre crescente ed è nata l'**esigenza di un modello che potesse riflettere in modo più efficace le proprietà semantiche dei dati stessi**. Nel classico modello ER sono presenti dei costrutti che non riescono a pieno a gestire livelli di astrazione capaci di governare la

complessità delle realtà che si vanno a rappresentare; per questo motivo, **diversi autori hanno proposto versioni estese o migliorate del modello ER** in cui sono stati introdotti i **concetti di generalizzazione/specializzazione e di superclasse/sottoclasse come strumenti di astrazione.**

Le entità tra loro possono avere in comune caratteristiche di tipo funzionali o strutturali, che ne consentono una **classificazione strutturata in una gerarchia di superclassi e sottoclassi.** Una **superclasse** è un'entità che include e accorpa uno o più sottoinsiemi distinti di singole entità, dette anche **sottoclassi;** in particolare, lungo la gerarchia, le sottoclassi **specializzano o dettagliano alcuni aspetti della superclasse.** Uno degli aspetti più importanti associati alle superclassi e alle sottoclassi è quello di **ereditarietà degli attributi e delle associazioni:** tutti gli attributi e le associazioni di una superclasse sono propagati nelle sottoclassi che specializzano la superclasse, in modo simile a come è strutturata l'ereditarietà nella **Programmazione Orientata agli Oggetti.**



L'estensione del modello ER in questione introduce nella progettazione concettuale il costrutto di **specializzazione**, con il quale è possibile definire un insieme di sottoclassi di una data entità E_0 . Con esso, **una o più entità** (dette sottoclassi o entità figlie) sono messe in relazione con l'entità E_0 (detta **superclasse** o entità padre); E_0 è detta **generalizzazione** delle entità E_1, \dots, E_n , le quali sono **specializzazioni della prima**. Si noti che **una stessa entità può essere coinvolta in più specializzazioni e che è possibile avere generalizzazioni con una sola entità figlia**, alla quale è dato il nome di **sottoinsieme**.

Considerate n sottoclassi di una classe E_0 , $\{E_1, \dots, E_n\}$, formalizzato come $E_i \subseteq E_0 \forall i \in [1, n]$, due sono i tipi di vincoli che possono essere specificati per una gerarchia:

- **Vincolo di copertura**, con il quale imporre quante delle occorrenze della superclasse si ritrovano tra le occorrenze delle sottoclassi. La generalizzazione si dice totale se ogni occorrenza del padre è occorrenza di almeno una delle entità figlie (non esiste un'occorrenza del padre che non sia occorrenza di un'entità figlia), in caso contrario la generalizzazione è detta parziale;

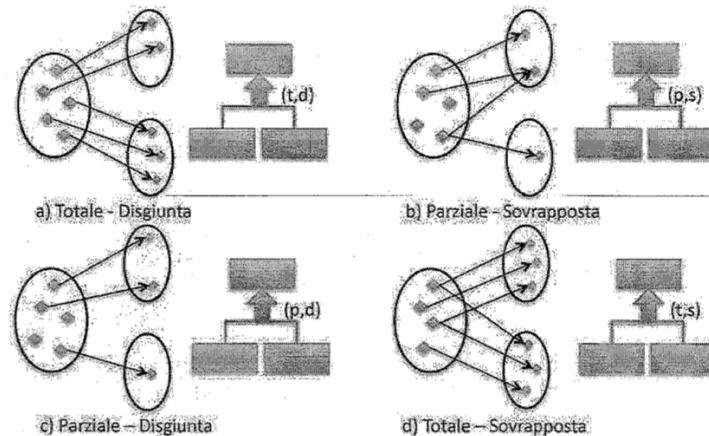
$$\bigcup_{i=1}^n E_i = E_0$$

- **Vincolo di disgiunzione**, con il quale imporre come si distribuiscono le occorrenze delle sottoclassi. Una generalizzazione è disgiunta (o esclusiva) se ogni occorrenza dell'entità padre è occorrenza al più di una delle entità figlie, altrimenti è sovrapposta.

$$E_i \cap E_j = \emptyset, \forall i \neq j$$

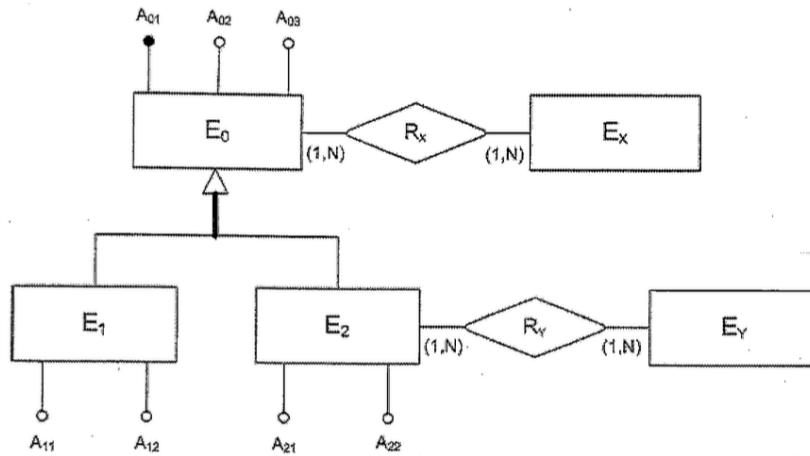
Segue che, **combinando le possibili tipologie di generalizzazione appena evidenziate**, si possono ottenere le seguenti configurazioni:

- Totale – disgiunta;
- Parziale – disgiunta;
- Totale – sovrapposta;
- Parziale – sovrapposta.

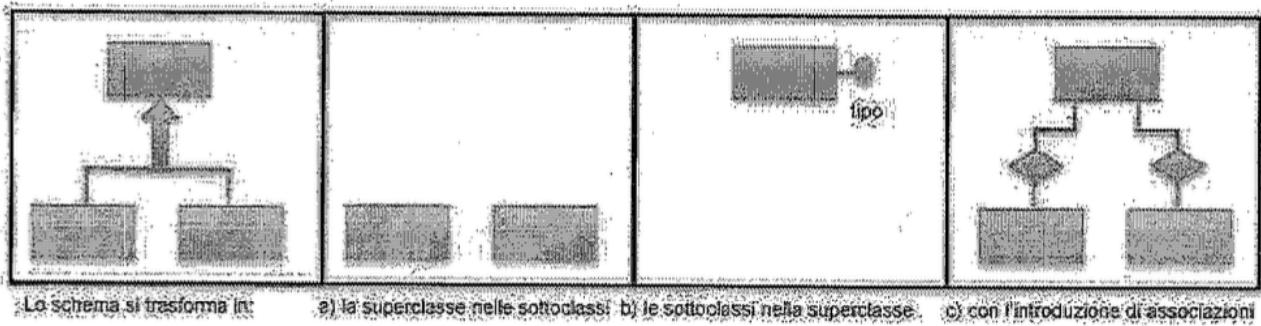


In presenza di una **specializzazione con un'unica entità figlia** si parla propriamente di **inclusione**.

Le gerarchie di generalizzazione/specializzazione non possono essere rappresentate nel modello logico relazionale, dal momento in cui, come noto, non è previsto un costrutto analogo. Al fine di tradurre il modello concettuale in un modello logico relazionale, è necessaria una loro preventiva trasformazione in costrutti più semplici e previsti dal modello ER originario. Si consideri, ad esempio, lo schema ER seguente, che modella una **generalizzazione tra una superclasse E_0 e le sottoclassi E_1 e E_2 , ognuna delle quali, oltre alle proprietà ereditate dal padre, possiede altre proprietà specifiche**:

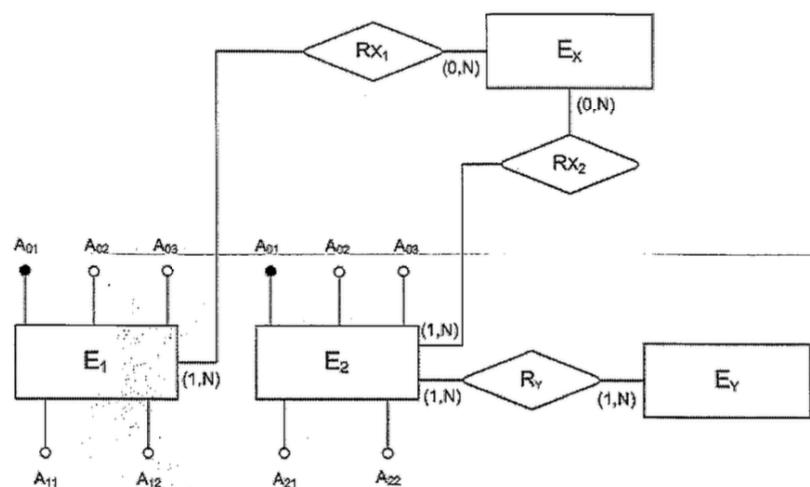


Esistono tre possibili modalità di trasformazione della generalizzazione in schemi traducibili nel modello logico:



- **Accorpamento della superclasse nelle sottoclassi**

È prevista l'eliminazione dell'entità padre; per la proprietà di ereditarietà, i suoi attributi, il suo identificatore e le relazioni a cui tale entità partecipa vengono ereditate dalle entità figlie. Con riferimento allo schema ER precedente, l'applicazione di tale soluzione produce lo schema:



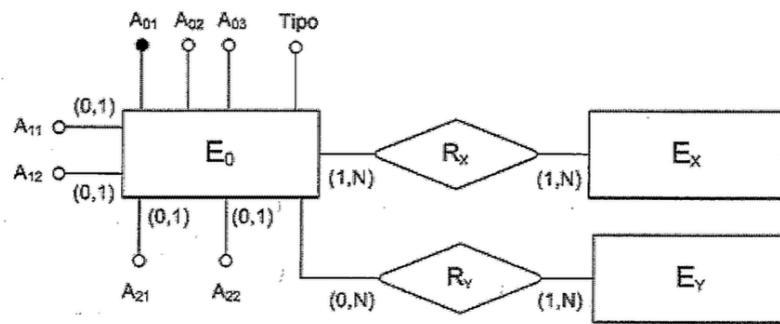
L'entità padre E_0 è stata rimossa, aggiungendo ad entrambe le entità figlie E_1 e E_2 il suo identificatore A_{01} e gli attributi A_{02} e A_{03} ; inoltre, le associazioni R_{x_1} e R_{x_2} rappresentano le restrizioni della associazione R_x sulle occorrenze delle entità E_1 e E_2 , rispettivamente. Si noti che la partecipazione

dell'entità E_x a tali associazioni è opzionale, in quanto se un'occorrenza di E_x era in relazione con un'occorrenza di E_0 , tale occorrenza potrebbe essere occorrenza di una sola delle due entità figlie.

Questa tipologia di trasformazione è **consigliabile quando ci si trova di fronte a generalizzazioni con vincoli totale-disgiunta**; infatti, la soluzione non è applicabile se la generalizzazione non è totale (se ci sono occorrenze della superclasse non presenti nella sottoclasse, non è possibile eliminare la superclasse) e, allo stesso tempo, se c'è sovrapposizione, alcune occorrenze di entità possono essere duplicate, introducendo risonanza tra i dati.

- **Accorpamento delle sottoclassi nella superclasse**

Tale soluzione prevede **l'eliminazione delle sottoclassi**, con la conseguente **aggiunta alla superclasse di tutti gli attributi e tutte le associazioni cui le sottoclassi partecipano**. Alla superclasse, inoltre, viene aggiunto **un attributo per distinguere il tipo di un'occorrenza**, ovvero per distinguere **a quale delle sottoclassi tale occorrenza apparterrebbe**. Con riferimento allo schema ER precedente, l'applicazione di tale soluzione produce lo schema:

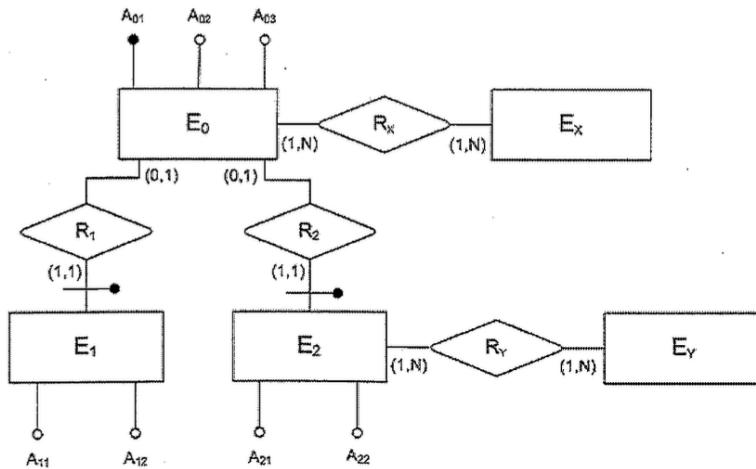


Gli attributi A_{11} e A_{12} di E_1 e gli attributi A_{21} e A_{22} di E_2 sono stati aggiunti all'entità padre E_0 , insieme all'attributo $Tipo$. Si noti, inoltre, che tali attributi possono ammettere valori nulli, essendo le occorrenze di E_1 e E_2 sottoinsiemi delle occorrenze di E_0 ; infatti, gli attributi A_{11} e A_{12} non sono significativi per le occorrenze di E_0 che sono istanze di E_2 e viceversa. Infine, è stata aggiunta ad E_0 la partecipazione all'associazione R_y , la cui cardinalità minima diventa 0, dal momento in cui le occorrenze di E_0 che siano occorrenze di E_1 non partecipano a tale associazione.

La soluzione appena proposta è particolarmente **consigliabile quando i vincoli di gerarchia sono totale-sovrapposta**; infatti, si evitano ridondanze dovute a sovrapposizioni, anche se si ha una presenza di valori nulli per gli attributi specifici delle classi. Si noti che, in linea di principio, tale soluzione può andare bene anche nel caso di gerarchie parziali; in particolare, è consigliabile quando non c'è differenza tra le operazioni che accedono agli attributi della superclasse e quelle operanti sugli attributi delle sottoclassi.

- **Sostituzione della generalizzazione con tante associazioni quante sono le sottoclassi**

Questa soluzione prevede **la sostituzione della generalizzazione con associazioni uno ad uno tra sottoclassi e superclasse**. Le sottoclassi, che avranno partecipazione obbligatoria a tali relazioni, saranno identificate esternamente dalla superclasse. Con riferimento allo schema ER precedente, l'applicazione di tale soluzione produce lo schema:

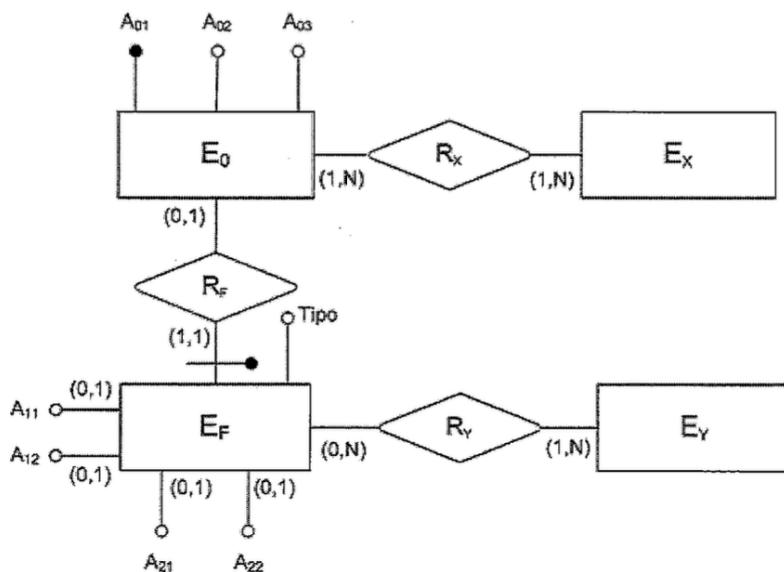


Le entità E_0 , E_1 e E_2 sono rimaste inalterate ma la generalizzazione è stata sostituita dalle due relazioni R_1 e R_2 . Si noti che la partecipazione di E_0 a tali relazioni è opzionale, in quanto un'occorrenza dell'entità padre non è occasione di tutte le figlie.

La soluzione proposta, in linea di principio, è **quella più intuitiva e sempre applicabile**. Si noti, tuttavia, che è **consigliabile laddove ci sia necessità di mantenere separate le superclassi e le sottoclassi, ovvero quando i vincoli sono di tipo parziale-disgiunta** e quando le operazioni sono sia attributi della superclasse che attributi delle sottoclassi.

- **Soluzione mista**

Ci sono anche soluzioni miste che si basano su una combinazione delle tre precedenti; di seguito è proposto un esempio che prevede la sostituzione delle sottoclassi con un'unica classe in associazione uno ad uno con la superclasse. La sottoclasse, che avrà partecipazione obbligatoria a tale relazione, sarà identificata esternamente dalla superclasse; l'unica entità rimasta, allora, avrà tutti gli attributi e tutte le associazioni cui le varie sottoclassi partecipano, con in più un attributo per distinguere il tipo di un'occorrenza, ovvero per distinguere quale delle sottoclassi tale occorrenza appartiene. Con riferimento allo schema ER precedente, l'applicazione di tale soluzione produce lo schema:



L'entità E_0 rimane inalterata, mentre E_1 e E_2 sono sostituite da un'unica entità E_F ; gli attributi A_{11} e A_{12} di E_1 e gli attributi A_{21} e A_{22} di E_2 sono stati aggiunti all'entità E_F , insieme all'attributo $Tipo$. La generalizzazione, inoltre, viene sostituita dalla relazione R_F ; si noti, che la partecipazione di E_0 all'associazione R_F è opzionale, in quanto un'occorrenza di E_0 non è detto sia occorrenza di E_F . Infine, è stata aggiunta a E_F la partecipazione all'associazione R_y , la cui cardinalità minima diventa 0, dal momento in cui le istanze di E_F che siano istanze di E_1 non partecipano a tale associazione.

Tale soluzione può essere particolarmente **consigliabile per gerarchie parziali e sovrapposte**.

PROGETTAZIONE FISICA E IL LIVELLO APPLICATIVO

ORACLE DBMS

I DBMS sono sistemi software evoluti che, come già ampiamente descritto, permettono la gestione efficace ed efficiente di basi di dati, ovvero di collezioni di dati di grosse dimensioni, persistenti e condivise. I DBMS si sono rapidamente diffusi in tutti i settori applicativi e, ad oggi, rappresentano una componente tecnologica fondamentale ed imprescindibile per una qualsiasi organizzazione, almeno per ciò che riguarda la memorizzazione e la gestione delle informazioni. Tendenzialmente, le scelte delle aziende ricadono su DBMS commerciali gestiti da aziende private piuttosto che da progetti open source a causa della necessità di avere soluzioni con requisiti stringenti in termini di affidabilità, sicurezza, concorrenza ed efficienza. Uno dei colossi nel campo dei DBMS commerciali è sicuramente l'**ORACLE Database**, che conta milioni di installazioni in tutto il mondo.

Come già anticipato, un generico DBMS presenta un'architettura a livelli in cui si riconoscono un livello esterno (viste sui dati a cui fanno riferimento gli utenti e le applicazioni), un livello logico (composto dall'insieme delle relazioni e dei vincoli su di esse) e un livello fisico (costituito da un insieme di file in memoria di massa su cui risiedono fisicamente le informazioni). Un DBMS deve fornire all'esterno una serie di funzionalità per:

- **Creare** lo schema di una base di dati;
- **Popolare** una base di dati;
- **Aggiornare** il contenuto di una base di dati;
- **Interrogare** una base di dati;
- **Garantire l'integrità** delle informazioni presenti nella base di dati in accordo con i vincoli definiti su di essa; a
- **Garantire l'accesso** concorrente di utenti/applicazioni alla base di dati, mediante l'impiego di meccanismi di locking in grado di serializzare l'accesso concorrente delle transazioni sui dati condivisi;
- **Garantire la sicurezza** sulla base di dati, mediante definizione di politiche di accesso ai dati;
- **Garantire l'affidabilità** della base di dati, tramite salvataggio periodico del contenuto (dump), di appositi file di log delle operazioni e di meccanismi di recovery per l'eventuale ricostruzione di dati persi.

Proprio per garantire queste funzionalità ed implementare al meglio l'architettura descritta, un DBMS deve prevedere una serie di componenti, che possono essere schematizzati come segue:

- **Gestore degli Accessi**, è il modulo di un DBMS che effettua il controllo degli accessi alla base di dati, garantendo che solo gli utenti e le applicazioni autorizzati abbiano l'accesso alle informazioni della base di dati e che le operazioni che questi possano eseguire siano compatibili e coerenti con i loro privilegi/ruoli;
- **Gestore delle Query** (o Ottimizzatore), è il componente che si occupa della gestione delle richieste utente in termini di operazioni DDL e DML. In particolare, traduce i comandi DDL e DML in un formato interno al DBMS (piano di accesso ai dati) trasformando, se possibile, la richiesta utente in una equivalente ma più efficiente;
- **Gestore della Memoria** (o Buffer Manager), è il componente che si occupa del trasferimento effettivo dei dati (paginati) dalla memoria di massa alla memoria centrale e viceversa, gestendo solitamente grosse aree di memoria centrale assegnate al DBMS e spesso condivise fra varie applicazioni;

- **Gestore File** (o **Gestore dei Metodi di Accesso ai Dati**), è il componente che si occupa di eseguire i piani di accesso ai dati fisici sui file relativi alla base di dati in memoria di massa definiti dal gestore delle query;
- **Gestore della Concorrenza**, è il modulo che ha l'obiettivo di gestire l'accesso di transizioni concorrenti a risorse condivise della base di dati, utilizzando particolari meccanismi di locking delle risorse per serializzare le transizioni in questione e, così, prevenendo queste ultime da anomalie nell'accesso alle informazioni;
- **Gestore dell'Integrità**, è il modulo che ha il compito di verificare che i vincoli di integrità sulla base di dati siano sempre rispettati;
- **Gestore dell'Affidabilità**, è il modulo che si occupa del salvataggio periodico dei file di log e dell'avvio delle procedure di ripristino della base di dati a valle di malfunzionamenti.

L'architettura del DBMS ORACLE si basa sul classico **paradigma di comunicazione “client-server”**, in cui sono presenti **uno o più processi client che richiedono un servizio e un processo server che eroga tale servizio**. Nel caso di un DBSM, il servizio offerto è quello di **accesso e gestione delle informazioni presenti all'interno della base di dati**, mentre il linguaggio di formulazione delle richieste da parte dei client è SQL. Dal lato client, quindi, sono disponibili per gli utilizzatori finali del database le funzionalità di controllo, la formulazione di richieste via SQL, l'immissione dei dati e la visualizzazione dei risultati; dal lato server, invece, si trova **tutta la logica di gestione della base di dati e il “data processing” vero e proprio** in termini di esecuzione di istruzioni SQL, gestione degli utenti e risorse, e gestione dello spazio sul disco.

In particolare, dal lato server, l'architettura del DBMS ORACLE si compone di tre parti fondamentali:

- **Una o più istanze ORACLE;**
- **Uno o più processi listener;**
- **Uno o più processi Oracle database, ognuno legato ad una data istanza ORACLE.**

Un'istanza ORACLE è costituita da **un insieme di processi e di strutture dati allocate in memoria che realizzano il meccanismo di accesso ai dati di un database ORACLE e ne permettono la gestione**. L'istanza, per poter accedere ai dati contenuti nel database, **dove essere attiva e risiedere**, insieme alle proprie strutture dati, **in una specifica area di memoria denominata SGA** (System Global Area); **nella SGA si trovano particolari strutture dati**, tra cui:

- **Shared pool**, area usata per memorizzare gli statement SQL recentemente utilizzati, consentendo ad un'eventuale istanza di reperire informazioni in maniera più veloce;
- **Database buffer**, area usata per memorizzare e gestire i blocchi dati oggetto delle elaborazioni, i quali sono letti e/o scritti nei data file;
 - ORACLE gestisce lo spazio disponibile in questo tipo di buffer usando un algoritmo LRU (Last Recently Used) per il quale, quando è necessario spazio libero, i blocchi utilizzati meno di recente vengono scritti sui data file;
 - La dimensione del buffer ha il maggiore impatto sulle prestazioni globali del sistema di base di dati;
- **Redo Log Buffer**, area contenente le informazioni relative ai blocchi dati modificati in seguito ad operazioni sulla base di dati e componente fondamentale per la ricostruzione della base di dati a valle di malfunzionamenti;
- **Data and Library caches**, aree di cache della shared pool utilizzate per contenere dati in transito da e verso la base di dati (data) e le istruzioni SQL e PL/SQL in esecuzione (library).

I processi di un'istanza ORACLE eseguono le funzionalità tipiche di un DBMS ma il loro numero e il loro tipo dipendono dalla particolare configurazione dell'istanza; tuttavia, alcuni sono sempre presenti e sono elencati di seguito:

- **Dispatcher process (D000), Query processor (QP) e Server process**, rispettivamente, responsabile dello scheduling delle richieste utente, il processore delle query (decodifica e determina il piano di accesso ai dati per ogni query) e responsabile di processare le richieste utente in un'area di memoria apposita, la Program Global Area (PGA), che contiene le informazioni sulla sessione utente corrente e che verifica i loro privilegi, stabilendo insieme al QP i piani di accesso;
 - È chiaro come questi processi implementino le **funzionalità tipiche di un Gestore delle Query e di un Gestore degli Accessi**;
- **Database Writer (DBW0), Process monitor (PMON) e User process (UP)**, rispettivamente, responsabile della scrittura dei blocchi di dati dal database buffer (in memoria) nei datafile (sul disco), responsabile del rilascio delle risorse allocate da un eventuale processo utente fallito (garbage collector) e responsabile del prelievo dei blocchi dati richiesti dai data file, in corrispondenza delle richieste utente, e del trasferimento nel buffer qualora non siano già presenti nella SGA;
 - È chiaro come questi processi implementino le **funzionalità tipiche del Gestore dei File**;
- **Log writer (LGWR), Checkpoint process (CKPT), Recoverer process (REC0) e Archiver process (ARC0)**, rispettivamente, responsabile della scrittura, degli eventi che si susseguono in una sessione di lavoro, dal Redo log buffer (in memoria) al log file (su disco), responsabile dell'aggiornamento dello stato del database, per quanto concerne i datafile, ogni volta che viene memorizzato permanentemente un record nel database (garantisce la sincronizzazione dei dati), responsabile della risoluzione delle transazioni fallite in ambiente distribuito, e responsabile del salvataggio automatico delle copie dei redo log in un'area di memoria stabile (Archived Log Files) specificata dall'amministratore del database;
 - È chiaro come questi processi implementino le **funzionalità tipiche di un Gestore dell'Affidabilità**;
- **System monitor (SMON)**, è il processo responsabile del controllo della consistenza e dell'integrità del database, se necessario inizia un recovery del database, quando attivo;
 - È chiaro come questo processo implementi le **funzionalità tipiche del Gestore dell'Integrità**;
- **Lock process (LMS)**, responsabile della gestione dei lock sulle transazioni in ambiente distribuito;
 - È chiaro come questo processo implementi le **funzionalità tipiche del Gestore della Concorrenza**.

I processi **D000, PMON, SMON, ARC0, REC0, LGWR e DWR** sono detti anche **processi di background**, dal momento in cui la loro esecuzione non prevede alcuna interazione con i processi utente.

Un **processo listner** è un **processo asincrono** che si occupa di gestire la comunicazione fra i processi client e un'istanza ORACLE, è **caratterizzato** da un nome, dal database Oracle che serve e da una porta di ascolto. È possibile **definire un processo listner per ogni Oracle database** o, al massimo, **più listner sullo stesso database** ma **non è possibile associare ad un listner più database**. La configurazione dei listner attivi su una base di dati ORACLE è presente un particolare file chiamato LISTNER.ORA.

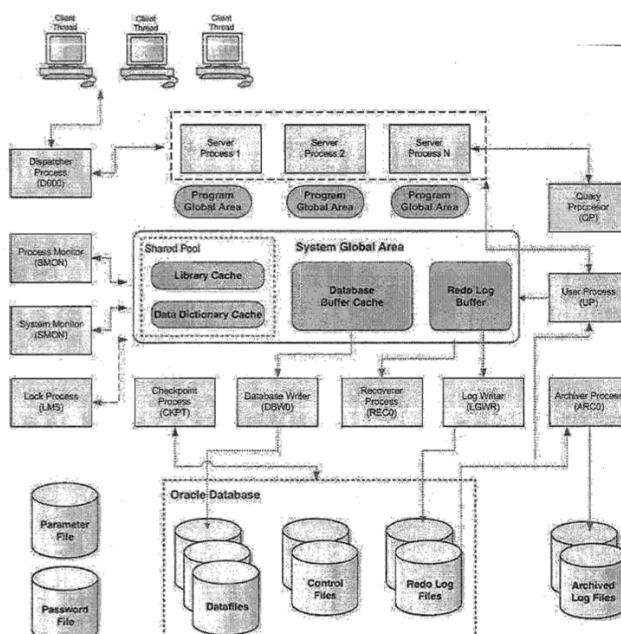
Un Oracle database è una **collezione di dati che è trattata come una singola unità**, dotata di struttura logica e fisica. La struttura logica si pone ad un livello superiore rispetto a quella fisica, potendo esplicitare in essa le relazioni componenti la base di dati ed i relativi vincoli, mentre la struttura fisica contiene effettivamente tutti i dati del database; in generale, un Oracle database è composto da tre tipi di file:

- **Data file**, contengono i dati attuali della base di dati, le informazioni sugli indici associati alle tabelle ed il catalogo ed è dotato delle seguenti caratteristiche:
 - Un data file può essere associato con un solo database;
 - I data file possono avere fissate automaticamente spazi di memoria, nel momento in cui il database supera lo spazio consentito;
 - Uno o più data file costituiscono un'unità logica di database, chiamata tablespace;
- **Redo Log file**, contengono informazioni, scritte in un record, per l'eventuale recovery della base di dati in caso di fallimento di qualche operazione;
- **Control file**, contengono le indicazioni per l'uso della base di dati e alcune informazioni specifiche dell'istanza, come nome del database, percorso file di log, ...).

Un Oracle database, in realtà, **utilizza anche altri file che non fanno parte del database**, che possono eventualmente **contenere i parametri caratteristici** (per il quale sono chiamati parameter file) riguardo un'istanza o la password degli utenti (per il quale sono chiamati password file).

Tutte le impostazioni che determinano il funzionamento dell'ambiente runtime dell'Oracle database sono contenute nel file **INIT.ORA**, il quale viene letto all'avvio di un'istanza ORACLE e in cui si trovano informazioni relative alla posizione su file system di tutti i file di sistema, le informazioni relative al dimensionamento della SGA e quelle relative alle funzionalità abilitate. Un Oracle database, poi, viene univocamente determinato, in ambito distribuito, **da un nome o SID** (System ID), dall'indirizzo IP o nome a dominio dell'host che lo ospita, e dalla porta di accesso su cui è in ascolto il processo **listner**. I parametri appena enunciati, che servono ad un'applicazione client per connettersi ad un database, possono essere specificati in un apposito file della macchina client, chiamato **TSNAME.ORA**.

Riassumendo, nella figura seguente è riportata la **schematizzazione completa dell'architettura di ORACLE**:



L'architettura ORACLE si compone di **uno o più processi utente**, che rappresentano gli utilizzatori finali della base di dati, e **di un processo server**, il cui compito è quello di **accettare richieste da parte dei processi utente e smistarle all'istanza**; inoltre, quest'architettura Client-Server di un DBMS ORACLE offre la **possibilità di effettuare connessioni dirette ad un database remoto**.

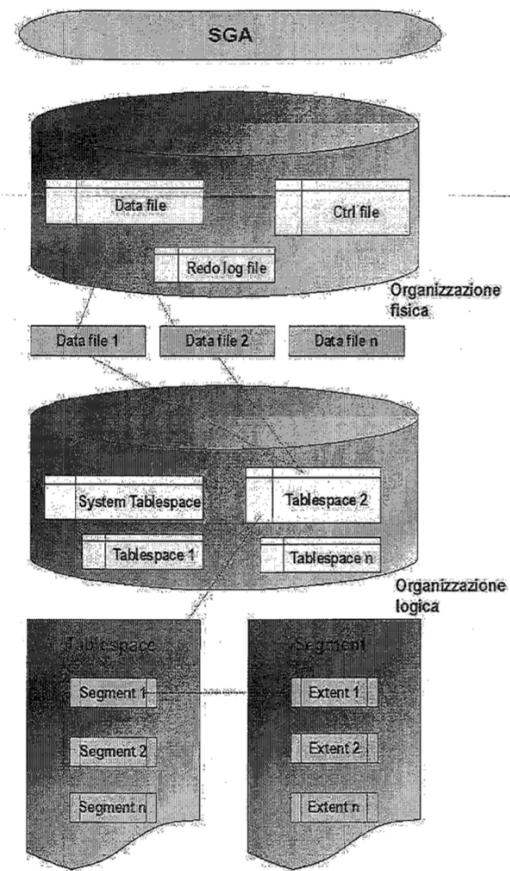
Di seguito è riportato l'intero processo di esecuzione di un'istruzione SQL da parte del server ORACLE. Si assuma che **un'utente**, tramite un'apposita applicazione, **esegua un'istruzione di update nella tabella T**, in modo che **una o più tuple ne siano interessate**; l'istruzione viene passata **dal dispatcher al server process dedicato, controllando se l'istruzione è già contenuta nella library cache** (in modo che le corrispondenti informazioni possano essere riutilizzate) tramite un albero di decodifica. Se **l'istruzione non è presente nella library cache**, viene decodificata e **verificata**, usando i dati della cache relativi al dizionario dei dati, poi viene generato un piano di esecuzione della query dal processore delle query, il quale viene memorizzato nella library cache insieme all'albero di decodifica. Per gli oggetti interessati dall'istruzione, viene controllato se i corrispondenti blocchi di dati esistono nel buffer del database, in caso negativo il processo user legge i blocchi di dati dai file fisici al buffer del database; se non c'è abbastanza spazio nel buffer, i blocchi di altri oggetti utilizzati meno recentemente vengono scritti su disco dal processo DBWR, che poi procede con la scrittura di un'immagine delle tuple sui segmenti di rollback prima che i blocchi vengano modificati. Mentre il buffer redo-log viene riempito a causa delle modifiche ai blocchi, il processo LGWR scrive sui file di redo-log le voci presenti nel redo-log buffer; dopo che **le tuple sono state modificate nel buffer del database, le modifiche possono essere confermate dall'utente tramite comando di commit**. Finché l'utente non invia il comando di commit, **le modifiche possono essere annullate utilizzando il comando di rollback**, sovrascrivendo eventualmente i blocchi modificati nel buffer del database con i blocchi originali memorizzati nei segmenti di rollback; tuttavia, **se l'utente invia un commit, lo spazio allocato per i blocchi nel segmento di rollback può essere liberato e reso disponibile per altre transazioni e i blocchi modificati nel buffer del database possono essere sbloccati**, in modo che altri utenti possano leggerli e modificarli. Infine, **la fine della transazione** (più precisamente, il commit) viene registrata nei file redo-log; i blocchi modificati vengono scritti su disco solo se si rende necessario altro spazio per altre transazioni.

Come già anticipato, **per ciò che riguarda la memorizzazione dei dati, nell'architettura di un database Oracle si distingue tra la struttura logica e quella fisica della base di dati**. Come ampiamente descritto, le strutture fisiche sono determinate dai file fisici del sistema operativo che costituiscono la base di dati, dove oggetti come le tabelle possono essere memorizzati. In particolare, **le strutture logiche di una base di dati ORACLE includono le seguenti componenti, elencate in ordine gerarchico**:

- **Database**, costituito da una o più partizioni logiche chiamate tablespace;
- **Tablespace**, una suddivisione logica del database atta a contenere gli oggetti della base di dati (tutti gli oggetti del database sono logicamente memorizzati in uno o più tablespace);
- **Segmenti**, porzione di tablespace alla quale viene associato un oggetto di un database in fase di allocazione;
 - Per ogni tabella esiste un segmento di tabella e per gli indici si utilizzano i segmenti di indice;
 - Un segmento non può essere collocato tra due tablespace;
 - I segmenti di rollback non contengono oggetti del database ma un'immagine di com'erano i dati prima di modifiche relative a transazioni che non hanno ancora fatto commit, in modo da ripristinare il più vicino stato stabile in caso di abort o eventi esplicativi di rollback;

- **Extent**, la più piccola unità logica di memorizzazione che può essere allocata per un oggetto di database e consiste in una sequenza contigua di blocchi di dati, ognuno dei quali corrisponde ad un numero di byte (multiplo della capacità del blocco usata dal sistema operativo) scelto dal DBA durante la creazione di una base di dati;
 - Se la dimensione di un oggetto di un database cresce (a causa, ad esempio, di inserimenti di tuple in tabelle), un extent aggiuntivo viene allocato per quell'oggetto.

Un Oracle database, solitamente, è costituito da un **tablespace principale (SYSTEM)** che contiene il dizionario dei dati ed altre tabelle interne, procedure, etc... ed un **tablespace per i segmenti di rollback**; ulteriori **tablespace** includono un **tablespace per i dati dell'utente (USERS)**, un **tablespace per i risultati temporanei di query (TEMP)** e uno **utilizzato dalle applicazioni (TOOLS)**. Di seguito è illustrata la struttura logica e fisica di un database Oracle:



La **creazione di una base** di dati è un processo complesso che prevede l'attuazione delle seguenti fasi:

1. **Dimensionamento fisico** della base di dati, sia globale che a livello dei singoli oggetti, in termini di calcolo dello spazio di memorizzazione richiesto su disco;
2. **Creazione** del database;
3. **Definizione** delle politiche di sicurezza;
4. **Creazione** degli utenti/ruoli;
5. **Creazione** degli oggetti (tabelle, indici) della base di dati e definizione dei vincoli;
6. **Creazione** di un'istanza.

In Oracle, tale processo può avvenire andando a considerare quella che è la struttura logica di una base di dati.

La creazione di una base di dati è un processo complesso che prevede l'attuazione delle seguenti fasi:

1. **Dimensionamento fisico** della base di dati, sia globale che a livello dei singoli oggetti, in termini di calcolo dello spazio di memorizzazione (storage) richiesto su disco;
2. **Creazione del database;**
3. **Definizione delle politiche di sicurezza;**
4. **Creazione degli utenti/ruoli;**
5. **Creazione degli oggetti** (tabelle, indici) della base di dati e definizione dei vincoli;
6. **Creazione di un'istanza** (popolamento della base di dati).

In ORACLE tale processo può avvenire andando a **considerare quella che è la struttura logica di una base di dati.**

Il **dimensionamento fisico** di una base di dati deve essere effettuato andando a **considerare per ogni oggetto la dimensione fisica in byte che questo dovrebbe occupare sul disco.** Lo storage totale si ottiene, poi, andando a considerare **la somma degli storage richiesti dai singoli oggetti:** per oggetti **di tipo tabella o vista materializzata,** va prima **calcolato lo storage richiesto per ogni singola tupla** e poi **moltiplicato per il numero di tuple previste a regime.**

Si supponga di **dover calcolare lo storage di una base di dati composta dalle due relazioni seguenti:**

DIPARTIMENTI (Codice, NomeDip)
IMPIEGATI (Matricola, Nome, Cognome, Dipartimento: DIPARTIMENTI)

Con:

- NomeDip, Nome e Cognome di tipo VARCHAR2, ovvero 70 byte;
- Matricola, Dipartimento e Codice di tipo CHAR, ovvero 5 byte.

Si ipotizzi, inoltre, che **all'atto della messa in esercizio della base di dati, dovranno essere inseriti 1000 impiegati e circa 50 dipartimenti.** Allora, **lo storage richiesto dalle due tabelle può essere calcolato come segue:**

$$\begin{aligned} S_{DIP} &= N_t \cdot (S_{\text{Codice}} + S_{\text{NomeDip}}) = 50 \cdot (5 + 70)b = 3,75Kb \wedge S_{IMP} \\ &= N_t \cdot (S_{\text{Matricola}} + S_{\text{Nome}} + S_{\text{Cognome}} + S_{\text{Dipartimento}}) = 100 \cdot (2 \cdot 5 + 2 \cdot 70)b \\ &= 150K \Rightarrow S = S_{DIP} + S_{IMP} + S_{TOL} = (3,75 + 150 + 56,25)Kb = 200Kb \end{aligned}$$

Con **S_{TOL} padding di tolleranza calcolato come arrotondamento al prossimo ordine di grandezza più vicino** (in questo caso, $S - S_{TOL} = 153,75Kb$ allora l'arrotondamento deve far arrivare S a $200Kb$, quindi il valore $56,25Kb$).

Una volta stimato lo storage richiesto, è possibile creare il database che il DBMS dovrà gestire. In questo processo **dovranno essere specificati i parametri di memorizzazione fisica** (data file, ctrl file, redo log file) e **logica** (tablespace, caratteristiche degli extent, dimensione dei block), nonché **altri parametri di inizializzazione.** La sintassi ORACLE per la creazione di un database è abbastanza complessa e richiede una certa esperienza; per questo motivo, **spesso si ricorre a tool con shell grafica** (come Database Configuration Assistant o DBCA) che assistono il DBA nella creazione della base di dati. Inoltre, **le distribuzioni più recenti del DBMS ORACLE**, di solito, **permettono la creazione assistita di un database iniziale a valle dell'installazione stessa.**

Si supponga di aver già creato un database con SID “Afferenze_Dipartimenti” e specificato correttamente tutti i relativi parametri di memorizzazione. A questo punto, è buona norma creare uno o più tablespace dedicati (e non usare quelli di default) per la memorizzazione degli oggetti della base di dati, attraverso l’istruzione (semplificata):

```
CREATE TABLESPACE Nome_Tablespace DATAFILE Nome_File SIZE Dimensione
{, DATAFILE Nome_File SIZE Dimensione, ...};
```

Ad esempio:

```
CREATE TABLESPACE Dipartimenti DATAFILE 'dip1.dbf' SIZE 500k
, DATAFILE 'dip2.dbf' SIZE 500k;
```

La definizione delle politiche di sicurezza e la creazione di utenti e ruoli seguono le problematiche illustrate in precedenza; su ogni database creato, ORACLE crea in automatico due utenti di tipo DBA di default, aventi username **SYS** e **SYSTEM**. Di norma, però, è utile creare un nuovo utente DBA per ogni database creato (associandogli il nuovo tablespace) e lasciare a quest’ultimo l’intera gestione della base di dati; ad esempio, con le credenziali di SYSTEM va lanciato il seguente script SQL:

```
CREATE USER dip_dba DEFAULT TABLESPACE Dipartimenti IDENTIFIED BY
vinni;
GRANT dba, unlimited tablespace to dip_dba;
```

Con il quale si crea un DBA per il servizio database “Afferenze_Dipartimenti”; tutti gli oggetti creati da quest’ultimo andranno ad essere memorizzati nel tablespace “Dipartimenti”. Il DBA, poi, può anche definire ruoli ed altri utenti, connettendosi con le proprie credenziali, concedendogli privilegi per la base di dati da esso gestita.

La creazione degli oggetti della base di dati avviene tramite i comandi DDL mostrati in precedenza; inoltre, in ORACLE, per le tabelle è possibile specificare delle apposite opzioni di storage. In particolare, per gli oggetti di database (tabelle, indici) che richiedono uno spazio proprio di archiviazione, viene creato un segmento in un tablespace. Dato che il sistema tipicamente non conosce la dimensione che l’oggetto del database avrà, vengono utilizzati alcuni parametri di default. L’utente, comunque, ha la possibilità di specificare esplicitamente i parametri di memorizzazione utilizzando la clausola **STORAGE** all’interno dell’istruzione di create table, sovrascrivendo i parametri di sistema e permettendo all’utente di specificare la dimensione prevista dall’oggetto in termini di extents.

Riprendendo l’esempio precedente:

```
CREATE TABLE DIPARTIMENTI (
    Codice CHAR(5) PRIMARY KEY,
    NomeDip VARCHAR2(70)
)
STORAGE(INITIAL 3k, NEXT 1kb MINEXTENTS 1 MAXTEXTENTS 4 PCTINCREASE
0);
```

```
CREATE TABLE IMPIEGATI(
    Matricola CHAR(5) PRIMARY KEY,
    Cognome VARCHAR2(70),
    Dipartimento VARCHAR2(70),
```

```

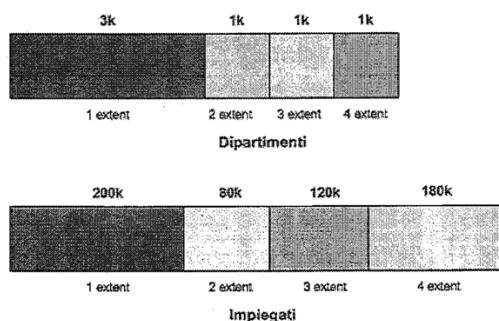
FOREIGN KEY (Dipartimento) REFERENCES DIPARTIMENTI (Codice)
)
STORAGE (INITIAL 200k NEXT 100kb MINEXTENTS 1 MAXEXTENTS 4
PCTINCREASE 50)

```

Si ha che:

- INITIAL e NEXT specificano, rispettivamente, **la dimensione del primo extent e di quelli successivi**;
- Il parametro MINEXTENTS specifica **il numero totale di extents allocati quando il segmento viene allocato**, permettendo di allocare grandi quantità di spazio alla creazione dell'oggetto anche se lo spazio disponibile non è contiguo;
- Il parametro MAXEXTENTS specifica **il numero massimo ammissibile di extents**;
- Il parametro PCTINCREASE specifica **la percentuale con la quale ogni extent, dopo il secondo, cresce rispetto all'extent precedente** (di default 50).

Basandosi sui due precedenti script DDL, si avrà, per le due tabelle, la seguente struttura logica:



Infine, come già osservato in precedenza, **tutti gli oggetti creati dal DBA 'dip_dba' saranno automaticamente raggruppati in uno schema avente lo stesso nome del DBA**.

La creazione dell'istanza avviene con i comandi DML di inserimento illustrati in precedenza; ad esempio, l'inserimento nelle due tabelle:

```

INSERT INTO DIPARTIMENTI VALUES ('000A1', 'Informatica');
INSERT INTO IMPIEGATI VALUES ('22081', 'Vincenzo', 'Moscato',
'000A1');

```

Con la crescente **affermazione di DBMS open-source**, le società commerciali si sono dovute adattare pubblicando versioni "light" gratuite (e spesso re-distribuibili) **dei loro prodotti di punta**, come **ORACLE 10g Express Edition (XE)**; questa versione presenta delle **limitazioni rispetto a quelle enterprise**, come **1GB di memoria SGA utilizzabile, 4GB di memoria disco (storage) utilizzabile, 1 CPU sfruttabile, 1 istanza attiva, un unico Oracle database utilizzabile** (con SID XE e creato all'atto dell'installazione) e la **disponibilità limitata a Windows e Linux**. Di contro, essa risulta essere **facilmente installabile ed utilizzabile anche da utenti non esperti**; oltre ad i **tradizionali e potenti client SQL con shell testuale** (come SQL Plus) questa distribuzione fornisce un **client web, Apex, attraverso il quale risulta molto semplice amministrare e manutenere la propria base di dati**. Le **principal funzionalità di amministratore** delle basi di dati sono disponibili in **forma grafica attraverso i menu principali**:

- **Amministrazione**, permette la creazione di utenti e la definizione delle politiche di sicurezza sulla base di dati, permette la gestione ed il monitoraggio della SGA, dei tablespace e dei data file e permette il monitoraggio delle sessioni utente attive sul database;
- **SQL**, fornisce l'interfaccia dei comandi SQL per la gestione dell'intera base di dati;
- **Browser Oggetti**, permette la navigazione, la creazione e la modifica degli oggetti (tabelle, viste, indici, ...) della base di dati e permette il popolamento e l'aggiornamento di ogni tipo di tabella;
- **Utility**, fornisce funzionalità avanzate come l'impor/export, il caricamento e lo svuotamento della base di dati, nonché quelle per la generazione di report sugli oggetti.

PL/SQL

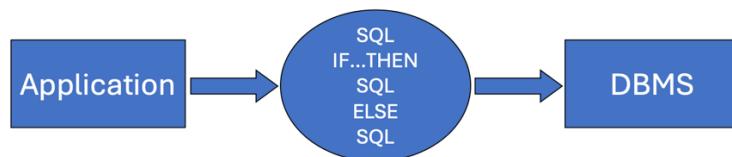
Per **PL/SQL**, acronimo di **Procedural Language/SQL**, si intende **il linguaggio di programmazione procedurale che Oracle ha implementato nei propri sistemi per estendere il linguaggio SQL con costrutti di programmazione**; in altre parole, **unisce la potenza interrogativa e dichiarativa del linguaggio SQL con la flessibilità di un linguaggio di programmazione**. Infatti, **SQL serve per interrogare e manipolare dati** (quindi leggere, inserire, modificare e cancellare) e **PL/SQL serve per gestire la logica intorno ai dati**.

PL/SQL colma il gap esistente tra la tecnologia dei DB e i linguaggi procedurali, usando le facilities di **ORACLE RDBMS** come **gestione delle transazioni, eccezioni, cursori**, facendo così in modo di **estendere SQL aggiungendo elementi di programmazione vera e propria**, come **variabili** (per la memorizzazione di valori temporanea), **strutture di controllo** (loop e condizioni), **gestione delle eccezioni, procedure e funzioni** (per riutilizzare codice).

I vantaggi di questo approccio sono i seguenti:

- **Capacità procedurali**, aggiunge logica di programmazione a SQL;
- **Prestazioni migliorate**, il codice viene eseguito direttamente nel database, riducendo il traffico tra applicazione e server;
- **Produttività migliorata**, permette di riutilizzare codice e riduce la ripetizione di query;
- **Portabilità**, i programmi PL/SQL possono essere eseguiti su qualsiasi sistema Oracle, senza modifiche necessarie;
- **Integrazione con il Relational DBMS**, parte del motore Oracle che permette l'accesso ai dati e la gestione delle transazioni e degli errori in modo nativo.

Il modello client server con cui un'applicazione si interfaccia con un DB, in un sistema tradizionale, prevede che questa invii, in quanto client, molte istruzioni SQL singole al DBMS; utilizzando **PL/SQL**, si consente di raggruppare più istruzioni SQL (e la relativa logica procedurale) in un unico blocco da inviare al DB, riducendo i tempi di scambio client-server non solo in termini di quantità di scambi ma anche in termini di durata dell'esecuzione delle operazioni.



PL/SQL ha diverse caratteristiche e funzionalità e l'unità base di elaborazione è il “blocco”; infatti, **ogni programma PL/SQL, qualunque sia la sua dimensione, è costituito da blocchi suddivisi in sezioni.** Nella sua accezione generale, **la suddivisione in sezione di un blocco avviene come segue:**



Ad esempio:

```
DECLARE
    Num_in_stock NUMBER(5);
BEGIN
    SELECT quantity INTO num_in_stock
    FROM inventory_table
    WHERE product = 'TENNIS RACQUET'
    IF num_in_stock > 0 THEN
        UPDATE inventory_table SET quantity=quantity-1
        WHERE product = 'TENNIS RACQUET'
    ELSE
        INSERT INTO purchase_record
        VALUES ('OUT OF TENNIS RACQUET.', SYSDATE)
    END IF;
    COMMIT;
END;
```

La sezione dichiarativa è quella sezione dove tutte le variabili, cursori e tipi sono dichiarati e, in linguaggio PL/SQL, assume la forma:

```
nomevar tipovar [CONSTANT] [NOT NULL] [:= valore]
```

La sezione esecutiva, invece, è dove si trovano le istruzioni del blocco da eseguire, sia quelle procedurali che quelle SQL. Della struttura a blocco precedentemente osservata, **quella in questione è l'unica ad essere richiesta e non opzionale;** le sezioni dichiarativa e gestionale degli errori sono opzionali. **La sezione di handling delle eccezioni del blocco viene utilizzata per rispondere a runtime ad eventuali errori che si incontrano durante l'esecuzione del programma;** pertanto, si può intuire come il codice di questa sezione non viene eseguito a meno che non si verifica l'errore per cui è stato programmato.

```

DECLARE
    Error_code      NUMBER(5);
    Error_msg       VARCHAR2(200);
    User            VARCHAR2(50);
    Error_info      VARCHAR2 (100);

BEGIN
    ...
EXCEPTION
    WHEN OTHERS THEN
        Error_code:=SQLCODE;
        Error_msg:=SQLERRM;
        User:=USER;
        Error_info='Errore rilevato il ' || TO_CHAR (SYSDATE) || 'dall'utente ' || User;
        INSERT INTO log_table (code, message, info)
        VALUES (Error_code, Error_msg, Error_info);
END;

```

In PL/SQL sono definite già delle variabili per la gestione delle eccezioni più comuni come:

- ZERO DIVIDE
- NO DATA FOUND

PL/SQL permette di definire anche delle eccezioni personalizzate, usando il tipo **EXCEPTION** che serve per **dichiarare variabili che rappresentano eccezioni**. L'istruzione **RAISE** attiva un'eccezione, che viene poi gestita nella sezione **EXCEPTION** del blocco. Inoltre, la clausola **INTO** memorizza il risultato di una query (singolo valore) in una variabile del programma, mentre la funzione **DBMS_OUTPUT.PUT_LINE()** consente di visualizzare messaggi di output a schermo (per debug o log).

Un esempio completo di blocco PL/SQL è il seguente:

```

DECLARE
    Importo Bonifico CONSTANT NUMBER(7):=100000;
    Codice Bonifico CONSTANT CHAR(5):='0A983';
    Saldo Attuale      CONTI CORRENTE.saldo%TYPE;
    Saldo Insufficiente EXCEPTION;

BEGIN
    SELECT saldo INTO Saldo Attuale FROM CONTI CORRENTE WHERE codice=Codice Bonifico;
    IF Saldo Attuale > Importo Bonifico THEN
        UPDATE CONTI CORRENTE SET saldo=saldo-Importo Bonifico WHERE codice=Codice Bonifico;
        COMMIT;
    ELSE
        RAISE Saldo Insufficiente;
    ENDIF;
EXCEPTION
    WHEN Saldo Insufficiente THEN
        INSERT INTO LOG ERROR(CodErrore, CodConto, TipoOp, Msg, Data)
        VALUES ('Err02', '0A983', 'Bonifico', 'Saldo non sufficiente', SYSDATE);
        COMMIT;
        DBMS OUTPUT.PUTLINE('Saldo non sufficiente');
END;

```

L'unità base di ogni programma PL/SQL, come già anticipato, è il blocco; **tutti i programmi PL/SQL sono costituiti da blocchi, in ordine sequenziale o innestati l'uno nell'altro**. Ci sono, però, differenti tipi di blocco:

- **Blocchi Anonimi e Con Nome**, generalmente costruiti dinamicamente e utilizzati una sola volta, differiscono per la presenza di un label (<<label>> prima della keyword DECLARE) che fornisce al blocco un nome;
- **Sottoprogrammi**, sono procedure o funzioni che sono memorizzate nel DB e invocate dall'utente (CREATE PROCEDURE nome AS prima della keyword DECLARE);
- **Triggers**, sono blocchi con nome memorizzati nel DB che sono eseguiti a valle di un evento.

I triggers, a differenza di blocchi con nome e sottoprogrammi, **sono realizzati con un meccanismo leggermente diverso e verranno approfonditi in futuro**. Prima della keyword DECLARE, vanno inserite le seguenti righe di codice:

```
CREATE OR REPLACE TRIGGER nomeTrigger
BEFORE INSERT OF nomeEvento
ON nomeTabella
FOR EACH ROW
```

Come ogni linguaggio di programmazione, **anche in PL/SQL è possibile documentare il codice con linee di commento, ignorate dal motore di compilazione**. Ci sono due stili di commento, single line e multiline:

```
-- Commento single line

/* Commento
Multiline */
```

Le informazioni sono trasmesse da un applicativo PL/SQL e il DB server per mezzo di “variabili”. Le variabili sono dichiarate all'interno della sezione dichiarativa di un blocco PL/SQL, mentre il tipo ne definisce il dominio di appartenenza; inoltre, PL/SQL supporta i tipi definiti dall'utente, come **tabelle e record**:

```
DECLARE
    Impiegato IS RECORD (
        CF NUMBER(20);
        Nome VARCHAR2(50);
        Cognome VARCHAR2(50);
    );
```

I tipi di PL/SQL sono:

- **Numerici**
 - NUMBER
 - DEC
 - DECIMAL
 - DOUBLE PRECISION
 - REAL
 - INTEGER
 - SMALLINT
 - BYNARY_INTEGER
 - NATURAL
 - POSITIVE
- **Carattere**

- VARCHAR
- VARCHAR2
- CHAR
- CHARACTER
- LONG
- **Data**
 - DATE
- **Compositi**
 - TABLE
 - RECORD
- **Booleani**
 - BOOLEAN

In molti casi, **una variabile PL/SQL sarà utilizzata per manipolare dati memorizzati nel DB**. Se si vuole **usare come tipo di dato il tipo usato nella definizione di un attributo di una tabella del DB** si può **usare l'attributo %TYPE** (ad esempio, Title bookinfo.booktitle%TYPE, dove bookinfo è il nome di una tabella e booktitle il nome di un attributo). **%ROWTYPE**, invece, è **utile per dichiarare una variabile di tipo record che ha la stessa struttura di una riga in una tabella o in una vista** (ad esempio, una variabile usata per un fetch da un cursore).

Un **tipo definito dall'utente**, o subtype, è un tipo PL/SQL basato su tipi esistenti e, all'interno di un programma PL/SQL, assume la valenza di un vero e proprio tipo. La sua definizione avviene **dopo la keyword DECLARE**:

```
SUBTYPE Contatore IS NUMBER(4);
```

I record PL/SQL sono simili ai tipi struct del linguaggio C; infatti, permettono di trattare variabili diverse come una sola entità:

```
TYPE record_type IS RECORD (
    field1 type1 [NOT NULL];
    field2 type2 [NOT NULL];
    field3 type3 [NOT NULL];
    ...
);
```

Ad esempio:

```
DECLARE
  TYPE ImpiegatoRecord IS RECORD (
    CF          NUMBER(20);
    Nome        VARCHAR2(50);
    Cognome     VARCHAR2(50);
  );
  ImpiegatoRecord impiegato;
  impiegato.Nome:='Pippo';
```

E, proprio come nel linguaggio C, **per accedere ad una delle variabili attributo del record si utilizza la dot notation**. Nel caso esista una tabella Impiegato con gli stessi campi del record, è possibile scrivere ImpiegatoRecord Impiegato%ROWTYPE.

Le tabelle PL/SQL sono simili agli array in C, vengono trattate come tali anche se sono implementate in maniera differente. La sintassi per la creazione di una tabella è la seguente:

```
TYPE MyTable IS TABLE OF Elements_Type INDEX BY Index_Type;
```

L'accesso agli elementi della tabella avviene con indice:

```
table(index)
```

Ed è anche possibile creare tabelle di record; ad esempio:

```
TYPE Impiegato_Table IS TABLE OF Impiegato%ROWTYPE  
INDEX BY BINARY_INTEGER;  
Impiegato_Table Impiegato_app;  
Impiegato_app(1).Nome:='Pippo';
```

Attributi utilizzabili con le tabelle sono:

- COUNT, ritorna il numero di righe nella tabella;
- DELETE, cancella le righe di una tabella;
- EXISTS, ritorna il valore vero se la entry specificata esiste nella tabella;
- FIRST, ritorna l'indice della prima riga nella tabella;
- LAST, ritorna l'indice dell'ultima riga nella tabella;
- NEXT, ritorna l'indice della riga nella tabella successiva a quella specificata;
- PRIOR, ritorna l'indice della riga nella tabella precedente a quella specificata.

PL/SQL mette anche a disposizione dell'utente delle funzioni per la conversione tra variabili appartenenti a tipi diversi:

- TO_CHAR;
- TO_DATE;
- TO_NUMBER.

In aggiunta, **PL/SQL mette a disposizione una vasta gamma di funzioni a supporto del programmatore**, come funzioni per la gestione di caratteri e stringhe, funzioni numeriche e matematiche e molto altro.

Come si è già potuto intuire, **PL/SQL impiega un diverso operatore di assegnazione dal classico =, :=**, ma **non è l'unico**; infatti, quando l'assegnazione va fatta da una query ad una variabile, l'operatore adatto è **INTO**. In generale, per l'assegnazione di valori a variabili si usa **:=**, mentre per l'assegnazione di risultati di query a variabili si usa **INTO**. Così come per SQL, gli operatori di confronto mettono in comparazione numeri, stringhe e date.

Così come avviene nei linguaggi procedurali classici, **PL/SQL presenta una varietà di strutture di controllo che permettono la manipolazione del flusso di esecuzione di un blocco**; le strutture in questione includono **statement condizionali e loop**. Per quanto riguarda i primi, si può generalizzare dicendo:

```
IF cond THEN  
    blockTrue  
ELSE
```

blockFalse

Per i loop si individuano:

- Loop semplici

```
LOOP
    istruzioni
    EXIT [WHEN CONDITION]
END LOOP;
```

- FOR loop

```
FOR counter IN [REVERSE] low..high LOOP
    istruzioni
END LOOP;
```

- WHILE loop

```
WHILE cond LOOP
    istruzioni
END LOOP;
```

PL/SQL utilizza, per il controllo del flusso di un blocco, anche l'istruzione GOTO label:

```
DECLARE
    contatore BINARY_INTEGER:=1;
BEGIN
    LOOP
        INSERT INTO Prova (id) VALUES (contatore);
        contatore:=contatore+1;
        IF contatore>10 THEN
            GOTO label_endloop;
        END IF;
    END LOOP;
    <<label_endloop>>
END;
```

Come si può facilmente immaginare, PL/SQL dà la possibilità di inserire all'interno di un blocco un qualsiasi statement SQL, DML (Insert, Delete, Select, Update) o DDL (Create Table, Create Index) che sia. Ad esempio, la SELECT:

```
DECLARE
    Impiegato_Record Impiegato%ROWTYPE;
BEGIN
    SELECT * INTO Impiegato_Record
    FROM Impiegato
    WHERE id=1;
END;
```

INSERT e DELETE:

```
BEGIN
    INSERT INTO IMPIEGATO (CF, Nome, Cognome)
        VALUES ('a','b','c');
    DELETE FROM IMPIEGATO
        WHERE CF='a' AND Nome='b' AND Cognome='c';
END;
```

Quando una query ritorna una tabella contenente un certo numero di righe, è possibile definire un cursore per elaborare tutte le righe tornate dalla query e tenere traccia di quale riga è attualmente in elaborazione. Il problema fondamentale è l'**impedance mismatch**: i linguaggi tradizionali gestiscono i record uno alla volta, mentre SQL produce insiemi di tuple; in quest'ottica, il cursore si rivela la soluzione adatta, potendo permettere di accedere a tutte le n-uple di una query in modo globale e di trasmetterle al programma una alla volta. La creazione e l'utilizzo di un cursore avvengono tramite le seguenti istruzioni:

- DECLARE CURSOR, permette di definire un cursore e specificare la query;
- OPEN, inizializza il cursore;
- FETCH, permette di recuperare la riga corrente e viene eseguito fino a che non si sono processate tutte le righe;
- CLOSE, chiude il cursore.

Mentre **per quanto riguarda gli attributi, si consideri che ogni cursore definito esplicitamente ha quattro attributi**:

- Cursor_name%NOTFOUND, ha valore TRUE se l'ultimo FETCH è fallito o sono finite le righe;
- Cursor_name%FOUND, opposto di NOTFOUND;
- Cursor_name%ROWCOUNT, ritorna il numero di righe fetched;
- Cursor_name%ISOPEN, ha valore TRUE se è aperto.

I **FOR LOOPS** permettono implicitamente di aprire un cursore, prelevare ogni riga e chiuderlo quando tutte le righe sono state elaborate.

Oracle apre implicitamente un cursore per elaborare qualsiasi statement SQL e PL/SQL permette di riferirlo come **SQL%**; su di esso non è possibile usare **OPEN, FETCH e CLOSE** ma consente di usare i seguenti attributi:

- SQL%NOTFOUND, ha valore TRUE se INSERT, UPDATE e DELETE non hanno inficiato nessuna riga;
- SQL%FOUND, contrario del precedente;
- SQL%ROWCOUNT, ritorna il numero di righe inficate da un'operazione;
- SQL%ISOPEN, è sempre FALSE essendo un cursore隐式的.

Una transazione è una sequenza di istruzioni SQL che Oracle tratta come un'unità; tutti i cambiamenti fatti in una transazione sono resi permanenti o annullati. PL/SQL permette l'uso di **COMMIT**, per rendere permanenti i cambiamenti fatti nella transazione corrente, **ROLLBACK**,

per terminare la transazione e annullare ogni cambiamento fatto fin dall'inizio della transazione, e **SAVEPOINT**, per marcare il punto corrente nell'elaborazione di una transazione.

Le funzioni in PL/SQL, così come in un qualsiasi linguaggio di programmazione procedurale, permettono di ritornare un tipo, che può essere un tipo classico come il numerico, character o data, oppure un tipo group (che, però, non può essere usato in espressioni PL/SQL). Inoltre, PL/SQL consente un accesso completo a tutti i predicati SQL, ovvero ciò che è usato nelle clausole WHERE, e a tutti gli operatori di confronto, inclusi BETWEEN, IS NULL, LIKE, EXISTS.

Fin qui si è sempre considerata la possibilità di utilizzare SQL in accoppiamento con un linguaggio appositamente creato per permetterne l'estensione; tuttavia, può spesso capitare di dover immergere SQL in applicazioni più complesse e sviluppate in linguaggi di programmazione tradizionali (ad esempio, COBOL, Pascal, C, C++, Java, ...) e PL/SQL smette di avere utilità. In queste circostanze, il codice è analizzato da precompilatori che traducono le istruzioni SQL in apposite istruzioni per il linguaggio specifico, in modo da poter utilizzare, ad esempio, le variabili del programma come parametri nelle istruzioni SQL (precedute da :). In genere, SELECT che producono una sola tupla e operazioni di UPDATE possono essere immerse senza problemi (ovviamente implementando un opportuno meccanismo di trasmissione dei risultati); esiste, di solito, una variabile di sistema, **sqlcode**, che dopo l'esecuzione di un'operazione assume valore "zero" se ha ricevuto degli accessi, altrimenti valore diverso. In C, ad esempio

```
void VisualizzaStipendiDipart(char NomeDip[]) {  
    char Nome[20], Cognome[20];  
    long int Stipendio;  
    $ declare ImpDip cursor for  
        select Nome, Cognome, Stipendio  
        from Impiegato  
        where Dipart = :NomeDip;  
    $ open ImpDip;  
    $ fetch ImpDip into :Nome, :Cognome, :Stipendio;  
    printf("Dipartimento %s\n",NomeDip);  
    while (sqlcode == 0){  
        printf("Nome e cognome dell'impiegato: %s %s", Nome, Cognome);  
        printf("Attuale stipendio: %d\n", Stipendio);  
        $ fetch ImpDip into :Nome, :Cognome, :Stipendio;  
    $ close cursor ImpDip; }
```

ORACLE Precompilers permette di inserire blocchi di PL/SQL in programmi scritti in altri linguaggi di programmazione, detti anche host languages, e tratta un blocco come una singola istruzione SQL embedded.

Quando un'applicazione non conosce a tempo di compilazione lo statement SQL da eseguire, occorre il "Dynamic SQL"; il problema principale da risolvere è la gestione del passaggio dei

parametri tra il programma e l'ambiente SQL, risolto essenzialmente da due modalità di esecuzione:

- **Esecuzione diretta**

```
execute immediate SQLStatement
```

- **Esecuzione con analisi preventiva**

```
prepare CommandName from SQLStatement  
...  
execute CommandName [into TargetList]
```

Dopo aver introdotto i blocchi PL/SQL “anonimi”, è **naturale procedere con il mostrare come questi vengono utilizzati all'interno di procedure e funzioni**, che rappresentano la forma strutturata e riutilizzabile del codice PL/SQL. SQL-2 permette la definizione di procedure, anche note come “stored procedure”, un blocco PL/SQL salvato nel database con un nome, che può essere richiamato e rieseguito in qualsiasi momento. Le procedure e le funzioni (note anche con il nome di sottoprogrammi) in PL/SQL sono molto simili a quelle dei linguaggi procedurali e **non sono altro che blocchi PL/SQL che è possibile attivare mediante una chiamata a procedura (o funzione)**. Come in altri linguaggi di programmazione, **ogni procedura e funzione può essere dotata di parametri di I/O da scambiare con l'esterno**.

La sintassi generale per la creazione di una procedura PL/SQL è la seguente (Corpo_Procedura non è altro che un blocco PL/SQL):

```
CREATE [OR REPLACE] PROCEDURE Nome_Procedura  
(  
    Parametro1 [{IN|OUT|IN OUT}] Tipo,  
    Parametro2 [{IN|OUT|IN OUT}] Tipo,  
    ...  
    ParametroN [{IN|OUT|IN OUT}] Tipo  
)  
  
{AS|IS}  
    -- definizioni variabili  
[BEGIN]  
    Corpo_Procedura  
[END Nome_Procedura;]
```

Dichiarazione delle variabili

Direzione dei parametri (input/output/input e output)

L'esecuzione di una procedura può avvenire in modi diversi sulla base della presenza di parametri di I/O con l'esterno:

Tipo di procedura	Come si esegue
Senza parametri	<code>EXEC Nome_Procedura;</code>
Con parametri IN	<code>EXEC Nome_Procedura(Parametro1, Parametro2, ..., ParametroN);</code> or <code>BEGIN</code> <code> Nome_Procedura(Parametro1, Parametro2, ..., ParametroN);</code> <code>END</code>
Con parametri OUT/IN OUT	<code>DECLARE</code> <code> variabile_da_stampare VARCHAR2(50); --variabile da usare</code> <code>BEGIN</code> <code> Nome_Procedura(variabile_da_stampare); -- chiamo la procedura</code> <code> DBMS_OUTPUT.PUT_LINE(variabile_da_stampare); -- visualizza il valor restituto</code> <code>END;</code>

Una funzione è molto simile ad una procedura ma presenta due caratteristiche principali differenti: restituisce sempre un valore, tramite il comando RETURN, e può essere chiamata all'interno di un SQL, in SELECT, WHERE o ORDER BY. La sintassi generale per la creazione di una funzione in PL/SQL è la seguente:

```
CREATE [OR REPLACE] FUNCTION Nome_Funzione
(
  Parametro1 IN Tipo,
  ...
  ParametroN IN Tipo
)
RETURN Tipo
{AS|IS}
-- definizione variabili
BEGIN
  Logica della funzione
  RETURN valore;
END Nome_Funzione
```

Dichiarazione delle variabili

Nelle funzioni, i parametri sono quasi sempre solo IN, perché lo scopo è restituire un valore tramite RETURN

Funzioni e procedure, poi, possono essere raccolte in packages, segnando il passaggio da semplici blocchi o procedure isolate ad una programmazione più strutturata e modulare. Un package è un contenitore logico che può racchiudere procedure, funzioni, variabili, cursori ed eccezioni, tutti relativi ad un determinato ambito o modulo applicativo. Un package è formato da due parti:

- **Package Specifications** (spec o intestazione), dove si dichiarano gli elementi che saranno visibili all'esterno ed è come se fosse l'interfaccia pubblica del pacchetto;
- **Package Body** (corpo), dove si scrive il codice vero e proprio (le implementazioni di procedure e funzioni dichiarate nella spec) ed è come la “parte interna” o “privata” del pacchetto.

Generalmente, la struttura di scrittura di un package è la seguente:

```
CREATE OR REPLACE PACKAGE Nome_Package AS
  -- Procedure pubbliche
  PROCEDURE Procedure1 (parametri);
  ...
  PROCEDURE ProcedureN (parametri);

  -- Funzioni pubbliche
  FUNCTION Function1 RETURN Tipo;
  ...
  FUNCTION FunctionN RETURN Tipo;
END Nome_Package;
```

L'esecuzione avviene utilizzando la **dot notation**; ad esempio, per un package chiamato Gestione_Film con procedura pubblica InserisciFilm:

```
Gestione_Film.InserisciFilm
```

I TRIGGER E LE BASI DI DATI ATTIVE

Generalmente, la creazione di un software applicativo segue un'architettura tri-layer:

- **Layer presentazione**, in cui vengono specificate e implementate tutte le azioni che l'utente può attivamente fare;
- **Layer applicazione**, in cui vengono specificate e implementate tutte le funzionalità che l'utente attiva con i propri input ma che vengono eseguite in maniera trasparente ad esso;
- **Layer dati**, costituito dal DBMS e dalle regole di implementazione della base di dati.

Una base di dati viene gestita da programmi che, unitamente ai sistemi hardware, **compongono il sistema informatico di supporto al sistema informativo aziendale**.

Ponendosi tra layer applicazione e layer dati, i linguaggi di programmazione dialogano con il database tramite comandi SQL; tuttavia, sulla base della posizione, nell'architettura, del comando SQL in sé e per sé, si determina una certa efficienza. A livello applicazione si parla di **SQL embedded**, è molto facile da implementare ma presenta diverse falle: in primis, se cambia una specifica dell'applicazione per la quale una determinata query SQL necessita di essere cambiata, vanno rintracciati tutti i comandi che includono quella query, vanno modificati e va compilata nuovamente tutta l'applicazione; inoltre, si va a violare il principio di **incapsulazione tra i vari layer**, dal momento in cui un merito del livello dati viene approfondito nel livello superiore, commettendo un errore semantico nella progettazione dell'applicazione. Molti di questi problemi sono risolti da linguaggi come **PL/SQL**, che permettono di far eseguire i comandi SQL al DBMS nativamente, in modo da non solo ripristinare la validità del principio di **incapsulazione** ma permettere anche alla specifica dell'applicazione di non richiedere una manutenzione troppo estensiva in caso di cambiamenti: il layer applicazione non viene inficiato nel caso in cui la specifica dell'applicazione cambia e le query vanno modificate, l'unica cosa che fa è chiamare delle stored procedure o indurre il sollevamento di Trigger che svolgono lo stesso ruolo del SQL embedded.

Una base di dati il cui DBMS non si configura unicamente come deposito ma permette l'esecuzione di comandi e procedure per cambiare sé stessa è detta **base di dati attiva** e permette di modellare realtà più dinamiche senza la necessità di dover interagire a livello applicativo, il quale è trasparente a tutti questi meccanismi: le operazioni sono eseguite automaticamente e indipendentemente dai livelli logici sovrastanti.

Sulla base di quanto appena detto, **una base di dati attiva viene realizzata impiegando stored procedure**, delle quali si è già discusso, e **Trigger**. Il Trigger è una specifica secondo la quale un'azione o una funzione deve attivarsi automaticamente ogni volta viene eseguita un'operazione predefinita su un determinato oggetto all'interno delle basi di dati. I Trigger si basano su tre concetti principali, che definiscono il **paradigma ECA**:

- **Evento**, il fattore scatenante del Trigger;
- **Condizione**, il criterio che deve essere verificato affinché il Trigger venga eseguito;
- **Azione**, l'operazione che viene eseguita se la condizione associata al Trigger attivato risulta soddisfatta.

I Trigger possono essere classificati in diversi modi, uno dei quali è **sulla base del tipo di operazione da cui sono attivati**:

- **Trigger DML** (Data Manipulation Language), sono attivati da operazioni di manipolazione dei dati, come istruzioni su tabelle o viste (sono quelli che, in questa sede, verranno approfonditi di più);
- **Trigger DDL** (Data Definition Language), sono attivati in risposta a eventi di sistema che riguardano la struttura o lo stato del database, come la creazione, la modifica o l'eliminazione di oggetti, nonché operazioni di avvio o arresto del database.

Un'altra classificazione prende in considerazione **l'evento a partire dal quale il trigger è attivato**:

- **Istruzioni DML**, come INSERT, UPDATE o DELETE;
- **Istruzioni DDL**, come CREATE, ALTER o DROP;
- **Eventi di sistema**, come SERVER ERROR, LOGON, LOGOFF, STARTUP o SHUT DOWN.

Oppure sulla **granularità del Trigger**, determinando **su quanti elementi l'evento scatenante del trigger deve agire**:

- **Livello di Tupla** (o **Row-Level**), il Trigger si attiva per ogni singola tupla coinvolta nell'operazione specificata;
- **Livello di Istruzione** (o **Statement-Level**), il trigger si attiva una sola volta per ciascuna istruzione DML specificata, indipendentemente dal numero di tuple coinvolte.

Oppure sulla **modalità di esecuzione**, specificando **il momento in cui il trigger deve essere attivato**:

- **Before Trigger**, il Trigger è eseguito immediatamente prima del verificarsi dell'evento, sempre all'interno della transazione che lo attiva;
- **After Trigger**, il Trigger è eseguito immediatamente dopo il verificarsi dell'evento, sempre all'interno della transazione che lo attiva.

L'impiego di Trigger trova spazio in **regole di business che non potrebbero essere espresse secondo i costrutti dichiarativi** di linguaggi come PL/SQL, nella **segnalazione automatica ad altri programmi** della necessità di dover effettuare delle azioni quando vengono apportate modifiche **ad una tabella**, nell'inserimento di **valori opportuni** in particolari colonne **al momento di un inserimento o di una modifica**, o nella **globalizzazione di alcuni controlli**, che vengono così **affidati al DBMS**.

La creazione di un Trigger è differenziata sulla base dell'istruzione che solleva il Trigger:

- Trigger DML

```
CREATE [OR REPLACE] TRIGGER Nome_Trigger
{BEFORE|AFTER} Evento_DML_Attivante
ON Tabella_Target
[WHEN Condizione_Trigger]
[FOR EACH | {STATEMENT|ROW}]
Corpo_Trigger
```

- Trigger DDL

```
CREATE [OR REPLACE] TRIGGER Nome_Trigger
{BEFORE|AFTER} Evento_DDL_Attivante | (Elenco Eventi Database)
[ON Database_Schema]
[WHEN Condizione_Trigger]
Corpo_Trigger
```

In entrambi i casi, **è necessario specificare il nome del Trigger, la tabella o la vista per cui viene definito, l'evento che ne scatena l'accensione, la granularità e la modalità di attivazione.**

Tuttavia, **utilizzare un Row-Level o un Statement-Level Trigger non è la stessa cosa**, così come **non lo è usare un Before Trigger o un After Trigger**; per comprendere il motivo di questa differenza, si analizzi in particolare come viene eseguita l'azione di un Trigger DML (quelli più utilizzati in questa sede). I Trigger agiscono su due tavole, una **tabella target** (che è quella del database, da effettivamente modificare) e una **tabella di transizione** (una tabella di appoggio non appartenente al database ma che serve solo come slot di memoria temporaneo): quando va fatta un'operazione su una tupla o su una tabella, si copiano gli elementi della tabella target di interesse in una o più tavole di transizione con i valori vecchio e/o nuovo dell'elemento da modificare. Nel caso di un Row-Level Trigger, si spostano una alla volta le tuple coinvolte nell'elaborazione, richiedendo un tempo di computazione lineare rispetto al numero di tuple da elaborare e comportando un overhead nel caso di quantità eccessive; nel caso di Statement-Level Trigger, invece, l'insieme di tuple viene spostato tutto insieme e le singole tuple non vengono trattate individualmente, limitando l'overhead sul sistema. Tuttavia, **non sempre gli Statement-Level Trigger sono da preferire**; infatti, paradossalmente, **quando il numero di tuple coinvolte è contenuto, un Row-Level Trigger è più conveniente e veloce** di un Statement-Level Trigger.

Come già anticipato, **le tavole di transizione possono anche non essere univoche**: nel caso di DELETE e di INSERT, la tabella di transizione è unica e contiene il valore, rispettivamente, vecchio e nuovo della tupla da elaborare, mentre **in caso di UPDATE ci sono due tavole di transizione, una tabella contenente il vecchio valore e una tabella contenente il nuovo**. Tutto ciò ha lo scopo di far permanere la validità delle proprietà ACID alla base di dati.

Per quanto riguarda la **distinzione tra Before e After Trigger**, si consideri il **ciclo di vita di un Trigger** (modello ECA):

1. L'evento si presenta;
2. Il Trigger verifica la condizione;
3. L'azione è eseguita.

A differenza di una query, per la quale è a discrezione dell'utente scegliere il momento in cui è eseguita, un Trigger rimane sempre in ascolto sulla base di dati, indipendentemente da cosa l'utente vuole ed è, quindi, **in grado di rilevare un evento prima che questo si rifletta fisicamente**.

sulla base di dati. Ciò significa che **con un Before Trigger l'azione avviene dopo che l'evento è stato rilevato ma prima che i suoi effetti si siano propagati sulla struttura fisica;** di contro, **un After Trigger assicura che l'azione avvenga dopo che l'evento si è propagato sulla struttura fisica.** Questa distinzione può portare, se non ben programmata, ad **elaborazioni indesiderate e/o onerose;** ad esempio, **si consideri il caso di un sanity check** (si verifica il rispetto sul vincolo relativo al dominio di un certo dato, come il NOT NULL):

```
CREATE OR REPLACE TRIGGER trg_Genere
BEFORE UPDATE ON FILM
FOR EACH ROW
BEGIN
    IF :NEW.Genere IS NULL THEN
        :NEW.Genere := '0';
    END IF;
END;
```

Con un **Before Trigger**, si evita di fare due accessi in memoria; infatti, supponendo di aver usato un **After Trigger**, il **primo accesso** cambierebbe il valore di **Genere** da **x** a **NULL**, mentre il **secondo accesso** cambierebbe il valore di **Genere** da **NULL** a **0**. Usando un **Before Trigger**, si **verifica prima il valore nuovo e, in caso non sia valido, lo si modifica prima che venga propagato in memoria;** **Genere** è, poi, **modificato una sola volta** quando si è sicuri che il suo prossimo valore sia possibile.

Si consideri che **:NEW accede alla tabella di transizione con il nuovo valore**, mentre **:OLD alla tabella di transizione con il valore vecchio.** In conclusione, **computazionalmente è preferibile il Before Trigger** ma, in alcuni specifici casi, **non si esclude la possibilità di necessitare di un After Trigger.**

Per creare un Trigger è necessario avere i privilegi per farlo; eventualmente, il Trigger viene inserito nello schema specificato o in quello associato alla tabella indicata, automaticamente accendendosi solo per l'utente associato allo schema. Più Trigger possono essere associati ad una stessa tabella e, in tal caso, l'ordine di esecuzione sarà il seguente:

1. Esecuzione dei **Before Trigger a livello istruzione**;
2. Esecuzione dei **Before Trigger a livello tupla**;
3. Esecuzione dell'**evento di innescio** dei Trigger;
4. Esecuzione degli **After Trigger a livello tupla**;
5. Esecuzione degli **After Trigger a livello istruzione**.

Come si è già potuto intuire, gli utenti finali non accedono al contenuto della base di dati direttamente attraverso l'uso del linguaggio SQL ma attraverso moduli applicativi o procedure che realizzano apposite interfacce di accesso; in questo modo, l'operazione SQL viene nascosta all'utente e racchiusa all'interno del modulo applicativo la cui implementazione è compito del programmatore. In quest'ottica, è importante osservare i modi con cui le applicazioni accedono ai dati; i principali sono di seguito riassunti:

- **SQL embedded**

Le istruzioni SQL per l'interazione con la base di dati sono direttamente **incapsulate nel linguaggio di programmazione ospite** e **identificate attraverso prefissi speciali (EXEC SQL).** Un **precompilatore o preprocessore** esamina prima di tutto il codice sorgente del programma per **identificare le istruzioni di interazione con la base di dati**, le quali vengono poi sostituite da

chiamate al codice generato dal DBMS. La comunicazione dei risultati elaborativi dal DBMS all'applicazione avviene, poi, tramite apposite variabili condivise, mentre le variabili del programma possono essere usate come parametri nelle istruzioni SQL. Le operazioni di selezione che producono una sola tupla e le operazioni di aggiornamento possono essere immerse senza problemi, di contro, le SELECT che ritornano più tuple o record vanno gestite con i cursori.

- **Linguaggi di programmazione per basi di dati**

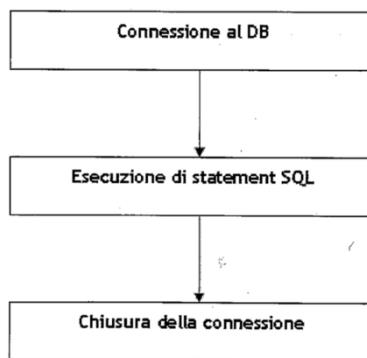
Vengono definiti **appositi linguaggi di programmazione** in cui le istruzioni SQL sono arricchite ed estese da istruzioni tipiche di un linguaggio di programmazione procedurale, come **PL/SQL**.

- **SQL/CLI (Call Level Interface)**

Delle **apposite librerie di funzioni** sono messe a disposizione del programmatore e contengono **funzioni per la connessione al database, per l'esecuzione di interrogazioni, per l'aggiornamento dei dati, ...** I comandi effettivi di interrogazione e aggiornamento della base di dati e, qualsiasi altra informazione necessaria, vengono inclusi come parametri nelle chiamate a funzione. Questo approccio fornisce delle **apposite API** per l'accesso ad una base di dati da programmi applicativi.

L'approccio **SQL embedded risulta essere statico**, in quanto il testo dell'interrogazione è riscritto all'interno del programma e **non può essere cambiato senza ricompilare o rielaborare il codice sorgente**; inoltre, poiché ogni applicazione, per interfacciarsi ad un DBMS, fa riferimento a delle **proprie librerie**, se l'applicazione deve utilizzare per qualche ragione un nuovo database, diverso dal precedente, occorre riscrivere tutto il codice di gestione dei dati.

Per rimediare alle problematiche sopra esposte, è stato creato uno standard a “**livello di chiamata**” per l'interfacciamento alle basi di dati, detto **SQL/CLI**. L'uso delle chiamate a funzione risulta essere un approccio dinamico per la programmazione delle basi di dati: per l'accesso alla base di dati viene usata una libreria di funzioni che, da un lato fornisce maggiore flessibilità e non richiede la presenza di alcun preprocessore, dall'altro, però, comporta che la verifica della sintassi e altri controlli sui comandi SQL avvenga solo al momento dell'esecuzione del programma. Con SQL/CLI viene definita la sequenza standard di chiamate a funzione che un'applicazione deve eseguire per accedere in modo corretto alle informazioni presenti nella base di dati:

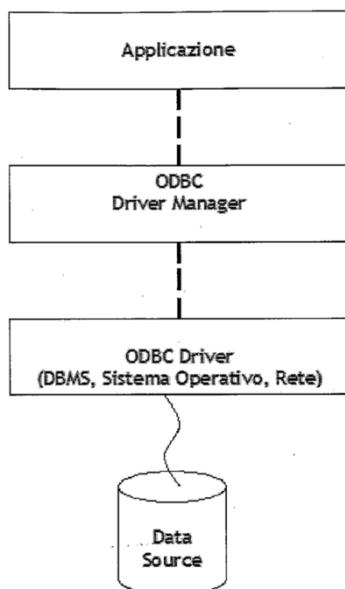


Nasce, allora, l'esigenza di standardizzare l'interfaccia attraverso la quale un'applicazione accede ad un qualsiasi sistema di base di dati relazionale (ORACLE, Access, DB2, MySQL, ...), garantendone l'interoperabilità; ad esempio, nel 1991 Microsoft ha proposto uno standard per la comunicazione con database noto col nome di **ODBC, Open Database Connectivity**. Tramite l'interfaccia ODBC, applicazioni eterogenee, a mezzo dei classici comandi di SQL (inizialmente una versione ristretta, caratterizzata da un insieme minore di istruzioni), possono accedere

direttamente a dati remoti presenti su un server di basi di dati. Tale architettura si basa sul **paradigma di comunicazione Client-Server**; nel caso di un **Database Management System**, il servizio offerto è quello di accesso ai dati presenti **all'interno della base di dati**, mentre il linguaggio di formulazione delle richieste da parte dei client è **SQL**. Nell'architettura ODBC il **collegamento tra un'applicazione client e il server della base di dati** richiede l'uso di un **driver**, ovvero di una libreria che viene collegata dinamicamente all'applicazione e da essa invocata quando si vuole accedere alla base di dati: **ogni driver maschera le differenze di interazione legate non solo al DBMS ma anche al sistema operativo e ai protocolli di rete utilizzati**; in altri termini, i **driver astraggono dai protocolli specifici dei produttori dei DBMS**, fornendo un'interfaccia di programmazione delle applicazioni comune ai client del database (i produttori di DBMS devono fornire diversi driver a seconda del protocollo di comunicazione scelto e dal sistema operativo della macchina server). **Usando nel proprio client le API all'interfaccia OCDB, si rende il proprio programma capace di accedere**, quindi, **a più server di database senza dover conoscere le interfacce proprietarie di ogni singolo database**.

L'accesso ad una base di dati remota mediante ODBC richiede la cooperazione di quattro componenti:

- **L'applicazione**, richiama le funzioni SQL per eseguire un'interrogazione e acquisire i risultati;
 - Tramite i driver, all'applicazione sono trasparenti il protocollo di comunicazione, il tipo di DBMS e il sistema operativo installato sul server;
- **Il driver manager**, è responsabile di caricare i driver di connessione su richiesta dell'applicazione;
- **I driver**, sono responsabili di eseguire le funzioni ODBC e, pertanto, le interrogazioni SQL, traducendole nel dialetto del DBMS locale restituendo i risultati alle applicazioni tramite meccanismi di buffering;
- **La sorgente dati (data source)**, è la fonte delle informazioni, il sistema remoto che esegue le funzioni trasmesse dal client.



In ODBC le interrogazioni SQL possono essere specificate in modo statico (e compilate) o essere incluse in stringhe che vengono generate ed eseguite dinamicamente.

JDBC è un insieme di API ad oggetti basato sullo standard CLI e definito dalla Sun Microsystems. Consiste di un **set di classi ed interfacce scritte in Java** (le API JDBC sono contenute nei package `java.sql` e `javax.sql`) e **costituisce un caso particolare di ODBC**, fornendo una collezione di oggetti e metodi per l'interazione con una base di dati; in particolare, **JDBC risulta molto utilizzato per garantire l'interazione di programmi Java con basi di dati**, fornendo ai programmatori i seguenti servizi per l'accesso ai dati residenti in database relazionali:

- Connessione al database;
- Invio di statement SQL;
- Processing dei risultati;
- Gestione delle transizioni.

JDBC è divenuto, ormai, **uno standard de facto per lo sviluppo di applicazioni Java DB-oriented** ed è **parte integrante della distribuzione software di Java** (il **JDK**).

BASI DI DATI DIREZIONALI

Come è già stato preannunciato, **un sistema informativo è l'insieme dei flussi informativi necessari ad un'organizzazione per poter perseguire i suoi scopi** ed affianca, pertanto, tutti i processi aziendali, agevolandone l'esecuzione e migliorandone i rendimenti generali. Un'azienda potrebbe essere **costituita da una serie di sezioni che necessitano l'impiego di sistemi informativi diversi**; infatti, **non esiste un unico sistema informativo per azienda**, questi devono gestire il ciclo di vita delle informazioni, che possono richiedere una complessità non gestibile. **Ogni azienda**, almeno dal punto di vista astratto, **può essere immaginata come costituita da due livelli**:

- **Livello operativo** (il primo ad essere costruito), l'azienda si occupa delle attività attraverso cui produce i propri servizi e/o prodotti;
- **Livello direzionale** (complemento statistico del secondo), vengono svolte tutte quelle attività necessarie alla definizione degli obiettivi da raggiungere ed alle azioni, eventualmente correttive, da intraprendere per raggiungerli.



Tra i due livelli, chiaramente, c'è uno scambio di informazioni: il **livello direzionale fornirà al livello operativo le istruzioni sulle strategie da seguire per perseguire gli obiettivi aziendali**, mentre il **livello operativo fornirà**, a sua volta, **al livello direzionale le informazioni sui risultati raggiunti**. Nell'elaborazione di queste informazioni di feedback, il **livello direzionale è supportato dai cosiddetti sistemi informativi direzionali**, che sostengono i tre ulteriori livelli della gerarchia direzionale introdotta da Anthony nel 1965:



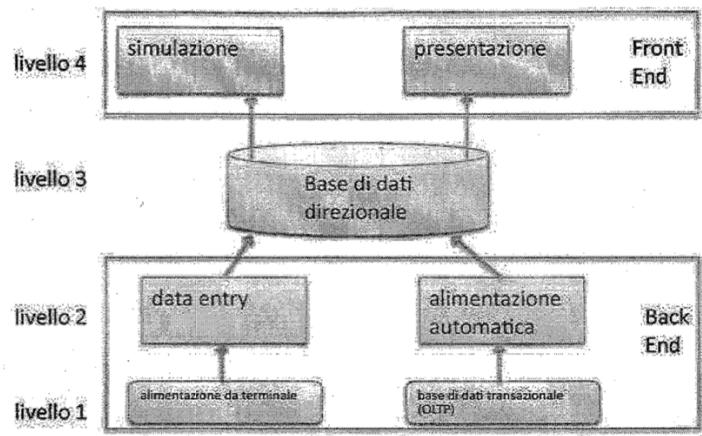
La **pianificazione strategica** determina gli **obiettivi generali dell'azienda**, il **controllo direzionale** definisce **traguardi economici** e la loro **verifica**, il **comando operativo assicura che le attività procedano nel modo prefissato**. Affinché i dirigenti aziendali ai tre livelli indicati possano svolgere correttamente i loro compiti, è necessario che essi ricevano dal livello operativo un feedback memorizzato in una base di dati direzionale. È chiaro che, ai vari livelli, le **informazioni della base di dati direzionale siano differenti**; ad esempio, la **pianificazione sarà interessata a scenari macroeconomici e ad analisi di mercato**, mentre il **controllo direzionale sarà interessato a verificare le informazioni relative al raggiungimento degli obiettivi a medio termine** e il **controllo operativo acquisirà continuamente informazioni dai processi aziendali per verificare che i piani esecutivi siano rispettati**.

Si capisce, quindi, come i **sistemi informativi direzionali abbiano la caratteristica di essere alimentati automaticamente da altri sistemi**; le **informazioni** in questione, inoltre, **risultano essere fortemente aggregate**, dovendo fornire ai dirigenti aziendali dati sintetici che possano quantizzare l'andamento dell'azienda in certi intervalli temporali. Il fatto che queste **informazioni abbiano significato solo in certi intervalli temporali** si esprime dicendo che le **informazioni temporali sono tempificate**. L'analisi delle prestazioni aziendali, inoltre, può essere condotta in diverse dimensioni: il tempo (può essere utile ad un dirigente avere indicazioni sull'evoluzione temporale di un certo indicatore) il prodotto (finalizzata all'analisi di costi e ricavi, tramite indicatori contabili o monetari) o i processi (finalizzata al controllo di indicatori di efficienza ed efficacia, come la tempestività).

I **sistemi informativi direzionali risultano essere dei sistemi piuttosto complessi**; al fine di poter dominare il problema, è **conveniente ricorrere ad una strutturazione del sistema su più livelli**, ciascuno dei quali opera delle trasformazioni sui dati provenienti dai sistemi di supporto operativo. Si ricorda che **questi sistemi hanno due caratteristiche fondamentali**:

- Presentano all'utente finale poche informazioni sintetiche aggregate, secondo le esigenze dell'utente manageriale;
- Sono sistemi parassiti, in quanto prelevano informazioni da altre fonti.

Gli strati che compongono questi sistemi sono il **front-end** e il **back-end**: il primo comprende tutte le **elaborazioni necessarie alla visualizzazione delle informazioni utili all'utente finale** (saranno contenuti moduli di simulazione o di presentazione dei risultati in forma grafica, secondo le esigenze previste), mentre il secondo provvede ad **alimentare automaticamente la base di dati direzionale in maniera periodica**, estraendo le informazioni di interesse per il sistema dai sistemi di supporto operativo.



Nel dettaglio:

- Al **livello 1** si trovano i serbatoi dei dati, che forniscono informazioni grezze al sistema direzionale;
 - A questo livello sono presenti le basi di dati transazionali classiche ed altre sorgenti di dati e le informazioni di carattere generale che gli utenti finali del sistema possono ritenere utili e che vanno inserite manualmente da terminale, perché non contenute nella base di dati transazionali di supporto;
- Il **livello 2** interfaccia la base di dati direzionali con il livello 1 ed individua due grossi sottosistemi:
 - Il data entry, per l'acquisizione manuale dei dati non ricavabili dalla base di dati transazionale di supporto;
 - Modulo per l'alimentazione automatica, che trasformerà le informazioni operazionali in macro-informazioni per la base di dati direzionale (in quanto tempificate ed aggregate);
- Il **livello 3** è il livello della base di dati direzionale (o data warehouse), il cui schema è multidimensionale e verrà approfondito a breve;
- Il **livello 4** è quello più alto dell'architettura, si rivolge all'utente realizzando, mediante appositi strumenti software, il front-end del sistema;
 - Le funzionalità sviluppate a questo livello (On Line Analytical Processing, OLAP) godono di alcuni requisiti indicati con l'acronimo FASMI (Fast Analysis of Shared Multidimensional Information), enfatizzando l'obiettivo fondamentale relativo alla velocità di recupero dati.

La soluzione relazionale alla rappresentazione della base di dati del precedente livello 3 viene indicata con il termine di **Data Warehouse (DWH)**; il concetto originario di DWH fu introdotto da IBM col termine **“Information Warehouse”** e proposto come una **soluzione per l’accesso a dati memorizzati in sistemi non relazionali**. Successivamente fu **riproposto come strumento per permettere alle aziende di usare le informazioni contenute nei loro archivi per effettuare analisi volte a facilitare il raggiungimento degli obiettivi aziendali**, supportando i processi direzionali e fornendo una solida piattaforma integrata di informazioni storico-temporali su cui poter effettuare le analisi di interesse.

Per **data warehouse (DWH)** si intende una base di dati relazionale progettata ai fini dell’analisi dei dati e che gode delle seguenti proprietà:

- **Subject Oriented;**
- **Integrata;**

- Time Variant;
- Non Volatile;

I dati di interesse per un DWH provengono dagli ambienti operativi in quasi tutti i casi; il DWH, però, è un deposito di dati opportunamente trasformati e fisicamente separati dai dati trovati negli ambienti operativi.

La prima caratteristica di un DWH è che esso è una collezione di dati orientata al soggetto, in contrasto con le più classiche applicazioni di basi di dati che avevano un orientamento processivo/funzionale. Le applicazioni classiche sono concepite progettando la base di dati ed i processi elaborativi che operano su di essa a partire dall'analisi delle procedure aziendali; di contro, il progetto di un DWH si focalizza esclusivamente sul modello dei dati e sul progetto della base di dati. La differenza tra applicazione ad orientamento processivo/funzionale e applicazione subject-oriented, inoltre, fa emergere una differenza sostanziale tra i contenuti in una base di dati classica ed un DWH: un DWH utilizza dati impiegati nei processi decisionali, mentre nel caso classico la base di dati contiene tutti dati che possono immediatamente soddisfare le esigenze delle procedure aziendali; un'ulteriore differenza tra i due mondi è nelle relazioni tra i dati: i dati operativi mantengono le relazioni tra due o più tabelle mediante legami dettati effettivamente dalle regole del business e, pertanto, godono di una certa stabilità, mentre un DWH copre, viceversa, un certo arco temporale e le relazioni che si possono individuare tra i dati sono molteplici e legate alle dimensioni di analisi di interesse (a seconda di come si intrecciano le diverse dimensioni di analisi, è possibile individuare molte regole del business da rappresentare nel DWH).

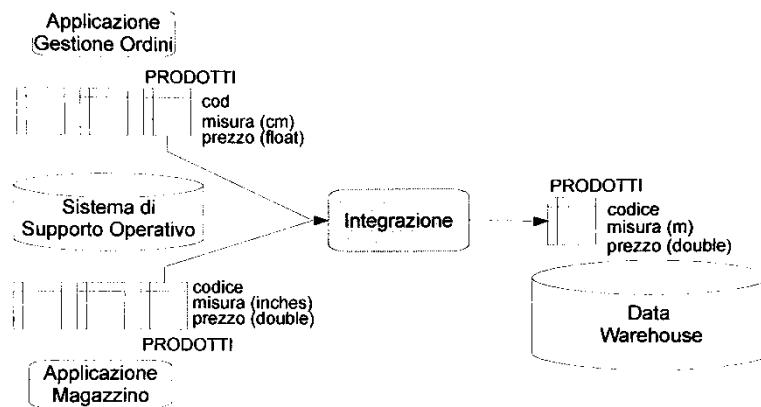
Da queste prime osservazioni, appare subito evidente che un DBMS pensato per sistemi OLTP risulta inadeguato per un DWH; tipicamente, i sistemi OLTP sono progettati per massimizzare la capacità di elaborazione delle transazioni, mentre un DWH è progettato per supportare elaborazioni di query ad hoc. Le differenze sostanziali tra i due sistemi sono riassunte, infine, di seguito:

Sistemi OLTP	Sistemi DWH
Memorizzano dati correnti	Memorizzano dati storici
Memorizzano dati dettagliati	Dati debolmente o fortemente aggregati
I dati sono dinamici	I dati sono pressoché statici
Le elaborazioni sono ripetitive	Effettuano elaborazioni ad hoc
Massimizzano il throughput rispetto alle transazioni	Throughput di transazioni medio-basso
Sono transaction driven	Sono Analysis driven
Sono orientati alle applicazioni	Sono orientati al soggetto
Supportano le decisioni day-to-day	Supportano decisioni strategiche
Servono un grosso numero di utenti operativi	Servono un numero di utenti manageriali relativamente basso

Si noti che nelle realtà aziendali, un'organizzazione possiede quasi sempre diversi sistemi OLTP per differenti business process ed un singolo sistema direzionale che memorizzerà dati storici dettagliati ed aggregati ai vari livelli; il DWH viene, inoltre, progettato per supportare un numero di transazioni basso e per fornire risposte a query non predibili, formulate ad hoc. Nonostante ciò, i due sistemi sono relazionati dal fatto che gli OLTP rappresentano le sorgenti dei dati per il DWH.

Probabilmente, il più importante aspetto di un ambiente DWH è che i dati in esso presenti sono integrati e questa integrazione può essere evidenziata in modi differenti, quali: consistenza tra le

misure delle variabili, attributi fisici dei dati, strutture di codifica e convenzioni sui nomi. Quando i dati vengono trasferiti in un DWH dal mondo operativo, vengono opportunamente **filtrati ed organizzati**, al fine di realizzare un'integrazione come quella illustrata di seguito:



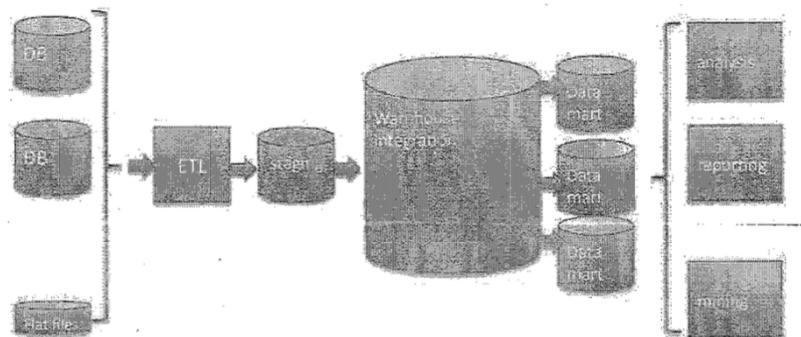
L'obiettivo di perseguire questa integrazione è fondamentale ed intuitivo se si pensa al fatto che i sistemi che alimentano un DWH possono essere differenti e, quindi, avere caratteristiche differenti e formati tra loro inconsistenti.

Relativamente alla caratteristica di tempo varianza, tutti i dati contenuti in un DWH si riferiscono ad un preciso arco temporale; questa dipendenza dal tempo si espleta in diversi modi: il più semplice è quello per cui un DWH rappresenta dati sul lungo periodo, come cinque o dieci anni, mentre il periodo di tempo nel quale i dati contenuti in una base di dati operativa risultano validi e consistenti è molto più breve, nell'arco dei giorni o delle settimane; i dati di un DWH, una volta che sono stati inseriti correttamente, non possono essere modificati (non volatilità), un DWH contiene una lunga serie di istantanee e i dati possono solo essere caricati ed acceduti.

Poiché i dati contenuti in un DWH provengono dagli ambienti operativi, si potrebbe pensare che tra i due sistemi (DWH e base di dati operativa) vi sia notevole ridondanza; in realtà, questa ridondanza è minima, dal momento in cui:

- I dati sono filtrati prima di passare dagli ambienti operativi al DWH;
- L'arco temporale coperto dai due sistemi è molto differente, i dati di un DWH che hanno la stessa temporizzazione di quelli della base di dati operativa sono molto pochi;
- I DWH contengono dati riassuntivi che non sono esplicitamente rappresentati nelle basi di dati operative.

I DWH hanno una struttura caratteristica: i dati sono organizzati a diversi livelli di aggregazione e dettaglio, come di seguito illustrato, e si possono distinguere le differenti componenti:



- **Data source**, il DWH utilizza sorgenti di dati eterogenee, formate dai sistemi operazionali (quindi dati prelevati in ambienti di produzione) di tipo relazionale, legacy (applicazioni aziendali esistenti che non rispondono a requisiti architetturali moderni ed attuali), provenienti da sistemi informativi esterni strutturati o da flat files (dati non strutturati in tabelle relazionali);
- **ETL**, i dati delle varie sorgenti devono essere estratti opportunamente, ripuliti per eliminare eventuali incongruenze ed inconsistenze, completati di parti mancanti ed integrati secondo uno schema comune (strumenti di ETL, Extraction Transformation and Loading, hanno lo scopo di integrare i vari dati provenienti dalle diverse sorgenti informative);
- I dati integrati, corretti, filtrati e validati sono materializzati in un'opportuna area di **Staging** che contiene i cosiddetti dati riconciliati (un modello di dati comune e di riferimento per l'azienda);
- **Warehousing integration**, le informazioni vengono, quindi, raccolte in un singolo contenitore logicamente centralizzato, che è, in senso più stretto, il DWH, e accanto alle quali assumono particolare rilevanza i metadati che mantengono informazioni relative alle sorgenti, ai meccanismi di accesso ai dati, alle procedure di pulitura e alimentazione, ...;
 - Su questa struttura sono fatte statistiche, analisi e data mining;
- **Data marts**, logicamente contiene un sottoinsieme o un'aggregazione dei dati presenti nel DWH centralizzato e informazioni relative, ad esempio, ad una particolare area di business o un particolare dipartimento aziendale (vendita, produzione, ...);
- **Strumenti di analisi dei dati**, ai fini della stesura di report (reporting) o di analisi e simulazione avanzate (OLAP).

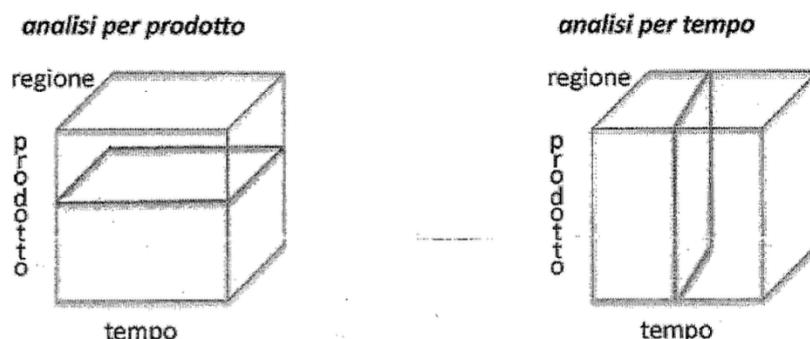
Un DWH viene costruito per **fornire un accesso facile a sorgenti contenenti una grossa quantità di dati**; pertanto, lo si può definire uno strumento sofisticato per arrivare ad effettuare analisi e prendere opportune decisioni. Le **tecniche di analisi dei dati** comunemente usate sono dettagliate di seguito:

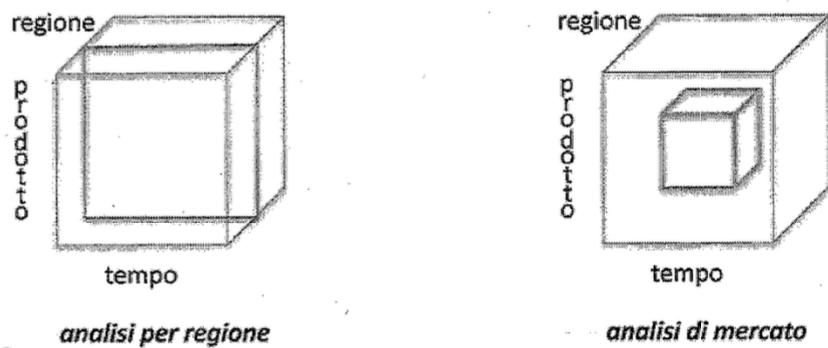
- **Query and reporting (Q&R)**

Il Q&R è il processo che permette di porre una query, recuperare i dati fondamentali dal DWH, trasformare i dati in modo appropriato, porre i risultati in un formato leggibile (reporting) e presentare il report ottenuto alla dirigenza.

- **Multidimensional Analysis**

L'analisi dei dati viene eseguita sui dati estratti dal DWH o dai Data Mart e viene rappresentata in forma multidimensionale, dove ogni dimensione rappresenta un punto di vista di interesse dell'analisi; ad esempio, in un'azienda per la quale sono rilevanti tre dimensioni fondamentali di analisi, prodotti, tempo e mercati geografici, si può pensare ad un cubo:





Gli approcci per l'implementazione dell'analisi multidimensionale legata ad un approccio OLAP sono due:

- **Multidimensional OLAP** (MOLAP, non più tanto utilizzato), se l'insieme dei dati è archiviato su una struttura dati a matrice sparsa, dove sono registrate tutte le sintesi statistiche degli incroci multidimensionali possibili;
 - Il visualizzatore dei dati, in questo caso, chiede i dati direttamente alla base di dati multidimensionale;
- **Relational OLAP** (ROLAP), se l'insieme di dati è registrato su una o più tabelle relazionali;
 - I dati, in questo caso, sono acceduti tramite query su di essi con le opportune sintesi necessarie per la visualizzazione dei dati.

I vantaggi di una soluzione rispetto all'altra sono i seguenti: **nel caso ROLAP i dati acceduti sono sempre gli ultimi disponibili ma**, di contro, **una volta usciti dal visualizzatore i dati di sintesi si perdono** e per riaccedervi è necessario eseguire nuovamente le estrazioni e le sommarizzazioni; per quanto riguarda il MOLAP, **il grosso vantaggio risiede nei tempi di risposta**, anche se di contro il **Multidimensional database deve essere allineato all'aggiornamento dei dati di base dal quale viene generato**. A ben vedere, la soluzione MOLAP ha come perno il **concepto di array multidimensionale** (o ipercubo) intesa come **tecnica per la riorganizzazione e la memorizzazione di dati opportunamente aggregati**, in modo che possano essere **analizzati facilmente da più prospettive**. Un ipercubo è costituito da un **insieme di celle di dati**, ciascuna delle quali contiene **il valore assunto da una specifica misura**, trovato in base alla formula di calcolo e alle dimensioni che determinano il processo di aggregazione: in tale struttura, ciascuna dimensione funge da indice per l'individuazione di un insieme di celle di dati, eventualmente composte da un singolo elemento.

Le operazioni sui dati multidimensionali sono:

- **Roll Up** (aggregazione dei dati);
- **Drill Down** (disgregazione dei dati);
- **Slice and Dice** (selezione e proiezione su un piano);
- **Pivot** (riorientamento del cubo dei dati).

Si noti che Gartner Group sostiene che **i database MOLAP permettono di concentrarsi sulla business view** (ovvero, sugli aspetti più propriamente legati alla Business Intelligence), mentre **i ROLAP favoriscono la system view** impedendo, di fatto, un'interazione diretta dei progettisti con i responsabili aziendali, poco pratici in materia sistematica.

I metadati rappresentano **informazioni sui dati** contenuti in una base di dati e **la loro gestione è un compito estremamente complesso**. Innanzitutto, **i metadati consentono di individuare i dati all'interno di un DWH**, sebbene svolgano anche altre funzioni legate alle **operazioni di loading, query generation e data management**. Relativamente alle **funzioni di loading**, **i metadati**

descrivono la sorgente dei dati e le modifiche effettuate sui dati soggetti ad operazioni di trasformazione. Nel caso di **gestione di dati**, invece, i metadati descrivono l'organizzazione dei dati contenuti nel DWH (tabelle, viste, indici, ...), sono contenuti nel catalogo relazionale, che può essere indicato come basi di dati di metadati, e presentano una struttura relazionale. Infine, i metadati sono legati anche al sistema per la **gestione delle query**, che può generare informazioni sui profili di query utilizzati da diversi gruppi di utenti al fine di migliorare le prestazioni delle query.

Associato al concetto di DWH è il concetto di **data mart**, come precedentemente annunciato. Un **data mart** è un sottoinsieme di un DWH che supporta un particolare dipartimento o una particolare funzione direzionale, possono essere autonomi o collegati al DWH centrale e, essendo più piccoli del DWH, sono più facilmente gestibili e più efficienti. Gli approcci per la costruzione dei data mart possono essere diversi: è possibile, infatti, creare dei **data mart** come viste a partire da un DWH oppure costruire delle infrastrutture dedicate e, successivamente, integrate col DWH centralizzato.

Spesso, al mondo del DWH, sono fortemente correlate le tecniche di **data mining**, usate largamente ai fini del supporto decisionale. Col termine Knowledge Discovery in Database (KDD, 1989) si intende mettere in evidenza che la conoscenza è il prodotto finale di una scoperta guidata dai dati ed individua, dunque, l'intero processo di scoperta della conoscenza dei dati. Il **data mining** è un passo fondamentale di questo processo (che include attività di preparazione, trasformazione, pulizia, interpretazione e valutazione): è un **algoritmo o un insieme di algoritmi per estrarre caratteristiche e regole significative dai dati**. In questo contesto, per **dato** si intende un insieme di fatti, mentre per **pattern** un sottoinsieme dei dati, ovvero un **modello che ben si presti a rappresentare le caratteristiche di questo sottoinsieme**. I pattern scoperti, modelli sintetici e ricchi di semantica, dovranno essere validati anche su tutti i nuovi dati, sebbene con qualche grado di incertezza. Il **data mining**, allora, consiste nell'applicazione di algoritmi di analisi dati che, sotto certe assunzioni accettabili di efficienza computazionale, producono un numero accettabile di pattern sui dati; in particolare, gli algoritmi di data mining cercano di individuare le regole nascoste nelle informazioni e le rendono visibili.

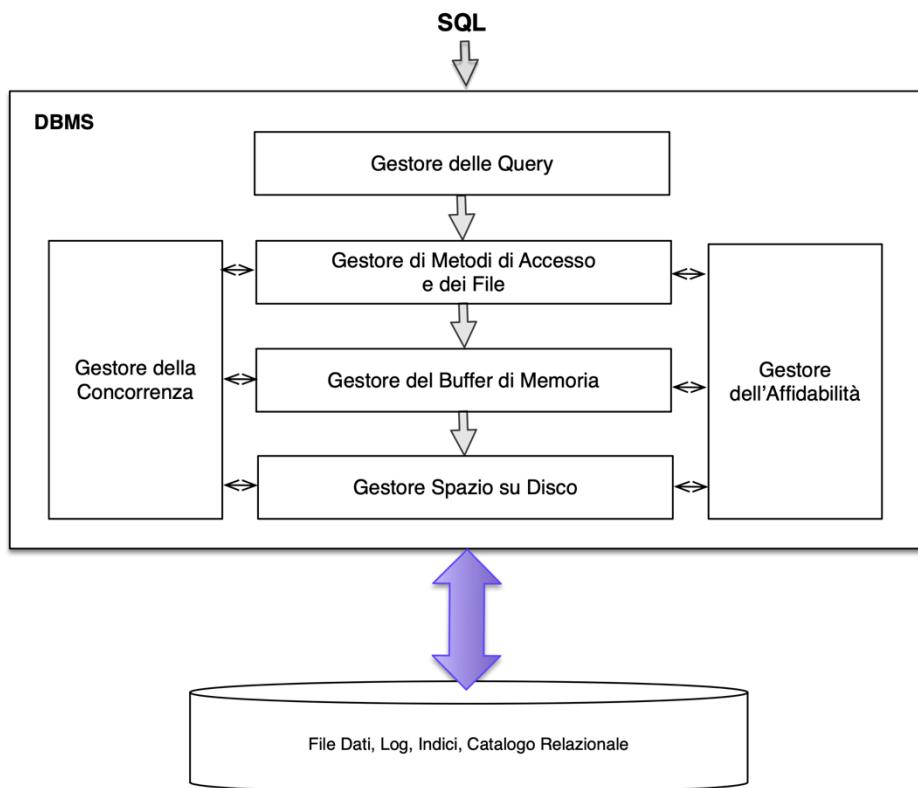
Il progetto di un DWH può partire da uno schema iniziale, detto **star-schema**, una struttura logica che ha al centro una tabella dei fatti, con intorno le diverse tabelle delle dimensioni. La tabella dei fatti memorizza misure di business numeriche, mentre la tabella delle dimensioni contiene una descrizione delle dimensioni di business: un **fatto** è un evento di interesse per l'impresa ed è descritto da un insieme di misure, le quali sono attributi a valori continui (tipicamente numerici) che descrivono il fatto da diversi punti di vista. In particolare, si parla si parla di **granularità di un fatto** per indicare il livello di aggregazione a cui un fatto è associato (più i dati sono dettagliati e più bassa ne è la granularità) e di **additività del fatto** per indicare la capacità di una misura di essere aggregata. La tabella dei fatti contiene una chiave esterna per ogni tabella delle dimensioni e i dati in essa contenuti devono essere considerati come dati di riferimento a sola lettura (se una tabella contiene solo chiavi esterne e non contiene misure è anche detta tabella dei fatti senza fatti). Una volta **progettate le tabelle dei fatti**, occorre **progettare le tabelle delle dimensioni**.

Lo **star-schema** può essere usato per incrementare le prestazioni delle query denormalizzando le informazioni, relative ad una dimensione di analisi, in una singola tabella; la tecnica appena illustrata è efficiente nel caso di informazioni accedute frequentemente, se evita operazioni ripetute di join. Esistono, infine, alcune varianti dello star-schema; in particolare, se si normalizzano le tabelle delle dimensioni, si consente a queste di avere a loro volta delle

dimensioni, parlando di **snowflake-schema**. Un altro **schema, ibrido**, noto con il nome di **starflake-schema**, consente di avere **un mix di schemi a stella** (denormalizzati) e **snowflake** (normalizzati).

TECNOLOGIE PER I SISTEMI DI BASI DI DATI

La struttura fondamentale di un tipico DBMS è schematizzata di seguito; in particolare, si può osservare come ogni DBMS riceva in ingresso delle generiche istruzioni SQL, generate da applicazioni o da utenti del sistema. Ogni query, quindi, viene analizzata attraverso un apposito modulo, denominato Gestore delle Query, ed eventualmente ottimizzata selezionando il piano di esecuzione più efficiente, in termini di operazioni di basso livello richieste per l'esecuzione della query stessa.



La risoluzione di una query in termini di operatori di basso livello necessita di metodi per l'accesso ai dati e per la gestione delle informazioni sui file della base di dati, considerando che ogni file può essere visto (logicamente) come una sequenza di record o, più in generale, come una collezione di pagine di record. Il Gestore dei Metodi di Accesso e dei File, a sua volta, necessita di informazioni legate a come il buffer dei dati è gestito nella parte di memoria centrale demandata a contenere un sottoinsieme delle pagine di record in memoria di massa. Come e quando trasferire queste informazioni da memoria centrale a memoria di massa è compito del Gestore del Buffer di Memoria, a sua volta connesso ad uno strato software che permette di implementare le funzioni di lettura, scrittura, allocazione e rilasciamento delle pagine su disco, detto Gestore dello Spazio su Disco. Lo schema mette anche in evidenza due moduli importanti per un DBMS: il Gestore della Concorrenza ed il Gestore dell'Affidabilità, che interviene in caso di guasti del sistema, garantisce l'esecuzione corretta delle transazioni sulla base delle proprietà ACID e filtra opportunamente le richieste degli utenti sulla base di dati tramite un meccanismo di lock. Infine, si fa notare che, sebbene non evidenziati nello schema, sono presenti altri due moduli: il Gestore dell'Integrità, che assicura la soddisfazione dei vincoli di integrità a valle di operazioni

di modifica dello stato della base di dati, e il **Gestore degli Accessi**, che **garantisce l'accesso alle informazioni della base di dati e la compatibilità delle operazioni solo agli utenti autorizzati**.

Per gestire in modo persistente le grosse quantità di informazioni tipiche di un sistema di basi di dati, **il DBMS deve memorizzare i relativi dati su dispositivi di memoria di massa**, la memoria centrale non è sufficiente, sebbene queste informazioni vi debbano comunque essere spostate quando richieste dall'elaborazione. **La struttura dati tipica per la memorizzazione delle informazioni di una base di dati è quella dei file di record** e la loro comprensione permette la progettazione di operazioni più efficienti; inoltre, l'**unità di informazione che viene letta o scritta da un dispositivo di memorizzazione di massa è detta anche pagina ed è tipicamente di 4 o 8 KB**. L'operazione di lettura o di scrittura di una pagina ha **un costo associato** (detto **costo di I/O**) che risulta essere di **gran lunga il più pesante**, dal punto di vista dei tempi di elaborazione, **rispetto ai costi delle altre operazioni su una base di dati**.

Sulla base di quanto detto finora, **un file è una sequenza di record, ognuno dei quali formato da uno o più campi e identificato da un unico identificatore**, che permette di individuare sul dispositivo l'**indirizzo fisico della pagina contenente il record su disco**. Si consideri una tipica tabella relazionale, come quella mostrata di seguito, e si assuma che ogni sua tupla sia mappata in un **record di un file gestito dal sistema operativo**; quando un utente richiede una tupla dal DBMS, quest'ultimo mappa un **record logico** in un **record fisico** e lo associa ad una pagina in memoria primaria gestita, come accennato precedentemente, dal **Gestore del Buffer di Memoria**. Il **record fisico** è a tutti gli effetti l'**unità di trasferimento tra la memoria primaria e quella secondaria** e lo si può considerare un **insieme di uno o più record logici**, il cui numero viene definito **fattore di blocco**.

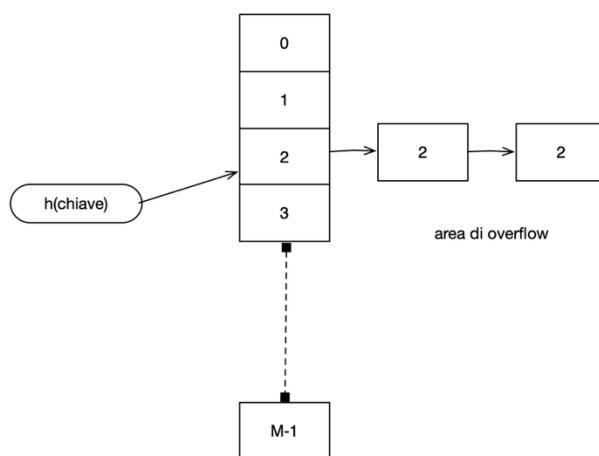
ID	Nome	Cognome	Stipendio
003	Antonio	De Aurelis	5000
001	Marco	Marchi	2000
005	Antonio	Di Padova	3000
004	Francesco	Di Sales	1000

L'ordine con cui i record sono memorizzati e acceduti nel file dipende dalla struttura dei file in memoria secondaria. A tal proposito, esistono diverse modalità di organizzazione per i dati all'interno di un file: le principali a cui si farà riferimento sono i **file non ordinati** (o file heap), i **file ordinati** (o file sequenziali) e i **file ad accesso calcolato** (o hash file). In questo contesto, si definisce **metodo di accesso** l'insieme di tecniche necessarie per memorizzare e per recuperare i record da un file, ovviamente **collegato al modo in cui un file è organizzato**.

La struttura di file più semplice è quella di un **file heap**, in cui i record sono memorizzati su file in modo del tutto casuale. Un **nuovo record** che viene inserito nel file viene di solito posizionato nell'ultima pagina del file ma se lo spazio è insufficiente viene **creata una nuova pagina** e il record viene aggiunto in coda. È chiaro che in questo modo le operazioni di inserimento nel file sono molto efficienti; di contro, non essendovi alcun ordine di memorizzazione, l'unica possibilità di recupero dei record è quella basata su una ricerca lineare, rendendo l'operazione estremamente lenta. L'operazione di **cancellazione del record** richiede, intrinsecamente, la ricerca nel file del record da cancellare; tipicamente il record viene marcato come logicamente cancellato e non viene più utilizzato (negli inserimenti successivi si accede al file sempre in coda). Si comprende facilmente come questo metodo di organizzazione deteriora progressivamente il file quando ci sono frequenti cancellazioni; in tal caso, occorre provvedere ad una riorganizzazione del file per recuperare gli spazi lasciati inutilizzati dalle cancellazioni.

Quando un file è organizzato o sequenziale, i record sono ordinati su uno o più campi (quando la chiave del file coincide con i capi di ordinamento si parla di chiavi di ordinamento). Considerando la tabella precedente, si faccia l'ipotesi che i record su file siano ordinati secondo il campo ID: l'operazione di ricerca di un record a partire dal campo chiave potrà fare uso di tecniche di ricerca binaria, notoriamente più efficienti delle tecniche di ricerca lineare ($O(\log n) < O(n)$, con n dimensione del file). Di contro, le operazioni di inserimento e cancellazione richiedono maggiori passi computazionali rispetto ai file disordinati, dovendo mantenere sempre l'ordine tra i record; l'inserimento, infatti, richiede prima l'individuazione della posizione in cui inserire il record e, successivamente, la liberazione di spazio (ad esempio, traslando a destra tutti i record che seguono). Solo a questo punto, è possibile fare inserimenti; se non c'è spazio nella pagina, occorrerà creare una nuova pagina e spostare uno o più record in essa. A causa della complessità evidenziata dell'operazione, in caso di frequenti inserimenti potrebbe essere utile utilizzare un file temporaneo in cui effettuare inserimenti non ordinati; poi, opportuni algoritmi di merge sort possono essere usati ad intervalli regolari di tempo per riottenere un unico file ordinato. Quanto appena detto può essere analogamente riadattato per la cancellazione, che richiede l'organizzazione del file per la rimozione dello spazio lasciato libero.

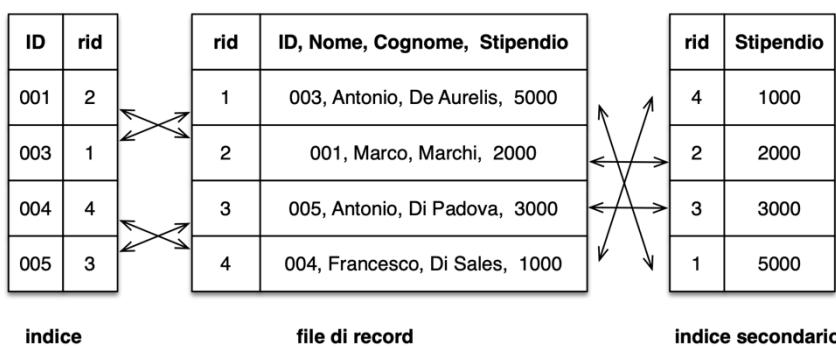
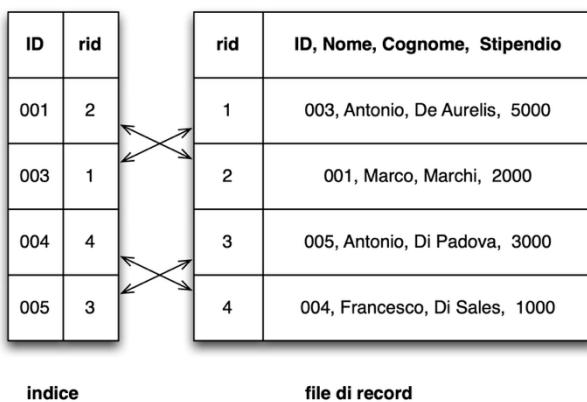
In un file di hash la posizione del record nel file viene calcolato attraverso un'apposita funzione di hash; la funzione in questione prende in ingresso uno o più campi del record e restituisce una posizione (se i campi di hash sono anche chiavi del file, si parla di chiavi di hash). La funzione di hash deve essere scelta in modo tale da distribuire i record nel file in modo uniforme, anche se apparentemente casuale. Una tecnica usuale che implementa la funzione di hash consiste nel convertire i valori delle chiavi di hash in numeri interi e nel sommare il numero ottenuto ad un opportuno offset (tecnica di folding), oppure effettuando una divisione in modulo per individuare la posizione (tecnica della divisione in modulo). In tutti i casi, tuttavia, dal momento in cui il numero dei possibili valori calcolati dei campi di hash è molto maggiore del numero di posizioni disponibili nel file, non è possibile garantire un indirizzo univoco ad ogni record: tipicamente, in questo caso ogni indirizzo generato dalla funzione di hash è associato ad un bucket su memoria di massa contenente più record. All'interno del bucket i record sono memorizzati secondo l'ordine di arrivo e, in caso di record sinonimi (cioè che hanno lo stesso indirizzo del bucket calcolato dalla funzione di hash) si parla di collisioni: in presenza di collisioni, se il bucket è pieno occorre inserire il record in una nuova posizione. La gestione delle collisioni viene di solito effettuata con tecniche più o meno sofisticate (che complicano però le prestazioni di tale tipo di file), come l'uso di un'area di overflow in cui posizionare record sinonimi (secondo l'ordine di arrivo o tramite una lista concatenata collegata ai vari bucket).



Per avere accesso alle informazioni all'interno di un file in modo più efficiente, si può far riferimento ad un indice di accesso: una struttura dati che permette di organizzare in modo

opportuno i record al fine di rendere efficiente il recupero dell'informazione, attraverso una chiave di ricerca sull'indice. Nei DBMS, un indice è una struttura dati ordinata che permette di localizzare un particolare record in un file dati in modo veloce, diminuendo i tempi di risposta di una query di un utente. È chiaro che, pur non essendo necessari al funzionamento di un DBMS, gli indici ne migliorano le prestazioni; ad ogni file, viene dunque associato un file di indice contenente la struttura dati dell'indice stesso (di solito formata da coppie chiave di ricerca e identificatore di un record). Un indice **velocizza le sezioni sui campi che compongono la chiave di ricerca per l'indice**, chiave che può essere formata da qualunque sottoinsieme dei campi di una relazione (si noti che la chiave di ricerca **non coincide necessariamente con una chiave primaria della relazione**).

Riprendendo l'esempio della tabella precedente, al fine di memorizzare queste informazioni in un file non ordinato, **si possono registrare le posizioni dei record in un file ordinato sul campo ID**, necessitando così di **due file: uno (disordinato) contenente i record dati identificati ognuno dal proprio identificatore (record_identifier, rid)** e **uno (ordinato) contenente i record dell'indice ordinati sul campo ID**, come si evince dalla figura seguente. Nel caso in cui si vogliano fare **query sul campo stipendio, può essere utile aggiungere un altro file di indice, ordinato sul campo stipendio** (file indice ausiliario o secondario).



In generale, si definisce **data entry** il record memorizzato in un file indice: da questo punto di vista, un indice risulta essere una collezione di data entry. A seconda di come è fatto il data entry, esistono diversi tipi di indice:

- Il data entry è costituito da un intero record di dati, con la sua chiave di ricerca, allora l'indice è un caso particolare di organizzazione dei file;
 - Il data entry è una coppia formata da (**chiave**, **rid**), come nell'esempio precedente, allora l'indice è completamente indipendente dall'organizzazione del file con i dati (heap o ordinato);

- Il **data entry** è una coppia (**chiave, lista_di_rid**), dove **lista_di_rid** è una lista di identificatori di record dati aventi un particolare valore della chiave di ricerca, allora l'indice è indipendente dall'organizzazione del file, permettendo una migliore gestione dello spazio.

In letteratura, sono stati presentati diversi tipi di indici:

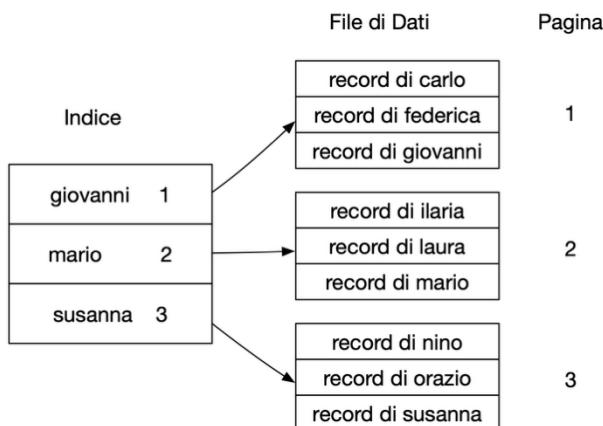
- **Indice primario**, si tratta di un indice costruito su un file sequenziale ordinato su una chiave (in altre parole, un indice su di un insieme di campi che include la chiave primaria di una relazione);
- **Indice secondario**, si tratta di un indice costruito su una chiave non primaria di una relazione;
- **Indice clustering**, si tratta di un indice costruito su un campo non chiave, di modo che ad ogni valore di tale campo corrispondono più record (detti cluster di record).

Un file può avere un solo indice primario, un solo indice di clustering e diversi indici secondari; inoltre, si distinguono gli indici sparsi, ovvero gli indici che hanno un record di indice solamente per alcuni valori della chiave nel file, dagli indici densi, ovvero gli indici che hanno un record per ogni valore di chiave di ricerca nel file.

Di seguito sono approfonditi i principali indici di cui sono dotati i DBMS commerciali:

- **Indexed Sequential Files** (File sequenziali indicizzati)

Un file sequenziale indicizzato è un **file ordinato con un indice primario** (il più famoso quello definito da IBM con la struttura ISAM, Indexed Sequential Access Method, fortemente dipendente dall'hardware dello storage e la cui evoluzione ha dato luogo al VSAM, Virtual Sequential Access Method) e, di solito, **organizzato in un'area di memorizzazione primaria**, in uno o più indici separati, e in un'area di overflow:

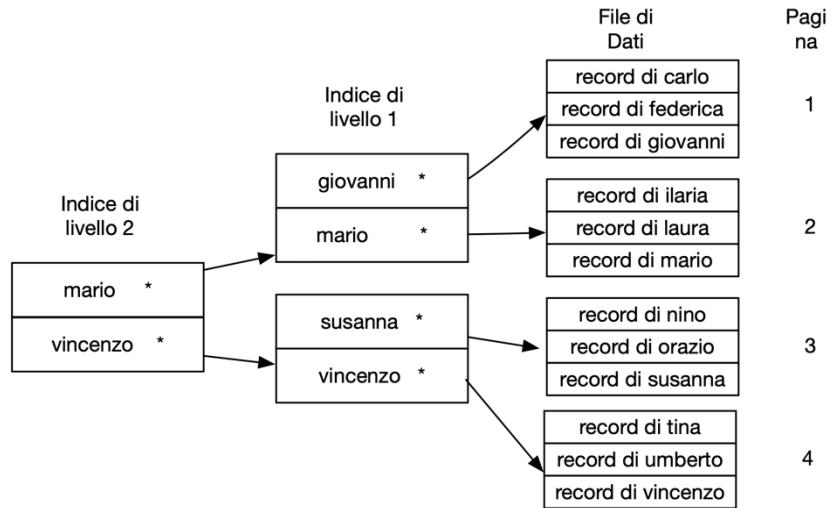


Si noti che **buona parte di un indice primario è memorizzato in memoria principale**; essendo ordinato, si possono sfruttare per le ricerche algoritmi di ricerca binaria. Allo stesso tempo, se il file è sottoposto a continue operazioni di inserimento e di cancellazione, risulta complesso il mantenimento dell'ordine nel file di dati e nel file di indice.

- **Indici multilivello**

Quando ci si trova in presenza di **file di indici molto grandi**, anche in presenza di algoritmi di ricerca binaria, **il tempo di ricerca aumenta**: se un indice è costruito su k pagine, il tempo di ricerca sarà **proporzionale**, come noto, a $\log_2 k$. Per diminuire i tempi, può essere utile l'introduzione di un

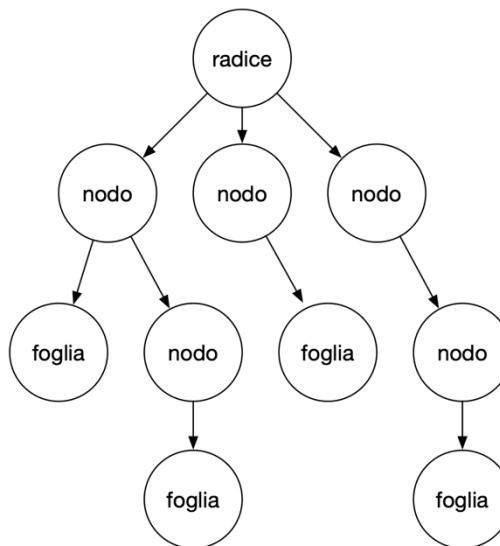
indice multilivello; intuitivamente, il file comprendente k pagine viene diviso in un numero di indici di dimensione molto minore di k e, per poter accedere a tali indici, si può pensare di utilizzare un indice degli indici. Si consideri, ad esempio, l'indice a due livelli di seguito illustrato, organizzato su un ordinamento alfabetico sul campo Nome:



Per accedere ad un'informazione, ad esempio al record individuato dal nome 'federica', si parte dall'indice di livello 2, accedendo alla pagina il cui nome è minore o uguale a 'federica', in questo caso 'mario'. Tale record contiene un indirizzo all'indice di primo livello, che permette di continuare la ricerca in maniera ricorsiva fino ad accedere alla pagina dove è presente il record cercato nel file dati.

- Indici B^+ Tree

La struttura dati più usata per gestire gli indici nei DBMS commerciali è, indubbiamente, la struttura ad albero. L'albero è una struttura dati gerarchica formata da nodi ed archi; ogni nodo, eccetto il nodo radice, ha un unico nodo padre e può avere più nodi figli, mentre un nodo senza figli è detto nodo foglia.

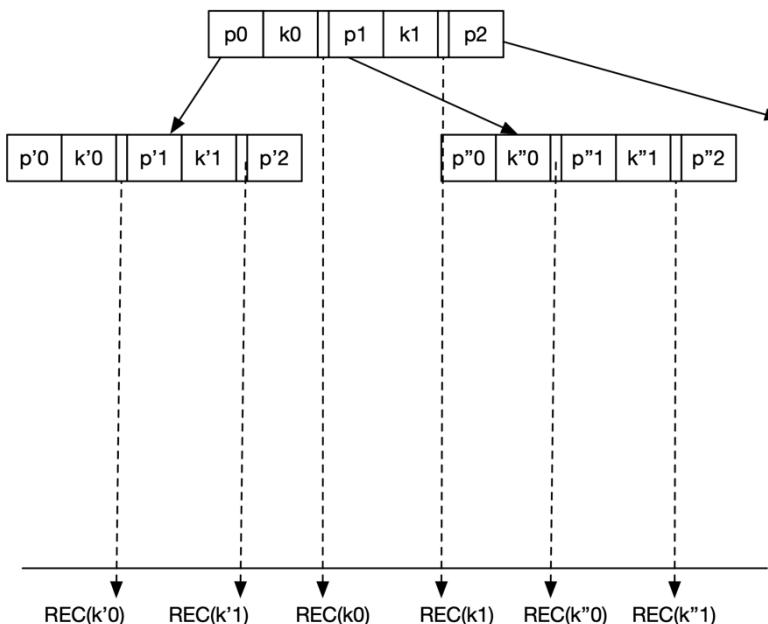


Un albero è detto bilanciato (Balanced Tree o B-Tree) se la sua profondità (ovvero il massimo numero di livelli tra il nodo radice e una foglia) è la stessa per ogni nodo foglia. In generale,

mantenere un albero bilanciato è molto importante ai fini dell'efficienza di una ricerca, garantendo che **nessun nodo si trovi a profondità troppo elevata rispetto agli altri** (altrimenti sarebbero richiesti troppi accessi ai blocchi durante la ricerca). A tal fine, è **necessario disporre di algoritmi che**, in fase di inserimento o di cancellazione di un nuovo record, **provvedano ad aggiornare l'albero cercando di mantenere tutti i nodi foglia allo stesso livello.**

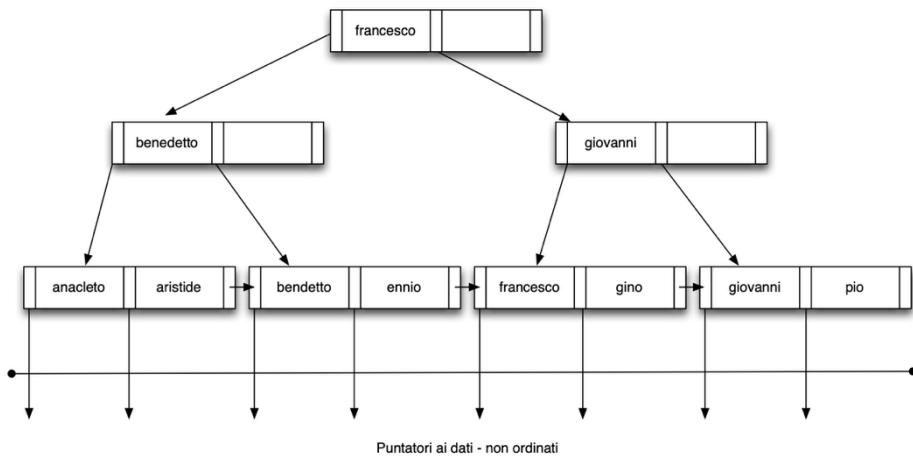
Di seguito sono esaminate più nel dettaglio le **caratteristiche di un albero bilanciato**; un B-Tree di ordine p è definito come segue:

- Ogni nodo dell'albero contiene $\langle P_1, K_1, P_{K_1}, P_2, K_2, P_{K_2}, \dots, P_q \rangle$, con $q \leq p$;
 - $K_1 < K_2 < \dots < K_{q-1}$ con, K_i una chiave;
 - P_{K_i} è un puntatore ai dati relativi alla chiave K_i , P_i il puntatore ad un sottoalbero che contiene tutti i valori di chiave $K < K_i$;
 - P_q punterà ad un sottoalbero con valori di chiavi $K > K_{q-1}$;
- Ogni nodo può contenere al massimo p puntatori dell'albero (definizione di ordine di un albero);
- Ogni nodo, tranne la radice e i nodi foglia, deve contenere almeno $p/2$ puntatori dell'albero e il nodo radice contiene almeno 2 nodi figli;
- Tutti i nodi foglia sono allo stesso livello (definizione di albero bilanciato).



Quando **il nodo radice diventa completo**, lo si divide (operazione di **split**) in due nodi di livello 1: **il nodo radice conterrà solo il valore centrale del nodo**, mentre **il resto dei valori sarà equamente diviso tra i due nodi figli**. Quando **uno dei due nodi si riempie**, viene nuovamente eseguita un'operazione di **split** e viene **posto un puntatore al nodo padre**, ponendo in esso i valori di chiave centrale. Quando **il nodo padre si riempie**, viene anch'esso diviso: se la divisione di propaga fino **al nodo radice**, e se anche la radice deve essere divisa, **si crea un nuovo livello dell'albero**. Per quanto riguarda **la cancellazione**, se fa sì che **il nodo sia completo per meno della metà**, allora **il nodo viene fuso** (operazione di **merge**) con uno dei suoi vicini; si noti che **anche la cancellazione può propagarsi fino alla radice**.

La maggior parte delle implementazioni commerciali utilizza una variante del B-Tree, detta B^+ Tree; in essa, i puntatori ai dati sono memorizzati nei nodi foglia dell'albero, i quali, inoltre, sono collegati tra loro (lista collegata):



Un B^+ Tree richiede approssimativamente lo stesso tempo per accedere ad un qualunque record da esso indicizzato e, quindi, assicura che vengono attraversati circa lo stesso numero di nodi prima di arrivare alla radice (il tempo di ricerca è, dunque, funzione della profondità dell'albero). L'indice in questione è un indice denso: ogni record è indicizzato, facendo sì che il file di dati non debba essere ordinato. Ciò che è costoso, in questa implementazione, sono le operazioni di inserimento e cancellazione; inoltre, rispetto al classico B-Tree, è permessa l'implementazione più efficiente di range query (ovvero ricerche su intervalli di valori).

In letteratura, sono stati presentati diverse varianti del B^+ Tree, in particolare il B^* Tree, che risulta essere un B^+ Tree con il vincolo che ogni nodo sia completo per almeno due terzi.

Alcuni DBMS commerciali memorizzano fisicamente assieme più tabelle che hanno in comune colonne e che spesso sono usate assieme. In questo caso, l'accesso al disco può essere effettivamente migliorato: le tabelle comuni sono memorizzate una volta sola e possono essere opportunamente indicizzate usando un indice o un meccanismo di hash (indexed clusters e hash clusters).

Come si è già potuto osservare, il Gestore del Buffer di Memoria è responsabile della gestione efficiente dei buffer che sono usati per trasferire pagine da e verso la memoria secondaria; tali meccanismi sono simili a quelli previsti nei moderni Sistemi Operativi. Si consideri che lo scopo fondamentale del buffer è ridurre il numero di accessi alla memoria secondaria; per buffer, si intende, dunque, un'area di memoria centrale (detta SGA), assegnata dal sistema operativo, gestita dal DBMS e condivisa tra le transazioni, e di solito organizzato in pagine di dimensioni pari o multiple di quelle dei blocchi di memoria secondaria (1KB-100KB). Il gestore del buffer deve, in generale, leggere pagine dal disco e portarle nel buffer fino a che quest'ultimo non si riempie: quando riempito, occorre definire una strategia per decidere come liberare spazio nel buffer per una nuova pagina, che deve, ad esempio, essere letta dal disco; le politiche più spesso utilizzate sono FIFO (First In First Out) e LRU (Last Recently Used, preferibile). È chiaro che un gestore di buffer non legge dal disco pagine che sono già presenti in memoria centrale, sono utilizzate due variabili di stato, entrambe poste inizialmente a zero:

- **Count**, indica quanti programmi stanno utilizzando una certa pagina;
- **Dirty**, indica se la pagina è stata modificata (in gerco, sporcata) o meno.

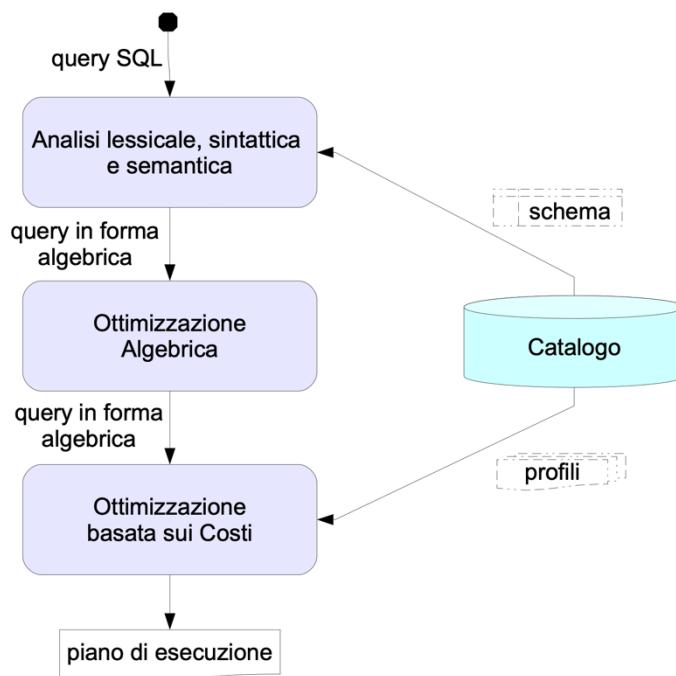
Il **gestore del buffer** ha un funzionamento molto semplice: **riceve dai layer di livello superiore richieste di lettura o di scrittura e le esegue utilizzando il buffer in memoria centrale o, quando non possibile, accedendo alla memoria di massa**. Quando si richiede una pagina dal disco, il gestore del buffer verifica se la pagina è già presente in memoria centrale; in particolare:

1. Se la pagina cercata è presente nel buffer, incrementa Count di 1, se viene modificata, pone Dirty di 1;
2. Se la pagina non è presente, viene scelta nel buffer una pagina libera usando una delle politiche FIFO o LRU e, se Dirty, viene salvata su memoria secondaria;
 - a. Se non esiste alcuna pagina libera, il gestore del buffer si può comportare con politiche di tipo steal o no steal;
3. A questo punto, la pagina richiesta viene trasferita in memoria centrale, con Count pari a 1 e Dirty pari a 0.

Più in dettaglio, **quando le pagine sono scritte su disco, si parla di:**

- **Politica steal**, permette al gestore del buffer di scrivere su disco prima del commit di una transazione (ruba una pagina dalla transazione);
 - Se ciò non avviene, si parla di **politica no steal** e l'operazione viene posta in attesa;
- **Politica force**, assicura che tutte le pagine modificate da una transazione siano immediatamente scritte su memoria di massa non appena la transazione ha fatto il commit;
 - Alternativamente, si parla di **politica no force**.

Ogni DBMS possiede, come già anticipato, **un modulo denominato Gestore delle Query** che ha il compito di effettuare l'ottimizzazione delle query. In particolare, l'ottimizzatore sceglie la strategia esecutiva “migliore” per una data query, proponendo di solito diverse alternative, a partire dall'istruzione SQL e al fine di minimizzare i tempi di risposta del sistema di basi di dati:



- Viene prima effettuata un'**analisi lessicale, sintattica e semantica**, basata sulle informazioni sullo schema della base di dati presenti nel catalogo, per verificare la correttezza della query, di cui viene, successivamente, determinata la relativa forma algebrica (da SQL ad algebra relazionale);

- Viene effettuata **una prima ottimizzazione algebrica** per trasformare la query in ingresso in una forma equivalente a quella di partenza ma che sia più efficientemente eseguibile;
- Viene effettuata **una seconda ottimizzazione basata sui costi** che, sfruttando le informazioni sulle relazioni della base di dati prelevate anch'esse dal catalogo, è in grado di determinare il piano di esecuzione finale (ovvero, la sequenza di operazioni di basso livello che consentono di eseguire la query).

Come già annunciato, l'**ottimizzazione algebrica** si basa sul concetto di **equivalenza algebrica dell'algebra relazionale**. Due espressioni dell'algebra relative ad interrogazioni sulla base di dati sono equivalenti se producono lo stesso risultato qualunque sia l'istanza attuale della base di dati. Nella pratica, ogni DBMS cerca, nella fase di ottimizzazione algebrica, di eseguire espressioni equivalenti a quelle date ma che sono meno costose ai fini dell'esecuzione. L'euristica fondamentale adottata è quella di **anticipare selezioni e proiezioni il prima possibile** (per ridurre le dimensioni dei risultati intermedi) **rispetto alle operazioni di join** (in gergo, **push selections down e push projections down**). Ad esempio, la seguente query:

```
SELECT *
FROM DIPARTIMENTI D JOIN IMPIEGATI I ON D.Cod=I.Dip
WHERE I.Cognome='Moscato'
```

Che corrisponde alla seguente espressione algebrica:

$$\sigma_{I.Cognome='Moscato'}(D \bowtie_{D.Cod=I.Cod} I)$$

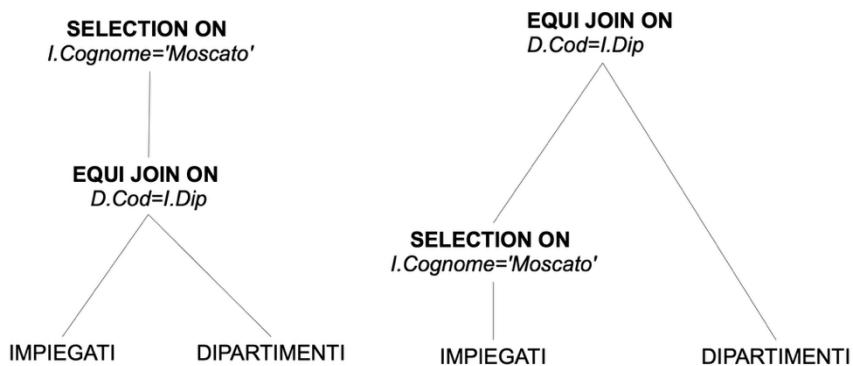
Può essere ottimizzata come segue:

$$\sigma_{\hat{\chi}}(A \bowtie_{\chi} B) \equiv (A \bowtie_{\chi} \sigma_{\hat{\chi}}(B))$$

Con A e B generiche relazioni e $\hat{\chi}$ una condizione di selezione valida sugli attributi della relazione B . Con riferimento alla query iniziale:

$$(D \bowtie_{D.Cod=I.Cod} \sigma_{I.Cognome='Moscato'}(I))$$

Le interrogazioni possono essere, poi, rappresentate attraverso appositi alberi, come di seguito:



Una procedura euristica di ottimizzazione di tipo generale prevede i seguenti passi:

- Decomporre le sezioni congiuntive (AND) in successive selezioni atomiche;
- Anticipare il più possibile le selezioni;

- In una sequenza di selezioni, anticipare le più selettive;
- Combinare prodotti cartesiani e selezioni per formare join;
- Anticipare il più possibile le proiezioni (anche introducendone di nuove).

Una volta trovato per una data query l'albero della sua rappresentazione algebrica più conveniente, il DBMS effettua un'ottimizzazione basata sui costi, sfruttando alcune informazioni sulle relazioni della base di dati; queste ultime, dette profili, contengono una serie di dati quantitativi come cardinalità di ciascuna relazione, dimensioni delle tuple, dimensioni degli attributi, numero di valori distinti degli attributi, valore minimo e massimo di ciascun attributo, ... I profili sono utilizzati nell'ottimizzazione delle query per stimare le dimensioni dei risultati intermedi, in modo da scegliere, di volta in volta, quale sia la migliore sequenza possibile di operazioni di più basso livello (scansione, accesso diretto, ordinamento, join). Una volta determinato il piano di esecuzione di una query, questo può essere memorizzato all'interno del catalogo e richiamato più volte qualora la stessa query necessiti di essere eseguita nuovamente all'interno dell'ambiente DBMS (approccio compile & store); di contro, è anche possibile determinare di volta in volta il piano di esecuzione delle query, senza salvare alcuna informazione nel catalogo (approccio compile & go).

La presenza simultanea di programmi utente che richiedono l'accesso al contenuto di una base di dati rappresenta, come visto, lo scenario tipico dei sistemi OLTP. In un contesto di questo tipo, la mancanza di un protocollo per il controllo della concorrenza potrebbe portare la base di dati in uno stato non consistente (ad esempio, utenti che acquistano più prodotti di quelli effettivamente disponibili), dall'altra parte l'assenza di procedure per il recupero (o recovery) della base di dati in corrispondenza di guasti potrebbe comportare la perdita di informazioni di vitale interesse per l'organizzazione (ad esempio, il numero totale di acquisti della giornata). Per questo motivo, ogni DBMS possiede un modulo per la Gestione della Concorrenza ed uno per la Gestione dell'Affidabilità, che si occupano di garantire l'accesso concorrente alla base di dati, preservandone il contenuto anche in corrispondenza di crash del sistema.

In pratica, un DBMS esegue delle particolari procedure, dette transazioni: una transazione su una base di dati è un'operazione, o una sequenza di operazioni, generata da un utente o da un programma applicativo che legge o aggiorna il contenuto di una base di dati. Quindi, una transazione rappresenta un'unità logica di elaborazione che corrisponde ad una serie di operazioni fisiche elementari (lettura/scritture) sulla base di dati e può coincidere con un intero programma, con una parte di un programma o con un singolo comando (come uno statement SQL di insert o update), che vengono eseguiti in un ambiente DBMS e che costituiscono un'entità atomica (non divisibile) di modifiche fatte allo stato di una base di dati. L'esecuzione di una transazione può avere due possibili esiti: una transazione o termina in uno stato finale previsto dal programma in esecuzione (commit) o porta il sistema ad uno stato precedente all'esecuzione della transazione (abort); nel caso di abort, eventuali azioni ed effetti parziali della transazione devono essere disfatti (undo o rollback), in quanto lascerebbero la base di dati in uno stato di inconsistenza.

Si noti che talora le transazioni possono anche contenere solo interrogazioni; in tal caso, l'atomicità è sempre assicurata, in quanto un'interrogazione non modifica lo stato di una base di dati. Di seguito è proposto un esempio di transazione:

```
UPDATE PRODOTTI SET Quantità=Quantità-y WHERE Codice=x;
INSERT INTO CARRELLO(Cliente, Prodotto, Qta, Data);
VALUES (z,x,y, '13-Gen-2026 20:30:54');
```

Le transazioni devono possedere alcune proprietà fondamentali, dette **ACID** (proposte da Haerder e Reuter nel 1983):

- **Atomicità** (o proprietà del “o tutto o nulla”), una transazione è atomica se è eseguibile o nella sua interezza o non è eseguibile per niente;
- **Consistenza**, una transazione è consistente se causa una trasformazione di uno stato consistente della base di dati in un altro stato consistente (un DBMS, poi, deve assicurare che tutti i vincoli definiti sulla base di dati siano soddisfatti durante l'esecuzione di transazioni);
- **Isolamento**, una transazione è isolata se è eseguibile in modo indipendente dalle altre (ovvero, gli effetti parziali di transizioni incomplete non devono essere visibili alle altre transazioni);
- **Durability** (o persistenza), una transazione è persistente se, una volta terminata con un commit, i suoi effetti sono registrati in modo permanente nella base di dati e non possono/devono essere persi per alcun motivo.

Più astrattamente, **una transazione coinciderà con un blocco di istruzioni caratterizzato da un inizio** (begin transaction, esplicitato in SQL) e **da una fine** (end transaction, di norma non esplicitato in SQL) e **al cui interno deve essere eseguito una e una sola volta almeno uno dei seguenti comandi**:

- **commit**, per terminare correttamente e rendere persistenti le azioni della transazione;
- **rollback**, per abortire la transazione.

Con riferimento all'esempio precedente:

```
begin transaction
    UPDATE PRODOTTI SET Quantità=Quantità-y WHERE Codice=x;
    INSERT INTO CARRELLO(Cliente, Prodotto, Qta, Data);
    VALUES (z,x,y, '13-Gen-2026 20:30:54');
commit
```

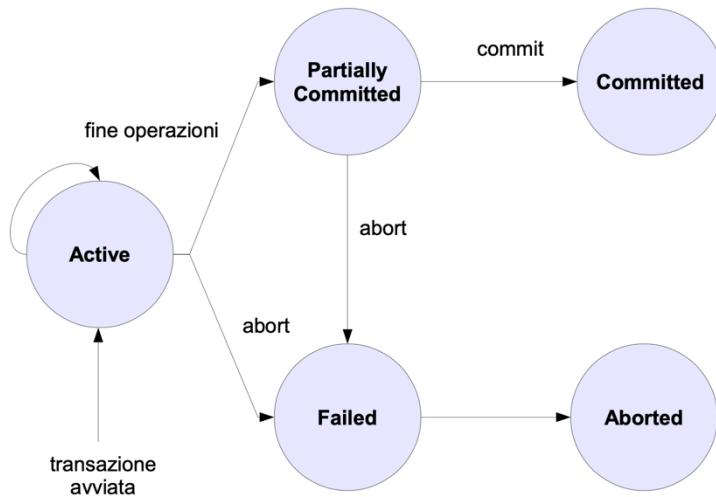
All'inizio e alla fine della transazione il sistema deve essere in uno stato consistente; invece, durante l'esecuzione della transazione stessa il sistema può temporaneamente essere in uno stato inconsistente. A tal proposito, i vincoli di integrità dei dati possono essere “controllati” in maniera differita, ovvero a valle della richiesta di commit della transazione, causando, in caso di violazione, un abort della stessa, o in maniera diretta, ovvero durante l'esecuzione della transazione stessa evitandone l'abort.

Inoltre, è possibile modificare il codice della transazione introducendo controlli diretti sulla violazione dei vincoli:

```
begin transaction
    SELECT Quantità-y INTO Q FROM PRODOTTI WHERE Codice=x;
    IF(Q>0) THEN
        UPDATE PRODOTTI SET Quantità=Quantità-y WHERE Codice=x;
        INSERT INTO CARRELLO(Cliente, Prodotto, Qta, Data);
        VALUES (z,x,y, '13-Gen-2026 20:30:54');
        commit
    ELSE
        rollback
    END IF
```

In realtà, le interfacce dei programmi applicativi, implementando appositi controlli, possono evitare queste situazioni, consentendo ad un utente di selezionare solo una quantità minore o uguale di quella effettivamente disponibile in magazzino; in tal caso, la consistenza può essere violata solo dall'esecuzione concorrente di più transazioni.

Di seguito è mostrato il **diagramma degli stati di una transazione in un ambiente DBMS**:



Si può notare che, oltre agli stati **Active** (in cui la transazione viene eseguita), **Committed** (in cui la transazione si trova dopo aver eseguito correttamente il commit) e **Aborted** (in cui la transazione si trova dopo essere stata abortita), sono stati introdotti due stati intermedi:

- **Partially Committed**, si verifica quando viene eseguita l'ultima istruzione di una transazione e codifica lo stato in cui la transazione potrebbe essere abortita (portandosi, in tal caso, nello stato **Failed**, in attesa di essere abortita) o completata con successo (portandosi, in tal caso, nello stato **Committed**);
- **Failed**, si verifica quando la transazione non può essere, per qualche motivo, committata in maniera corretta o se la transazione viene abortita mentre si trova in esecuzione.

Infine, si vuole far notare come, con riferimento all'architettura di un DBMS precedentemente descritta per ciò che concerne la gestione delle transazioni, si ha che:

- **L'atomicità e persistenza saranno garantite dal Gestore dell'Affidabilità;**
- **L'isolamento sarà garantito dal Gestore della Concorrenza;**
- **La consistenza sarà garantita dal Gestore dell'Integrità a tempo di esecuzione** (con il supporto del compilatore del DDL).

È stato già illustrato come uno degli obiettivi principali di un sistema di basi di dati sia quello di supportare l'accesso concorrente da parte di utenti ed applicazioni a dati condivisi; in particolare, gli accessi concorrenti, di cui almeno uno prevede un aggiornamento, potrebbero provocare problemi di integrità e consistenza dei dati, noti anche con il nome di anomalie. Una soluzione banale preverrebbe la prescrizione di un accesso “serializzato” alla base di dati, in cui le operazioni delle varie transazioni si succedono temporalmente in sequenza senza causare anomalie; tuttavia, è impensabile nei sistemi OLTP, con decine o centinaia di transazioni generate al secondo, prevedere un accesso di tipo seriale, ne rallenterebbe il tempo di risposta.

Un DBMS deve fornire un apposito modulo, il Gestore della Concorrenza, per il controllo della concorrenza, il cui obiettivo sia massimizzare il numero di transazioni per secondo (detto

throughput o tps), **minimizzando così i tempi medi di risposta**. Ciò significa eseguire più transazioni in concorrenza, alternando l'esecuzione di operazioni di una transazione con quella di operazioni di altre transazioni (**interleaved execution**), in modo del tutto analogo a come avviene nei sistemi operativi, sfruttando il fatto che, mentre una transazione è in attesa del completamento di un'operazione di I/O, un'altra può utilizzare la CPU. L'esecuzione alternata di più transazioni concorrenti, però, non previene il verificarsi di possibili anomalie: il Gestore della Concorrenza dovrà, quindi, implementare un'apposita politica in grado di prevenire la maggior parte delle anomalie.

In questo modo, il **Modello di Transazione** prevede un solo livello di controllo in cui il comando **begin transaction (bot)** dichiara l'inizio della transazione, il comando **commit** indica il raggiungimento di un nuovo stato consistente e il comando **rollback** un abort. Le singole operazioni di una transazione sono modellate attraverso operazioni di `read(x, y)` e `write(x, y)`, dove x rappresenta un oggetto astratto della base di dati (come, ad esempio, campi di tabelle) e y il valore letto o da scrivere; inoltre, le singole operazioni sono etichettate attraverso l'istante di tempo in cui avvengono. Nella pratica, viene effettuata un'esemplificazione delle operazioni in memoria centrale su tali oggetti, ignorando che le operazioni richiedono l'intera pagina dove risiedono i dati. Un esempio di Modello di Transazione è il seguente:

```
t1: bot
t2:   read(qtaPx, A)
t3:   A = A - y
t4:   write(qtaPx, A)
t5:   B = z
t6:   write(clienteCxz, B)
t7:   C = x
t8:   write(prodCxz, C)
t9:   D = y
t10:  write(qtaCxz, D)
t11:  E = '13-Jan-2015 20:30:54'
t12:  write(dataCxz, E)
t13: commit
```

Supponendo che, nel momento della procedura di acquisto di un prodotto, sia automaticamente creato per ogni utente un record nel carrello con la data di sistema (con NULL in corrispondenza della quantità già acquistata), la relativa transazione potrebbe semplificarsi come segue:

```
t1: bot
t2:   read(qtaPx, A)
t3:   A = A - y
t4:   write(qtaPx, A)
t5:   write(qtaCxz, y)
t6: commit
```

Per perdita di aggiornamento (Lost Update) si intende l'evento che si verifica quando un'operazione di aggiornamento di una transazione si sovrappone a quella di un'altra che stava aggiornando lo stesso oggetto (prima che quest'ultima abbia effettuato il commit), annullandone nella pratica gli effetti. Ad esempio:

Time	\nwarrow	\nwarrow	$qtaP^+$
t_1	bot		100
t_2	$read(qtaP_x, A)$	bot	100
t_3	$A = A - 50$	$read(qtaP_x, A)$	100
t_4	$write(qtaP_x, A)$	$A = A - 30$	50
t_5	$write(qtaC_{xz_1}, 50)$	$write(qtaP_x, A)$	70
t_6	commit	$write(qtaC_{xz_2}, 30)$	70
t_7		commit	70

Per **lettura sporca (Dirty Read)** si intende l'evento che si verifica quando una transazione, durante la sua esecuzione, legge un valore “sporco” di un oggetto, ovvero modificato da un'altra transazione che poi viene abortita. Si consideri la situazione in cui due transazioni di acquisto dello stesso prodotto, generate da due utenti diversi, si sovrappongono temporalmente. Ad esempio:

Time	\nwarrow	\nwarrow	$qtaP^+$
t_1	bot		100
t_2	$read(qtaP_x, A)$		100
t_3	$A = A - 50$		100
t_4	$write(qtaP_x, A)$		50
t_5	$write(qtaC_{xz_1}, 50)$	bot	50
t_6	...	$read(qtaP_x, A)$	50
t_7	...	$A = A - 30$	50
t_8	...	$write(qtaP_x, A)$	20
t_9	rollback	$write(qtaC_{xz_2}, 30)$	70
t_{10}		commit	70

Per **lettura inconsistente (Inconsistent Read)** si intende l'evento che si verifica quando una transazione legge in due diversi istanti della sua esecuzione due valori differenti dello stesso oggetto che viene, intanto, modificato da un'altra transazione. Ad esempio:

Time	\nwarrow	\nwarrow	$qtaP^+$
t_1	bot		20
t_2	$read(qtaP_x, A)$		20
t_3	...		20
t_5	...	bot	20
t_6	...	$read(qtaP_x, A)$	20
t_7	...	$A = A - 20$	20
t_8	...	$write(qtaP_x, A)$	0
t_9	...	$write(qtaC_{xz}, 30)$	0
t_{10}	...	commit	0
t_{11}	$read(qtaP_x, A)$		0
t_{12}	...		0
t_{13}	commit		0

Per **aggiornamento fantasma (Ghost Update)** si intende l'evento che si verifica quando una o più operazioni di aggiornamento di una transazione che riguardano due o più oggetti della base di

dati, il cui valore è correlato (ad esempio, causa presenza di un vincolo di integrità), **non vengono correttamente visualizzate correttamente da altre transazioni temporalmente sovrapposte**. Per queste ultime, è come se alcuni degli aggiornamenti non siano mai stati effettuati sulla base di dati e per questo motivo sono dette fantasmi.

Time	\wedge_1	\wedge_2	$qtaP_{x_1}$	$qtaP_{x_2}$
t_1	bot		100	100
t_2	$read(qtaP_{x_1}, A)$		100	100
t_2	...	bot	100	100
t_3	...	$read(qtaP_{x_1}, A)$	100	100
t_4	...	$A = A - 30$	100	100
t_5	...	$write(qtaP_{x_1}, A)$	70	100
t_6	...	$write(qtaC_{x_1z}, 30)$	70	100
t_7	...	$read(qtaP_{x_2}, B)$	70	100
t_8	...	$B = B - 30$	70	100
t_9	...	$write(qtaP_{x_2}, B)$	70	70
t_{10}	...	$write(qtaC_{x_2z}, 30)$	70	70
t_{11}	...	commit	70	70
t_{12}	$read(qtaP_{x_2}, B)$		70	70
t_{13}	$qtaP_{x_1} \neq qtaP_{x_2}$		70	70
t_{14}	commit		70	70

Ogni transazione coincide, come già anticipato, con una sequenza temporale di azioni di lettura/scrittura su oggetti della base di dati; ad esempio:

$$T_1: r_1(X), w_1(X), r_1(Y), r_1(Z), w_1(Z)$$

$$T_2: r_2(X), r_2(Y), r_2(Z), w_2(Y)$$

Nella pratica viene omesso qualsiasi riferimento alle operazioni di manipolazione in memoria da parte della transazione, così come sono omessi il comando di **bot**, **commit** e **rollback**. Le transazioni avvengono concorrentemente e, pertanto, le operazioni di lettura e scrittura vengono richieste in istanti successivi da varie transizioni; l'obiettivo di un **protocollo di controllo di concorrenza (CdC)** è quello di eseguire le singole operazioni delle varie transazioni in un ordine tale da evitare il verificarsi di anomalie.

Per **schedule** si intende una sequenza di operazioni di lettura/scrittura generate da un insieme di transazioni concorrenti che preserva l'ordine temporale di esecuzione delle operazioni di ciascuna transazione. Uno **scheduler** è un sistema che accetta o rifiuta o riordina le operazioni richieste dalle transazioni; ad esempio, per T_1 e T_2 , due possibili schedule sono:

$$S_1: r_1(X), r_2(X), w_1(X), r_1(Y), r_1(Z), w_1(Z), r_2(Z), r_2(Y), w_2(Y)$$

$$S_1: r_2(X), r_1(X), w_1(X), r_1(Y), r_2(Z), r_2(Y), w_2(Y), r_1(Z), w_1(Z)$$

Vengono ignorate le transazioni che vanno in **abort**, rimuovendo tutte le loro azioni dagli schedule in cui sono inclusi. Uno **schedule seriale** è un particolare schedule in cui le operazioni di ciascuna transazione sono eseguite in maniera consecutiva senza la sovrapposizione di operazioni generate da altre transazioni, mentre uno **schedule serializzabile** è uno **schedule non seriale** che, però, produce gli stessi risultati (in termini di aggiornamento dello stato di una base di dati) di uno **schedule seriale**.

Uno schedule seriale è quello seguente:

$$S_3: r_1(X), w_1(X), r_1(Y), r_1(Z), w_1(Z), r_2(X), r_2(Y), r_2(Z), w_2(Y)$$

Mentre uno schedule serializzabile:

$$S_4: r_1(X), w_1(X), r_2(X), r_2(Y), r_2(Z), w_2(Y), r_1(Y), r_1(Z), w_1(Z)$$

Infatti, l'oggetto X è usato solo dalla prima transazione; pertanto, dopo aver utilizzato l'oggetto in questione, la prima transazione può lasciare spazio alle operazioni della seconda transazione prima di riprendere la propria esecuzione.

Un **Gestore della Concorrenza** dovrà implementare, quindi, **apposite politiche di controllo di concorrenza (scheduler) capaci di produrre schedule serializzabili per le transazioni in esecuzione corrente**. L'obiettivo della teoria del controllo di concorrenza è, quindi, **individuare classi di schedule serializzabili che siano sottoclassi degli schedule possibili e la cui proprietà di serializzabilità sia verificabile a basso costo**.

Nella pratica, i DBMS implementano apposite tecniche di controllo di concorrenza che garantiscono direttamente la serializzabilità degli schedule generati in ambienti concorrenti. Tali tecniche si suddividono in due classi principali:

- **Metodi basati su lock**

Ogni oggetto della base di dati è protetto da un apposito lock e ogni transazione che vuole effettuare una lettura di un oggetto deve “richiederne” l'autorizzazione attraverso un'operazione di **`read_lock(x)`**; solo una volta acquisito il lock, la transazione può effettuare la lettura. Il lock in lettura su un dato oggetto, inoltre, può essere condiviso da più transazioni. Ogni transazione che vuole effettuare una scrittura su un oggetto deve “richiederne” l'autorizzazione attraverso un'operazione di **`write_lock(x)`**; solo una volta acquisito il lock, la transazione può eseguire la scrittura. Il lock in scrittura su un dato oggetto è esclusivo, può essere acquisito da una sola transazione per volta.

Quando una stessa transazione prima legge e poi scrive un oggetto, può richiedere subito un lock esclusivo in scrittura o chiedere prima un lock condiviso in lettura e, successivamente, uno esclusivo (lock escalation). Una volta che una transazione ha ottenuto il lock in lettura e scrittura su un oggetto e terminata l'operazione, deve necessariamente rilasciarlo attraverso un'operazione di **`unlock`**. Ogni oggetto della base di dati si può trovare in uno dei seguenti stati: libero (**`free`**), bloccato in lettura (**`r_locked`**) o bloccato in scrittura (**`w_locked`**).

Il gestore della concorrenza possederà una componente particolare, detta **lock manager**, che riceve le varie richieste di lock dalle transazioni e le accoglie o rifiuta, sulla base della tavola dei conflitti, dove per ogni richiesta di lock su un oggetto e stato dello stesso vengono riportati l'esito della richiesta (Si/No) e il nuovo stato.

Stato dell'Oggetto			
Richieste	<code>free</code>	<code>r_locked</code>	<code>w_locked</code>
<code>read_lock</code>	(Si, <code>r_locked</code>)	(Si, <code>r_locked</code>)	(No, <code>w_locked</code>)
<code>write_lock</code>	(Si, <code>w_locked</code>)	(No, <code>r_locked</code>)	(No, <code>w_locked</code>)
<code>unlock</code>	errore	(Si, <code>free/r_locked</code>)	(Si, <code>free</code>)

La granularità del lock può essere variegata: è possibile imporre dei lock su interi file della base di dati, su particolari pagine, su record o su campi di un record. Per **Lock a Due Fasi** (o 2PL, 2 Phase Locking) si intende il processo di locking per il quale si richiede che ogni transazione acquisisca mediante richieste di **read_lock** e **write_lock** (in una prima fase, detta crescente) tutti i lock sugli oggetti su cui dovrà effettuare operazioni di lettura/scrittura. Solo dopo aver terminato tutte le operazioni, la transazione rilascerà (in una seconda fase, detta decrescente) tutti i lock precedentemente acquisiti mediante le primitive di **unlock**. In altri termini, i vincoli imposti da 2PL sono:

1. Ogni transazione deve proteggere tutte le letture e scritture con lock;
2. Una transazione, dopo aver rilasciato un lock non può acquisirne altri.

Time	\sqsubset	\sqsupset	$qtaP^+$
t_1	bot		100
t_2	$read(qtaP_x, A)$	bot	100
t_3	$A = A - 50$	$read(qtaP_x, A)$	100
t_4	$write(qtaP_x, A)$	$A = A - 30$	50
t_5	$write(qtaC_{xz_1}, 50)$	$write(qtaP_x, A)$	70
t_6	commit	$write(qtaC_{xz_2}, 30)$	70
t_7		commit	70

Una variante di 2PL in grado di eliminare anche l'anomalia delle letture sporche è il **2PL Ristretto** (o Strict 2PL), il quale aggiunge rispetto alla variante classica la condizione per cui i lock possono essere rilasciati solo dopo il **commit** o l'**abort**.

Uno dei problemi del 2PL è la possibilità di situazioni di deadlock: due o più transazioni sono bloccate nella loro esecuzione perché in attesa del rilascio di un lock, l'uno posseduto dall'altra. Per tale motivo, i DBMS utilizzano tecniche di prevenzione del deadlock per riconoscere in anticipo le situazioni in cui è più probabile che si verifichi deadlock, andando a negare la concessione di lock sugli oggetti.

- Metodi basati su timestamp

I metodi basati su timestamp sono in grado di generare schedule serializzabili sfruttando da un lato le informazioni sugli istanti temporali in cui le transazioni nascono e dall'altro tenendo traccia dei tempi in cui le transazioni accedono agli oggetti per effettuare le operazioni di lettura/scrittura. Ad ogni transazione è associato un **timestamp**, ovvero un identificatore che rappresenta l'istante di inizio della transazione stessa; sotto l'ipotesi della commit-proiezione, uno schedule è accettato solo se riflette l'ordinamento seriale delle transazioni indotto dai timestamp. In particolare, una transazione non può leggere (o, rispettivamente, scrivere) un oggetto scritto (o, rispettivamente, letto) da una transazione più “giovane” (con un timestamp superiore).

Un caso particolare di timestamp è **TS Monoversione**. Ad ogni oggetto X della base sono associati due variabili: **RTM(X)**, che rappresenta il timestamp dell'ultima transazione che ha effettuato una lettura sull'oggetto X, e **WTM(X)**, che rappresenta il timestamp dell'ultima transazione che ha effettuato una scrittura sull'oggetto X. Lo schedule riceve richieste di lettura, $read(\tau, X)$, e scrittura, $write(\tau, X)$, con indicato il timestamp τ della transazione. Nel caso di richiesta di lettura, questa viene respinta se $\tau < WTM(X)$ e la transazione viene uccisa, altrimenti viene

accolta e $RTM(X) = \max(\tau, RTM(X))$. Nel caso di richiesta di **scrittura**, questa viene **respinta se $\tau < WTM(X)$ o se $\tau < RTM(X)$** e la transazione viene **uccisa**, altrimenti viene **accolta** e $WTM(X) = \tau$.

E sono entrambi metodi “pessimistici” (o “conservativi”) perché tendono a ritardare l’esecuzione di transazioni che potrebbero generare conflitti, e quindi anomalie, rispetto allo schedule corrente; di contro, esistono anche dei metodi “ottimistici” ed applicabili in caso di ambienti concorrenti in cui conflitti tra le diverse operazioni delle transazioni sono rari e permettono l’esecuzione sovrapposta e non sincronizzata di transazioni, effettuando un **controllo sui possibili conflitti generati solo a valle del commit**.

In alcuni sistemi di basi di dati i **conflitti tra transazioni concorrenti che tentano di accedere agli stessi oggetti sono rari**; pertanto, l’utilizzo di tecniche basate su lock o su timestamp risultano essere **overkill e non necessarie**. In questi ambienti, si preferiscono utilizzare le cosiddette tecniche ottimistiche in cui **ogni transazione effettua liberamente le proprie operazioni sugli oggetti della base di dati secondo l’ordine temporale con cui le operazioni stesse sono generate**; solo all’atto del **commit** viene effettuato un controllo per stabilire se sono stati riscontrati eventuali conflitti e, nel caso, viene effettuato il **rollback delle azioni delle transazioni e la relativa riesecuzione** (restarting). La riesecuzione di transazioni comporta, inevitabilmente, **un allungamento significativo dei relativi tempi di risposta** e, quindi, risulta essere **tollerabile solo se accade raramente** (motivo per cui in sistemi in cui i conflitti non sono rari le tecniche ottimistiche non vengono minimamente prese in considerazione).

In generale, **un protocollo di controllo di concorrenza ottimistico è basato su tre fasi**:

- **Fase di lettura**, ogni transazione legge i valori degli oggetti della base di dati su cui deve operare e li memorizza in variabili (copie) locali dove sono effettuati eventuali aggiornamenti;
- **Fase di validazione**, vengono effettuati dei controlli sulla serializzabilità degli schedule nel caso che gli aggiornamenti locali delle transazioni dovessero essere propagati sulla base di dati;
- **Fase di scrittura**, gli aggiornamenti delle transazioni che hanno superato la fase di validazione sono propagati definitivamente sugli oggetti della base di dati.

Si vuole far notare come **la fase di rollback delle transazioni**, che non superano la fase di validazione e devono essere rieseguite, è **meno onerosa in quanto riguarda variabili locali e non oggetti della base di dati**.

Il DBMS permette al DBA di stabilire la politica per il controllo di concorrenza che si desidera adottare per il sistema di basi di dati. In SQL è possibile differenziare transazioni che effettuano solo letture da quelle che effettuano letture/scritture, è possibile stabilire il cosiddetto livello di isolamento richiesto, tra i seguenti:

- **READ UNCOMMITTED**, la transazione accetta di leggere dati modificati da una transazione che non ha ancora fatto il commit (ignora i lock esclusivi e non acquisisce lock in lettura);
- **READ COMMITTED**, la transazione accetta di leggere dati modificati da una transazione solo se questa ha fatto il commit ma, se un dato è letto due volte, si potranno trovare dati diversi;
- **REPEATABLE READ**, la transazione accetta di leggere dati modificati da una transazione solo se questa ha fatto il commit e, se un dato è letto due volte, si avrà sempre lo stesso risultato;
- **SERIALIZABLE**, produce schedule serializzabili senza anomalie.

Il **Controllo di Affidabilità** (CdA) ha come obiettivo quello di “ripristinare” il corretto stato (recovery) di un sistema di basi di dati a valle di un possibile malfunzionamento delle sue componenti hardware o software, dovuto a guasti accidentali o intenzionali. Esso deve garantire le proprietà di atomicità e persistenza delle transazioni che rappresentano l’unità base delle attività di recovery.

La memorizzazione dei dati in un DBMS coinvolge, come visto, tre differenti tipologie di **memorie** (storage): memoria centrale, memoria di massa e memoria stabile (come nastri, repliche RAID, ecc...). La **memoria centrale** rappresenta lo **storage primario** di un sistema di basi di dati con caratteristiche di **elevata velocità** per le operazioni di lettura/scrittura, **capacità ridotta** e **volatilità delle informazioni**; complementarmente, la **memoria di massa** e la **memoria stabile** rappresentano lo **storage secondario**, con caratteristiche di **persistenza ed elevata capacità**, al costo di **velocità ridotte**. Infine, la **memoria stabile** identifica in maniera astratta una particolare memoria che non può danneggiarsi. In quest’ottica, il Gestore dell’Affidabilità deve garantire che gli effetti di tutte le transazioni che hanno effettuato il commit siano memorizzate in memoria permanente, in modo che a valle di guasti sia sempre possibile ripristinare il contenuto corretto di una base di dati. Il problema principale è dovuto al fatto che le **operazioni di scrittura delle transazioni non sono azioni atomiche**: gli effetti di una transazione che hanno effettuato il commit in memoria primaria potrebbero essere non riportati subito su memoria secondaria e quindi essere resi persistenti.

Il Gestore dell’Affidabilità deve gestire l’esecuzione dei comandi transazionali di bot, commit, rollback e tutte le operazioni di ripristino dopo guasti. Un **log** è un file presente su memoria stabile che regista tutte le operazioni svolte dalle transazioni nel loro ordine di esecuzione. Il log è, quindi, una sorta di “diario di bordo” che, in un qualsiasi istante, permette di ricostruire il contenuto corretto della base di dati a seguito di malfunzionamento. In questo ambiente si distinguono due operazioni fondamentali:

- **CHECKPOINT**, serve ad effettuare il punto della situazione, registrando le transazioni attive in un certo istante e verificando che le altre o non siano iniziate o siano finite (viene fatta una sorta di sincronizzazione tra il contenuto della base di dati ed il file di log, semplificando le successive operazioni di ripristino);
- **DUMP**, corrisponde alla classica operazione di backup o copia (solitamente pianificata nel tempo e prodotta mentre il sistema non è operativo) del contenuto della base di dati su memoria stabile necessaria alla ricostruzione del suo contenuto, soprattutto nel caso di danneggiamento della memoria secondaria non stabile.

In realtà, un’operazione di checkpoint è abbastanza complessa; nella sua forma più semplice, sono previste le seguenti azioni:

- Si sospende l’accettazione di richieste di ogni tipo da parte delle transazioni;
- Si trasferiscono in memoria di massa (tramite le primitive force del buffer manager) tutte le pagine sporche relative a transazioni andate in commit;
- Si registra sul log in modo sincrono (con un force) un record di checkpoint contenente gli identificatori delle transazioni attive;
- Si riprende l’accettazione delle operazioni.

A valle di un checkpoint si è certi che, per tutte le transazioni che hanno effettuato il commit, i dati sono in memoria di massa e che siano state individuate correttamente le sole transazioni “in corso”.

Nel dettaglio, il log di sistema di basi di dati contiene le seguenti informazioni:

- *Record di transazione*, contenenti a loro volta:
 - ▶ l'identificativo della transazione (T);
 - ▶ il timestamp delle varie azioni (t_s);
 - ▶ la particolare azione (O_p) svolta da una transazione: begin (B), update (U), delete (D), insert (I), commit (C), abort (A);
 - ▶ l'identificativo dell'oggetto della base di dati coinvolto (O);
 - ▶ il valore dell'oggetto prima della modifica (B_i) da parte di una transazione (in gergo *before-image*);
 - ▶ il valore dell'oggetto dopo della modifica (A_i) da parte di una transazione (in gergo *after-image*);
- *Record di sistema*
 - ▶ *Dump (DP)* contenente l'istante di tempo (t_s) in cui è stato effettuato l'ultimo backup della base di dati ed alcuni dettagli pratici (es. file e dispositivi coinvolti);
 - ▶ *Checkpoint (CK)* contenente l'insieme delle transazioni attive (\mathcal{T}) in un dato istante (t_s) sulla base di dati.

L'esito di una transazione è determinato quando viene scritto il record di commit (in modo sincrono) nel log. Un guasto prima di tale istante potrebbe comportare un **disfacimento (undo)** di tutte le azioni di una transazione registrate all'interno del log, qualora i dati siano già stati memorizzati sulla base di dati; di contro, un guasto successivo al commit potrebbe comportare il **rifacimento (redo)** di tutte le azioni svolte da una transazione all'interno del log, qualora queste non siano state ancora registrate nella memoria secondaria della base di dati. Pertanto, per garantire un corretto ripristino di un sistema di basi di dati, esistono delle precise regole per la scrittura su log:

- **Write ahead**, ogni transazione scrive sul log i relativi record prima di effettuare la medesima azione sulla base di dati, consentendo di disfare le azioni perché per ogni aggiornamento viene reso disponibile nel log il valore prima della scrittura su database;
- **Commit precedenza**, ogni transazione scrive sul log tutti i relativi record prima di effettuare il commit, consentendo di rifare le azioni perché se le pagine modificate non sono state ancora trascritte dal buffer manager viene reso disponibile nel log il valore in esse registrato.

Per quanto riguarda la scrittura sulla base di dati, esistono tre differenti modalità:

- **Modalità immediata**, ogni scrittura su log è immediatamente seguita da una scrittura sulla base di dati, evitando operazioni di redo ma aumentando la probabilità di incorrere in operazioni di undo;
- **Modalità differita**, ogni scrittura sulla base di dati è differita alla successiva memorizzazione del record di commit della relativa transazione nel file di log, evitando operazioni di undo ma aumentando la probabilità di incorrere in operazioni di redo;
- **Modalità mista**, la scrittura può avvenire in modalità sia immediata che differita e richiede sia undo che redo.

Le azioni di recovery, quindi, mireranno a ricostruire, sulla base dell'analisi del file di log e partendo dall'ultimo dump utile della base di dati, il relativo contenuto intervenendo con azioni di undo o redo.

Per quanto riguarda i guasti, si distinguono:

- **Guasti soft** (errori di programma, crash di sistema, caduta di tensione, ...), si perde il contenuto della sola memoria centrale mentre rimangono intatte la memoria secondaria e quella stabile;
 - In questi casi, viene effettuata la cosiddetta **ripresa a caldo (warm restart)**;
- **Guasti hard**, avvengono su dispositivi di memoria di massa e si perde sia il loro contenuto che quello della memoria centrale, mentre rimangono intatte le memorie stabili;
 - In questi casi, viene effettuata la cosiddetta **ripresa a freddo (cold restart)**.

In entrambi i casi, **la procedura di ripristino avviene nelle seguenti fasi**, descritte nel **modello fail-stop**:

- Si forza l'arresto completo delle transazioni attive sul sistema di basi di dati;
- Viene ripristinato il corretto funzionamento del sistema operativo;
- Viene effettuata la procedura di ripristino.

Volendo entrare nel dettaglio dei due tipi di ripresa, si consideri in prima istanza la **ripresa a caldo**. Viene ripercorso il log a ritroso finché non viene trovato l'ultimo record di checkpoint e, sulla base delle transazioni attive, vengono costruiti l'insieme undo delle transazioni da disfare, ovvero quelle che hanno effettuato l'abort prima del guasto o che non hanno effettuato in tempo il commit ma hanno scritto sulla base di dati, e l'insieme redo delle transazioni da rifare, ovvero quelle che hanno effettuato il commit prima del guasto. A questo punto il log viene ripercorso all'indietro, fino alla più vecchia azione delle transazioni negli insiemi undo e redo, disfaccendo tutte le azioni delle transazioni presenti nel primo dei due insiemi: nel caso di update, si ripristina il valore B_I , nel caso di inserti si effettua la cancellazione dell'oggetto inserito, mentre nel caso di delete si rieffettua l'inserimento; successivamente, viene ripercorso il log in avanti, rifacendo tutte le azioni delle transazioni appartenenti all'insieme redo: nel caso di update si ripristina il valore A_I , nel caso di inserti si rieffettua l'inserimento dell'oggetto e nel caso di delete la cancellazione.

La **ripresa a freddo**, invece, avviene a seguito di guasti che provocano danni alla base di dati (siano essi hard o soft); la tecnica più diffusa prevede l'esecuzione dei seguenti passi:

- Si ripristinano i dati a partire dal backup, accedendo al più recente record di dump del log;
- Si eseguono tutte le operazioni registrate sul log relativamente alla parte deteriorata, riportandosi all'istante precedente il guasto;
- Si esegue una ripresa a caldo.

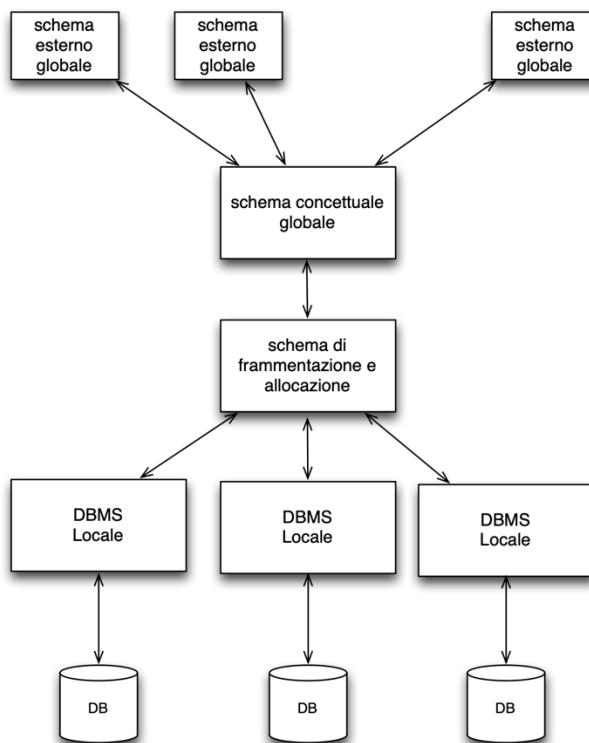
BASI DI DATI DISTRIBUITE

Per **Base di Dati Distribuita** si intende una **collezione di dati condivisi e logicamente correlati che sono fisicamente distribuiti su una rete di calcolatori**, mentre il **DDBMS è il sistema software che gestisce una base di dati distribuita**, rendendo la **distribuzione completamente trasparente** all'utente finale. La necessità di passare da un **sistema centralizzato**, con un unico database locale controllato da un DBMS, a **sistemi distribuiti**, che permettono l'accesso a dati memorizzati in più siti remoti, nasce da due necessità: la prima riguarda la **collezione di una quantità di dati sempre crescente, rendendo l'infrastruttura fisica locale non più conveniente da un punto di vista gestionale**, mentre la seconda riguarda l'**utenza di un database sempre più sparsa nel mondo, rendendo necessario avvicinare fisicamente i dati agli utenti** per garantire una velocità di accesso maggiore.

Una **base di dati distribuita** consiste in **un unico database, dal punto di vista logico, che è costituito da un insieme di frammenti, ognuno dei quali è a sua volta memorizzato in uno o più server in rete sotto il controllo di un proprio DBMS**. In un sistema di questo tipo, **ogni server è in grado di elaborare in modo indipendente le richieste degli utenti che richiedono accesso a dati locali ed i dati che sono memorizzati in altri siti della rete**. Volendo valutare la scelta di un DB distribuito, si deve considerare che:

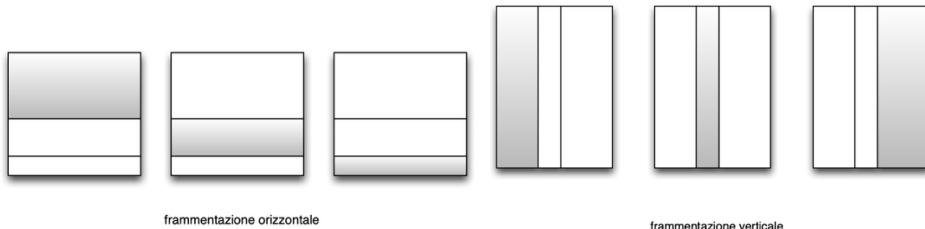
- **PRO**, la distribuzione dei dati su più server con hardware meno costoso e performante può favorire l'economicità di una soluzione, la scalabilità del sistema, la semplicità della modalità di integrazione di sistemi differenti, l'affidabilità, la disponibilità dei dati e le prestazioni complessive;
- **CONTRO**, una soluzione distribuita aumenta la complessità del sistema hardware e software, anche a fronte di mancanza di standard di mercato.

Se su tutti i server è usato lo stesso DBMS, si parla di **database omogenei**, altrimenti si parla di **database eterogenei**. Se **ogni server mantiene autonomia completa**, poi, si parla anche di **sistemi multidatabase**. L'architettura di riferimento per questo tipo di sistemi è la seguente:



Si distinguono i seguenti blocchi: **un insieme di schemi esterni globali**, **uno schema globale dedicato alla descrizione dell'intera base di dati**, come se essa fosse non distribuita (astrazione della distribuzione), **uno schema di frammentazione e di allocazione**, ovvero la descrizione di come i dati sono logicamente partizionati e di dove i dati sono allocati, e **un insieme di DBMS locali** basati sull'architettura ANSI SPARC. La frammentazione può essere di due tipologie: **orizzontale**, consiste in **un insieme di tuple** (pertanto, è detta anche frammentazione della tabella), o **verticale**, consiste in **un insieme di attributi** (pertanto, è detta anche frammentazione dello schema). Un'operazione di **frammentazione di una relazione r in frammenti r_1, \dots, r_m** è corretta se e solo se valgono le seguenti proprietà:

- **Completezza**, ogni tupla che compare in r deve poter essere trovata in almeno un frammento r_i ;
- **Ricostruibilità**, è possibile definire un'operazione (combinando gli operatori dell'algebra relazionale) che, a partire dai frammenti r_i , permetta di ricostruire la relazione r in modo da mantenere le dipendenze funzionali di r stessa;
- **Disgiunzione**, se un dato appare in un frammento, non deve essere presente in nessun altro frammento delle frammentazioni, per evitare ridondanza.



Data una relazione r , un **frammento orizzontale** si può ottenere attraverso un'operazione di selezione avente una condizione di selezione χ che coinvolge uno o più attributi dello schema di relazione R di r : $\sigma_\chi(r)$. L'operazione di ricostruzione, in questo caso, si può effettuare attraverso un'operazione di unione insiemistica:

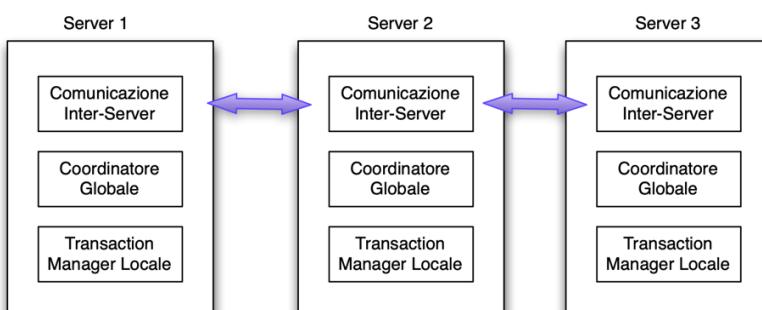
$$r = r_1 \cup \dots \cup r_m$$

Un'operazione di frammentazione verticale, invece, si può ottenere con un'operazione di proiezione su un sottoinsieme di attributi A dello schema R di r : $\pi_A(r)$. L'operazione di ricostruzione, in questo caso, si può effettuare attraverso un'operazione di join:

$$r = r_1 \bowtie \dots \bowtie r_m$$

Ogni frammento è realizzato tramite una o più tabelle in un database allocato su un certo server. L'allocazione è non ridondante se ogni frammento viene allocato su un solo server e ridondante se alcuni frammenti sono allocati su più server. La trasparenza nasconde i dettagli implementativi, è possibile avere sia trasparenza di implementazione (quando l'utente non sa come i dati siano frammentati) che trasparenza di allocazione (quando l'utente non sa dove risiedono i dati).

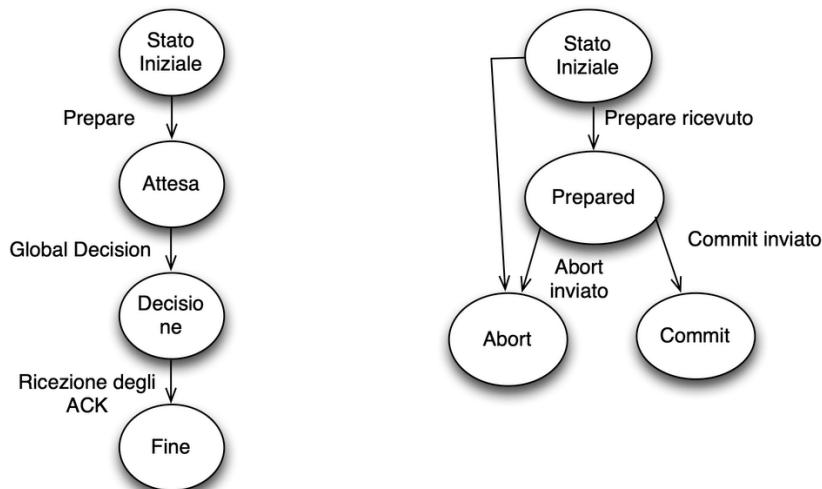
Chiaramente, anche per i DDBMS le transazioni devono verificare le proprietà ACID; tuttavia, a differenza del caso concentrato, nei sistemi distribuiti è necessario un Coordinatore Globale che, su ogni server, ha il compito di coordinare l'esecuzione sia delle transazioni globali che di quelle locali sul sito (gestite attraverso un Transaction Manager Locale). Inoltre, le comunicazioni tra i diversi server devono avvenire attraverso una componente di comunicazione (Comunicazione Inter-Server).



Andando più nel dettaglio circa le proprietà ACID, **Consistenza e Atomicità sono vincoli locali e i meccanismi usati per il caso centralizzato restano validi anche nel caso globale**. Per quanto riguarda l'isolamento, se un server locale usa il 2PL ristretto o il metodo dei timestamp (e quest'ultimi sono assegnati in maniera globale alle sottotransazioni), lo scheduling globale è serializzabile; pertanto, se lo schedule dell'esecuzione delle transazioni ad ogni server è serializzabile, allora lo è anche quello globale, inteso come l'unione di tutti gli schedule globali. Infine, per la persistenza è richiesta un po' di attenzione, soprattutto per la gestione di eventuali fallimenti; infatti, occorre tenere presente il verificarsi di perdita di messaggi, caduta di connessione, caduta di un server e l'eventuale partizionamento della rete di comunicazione.

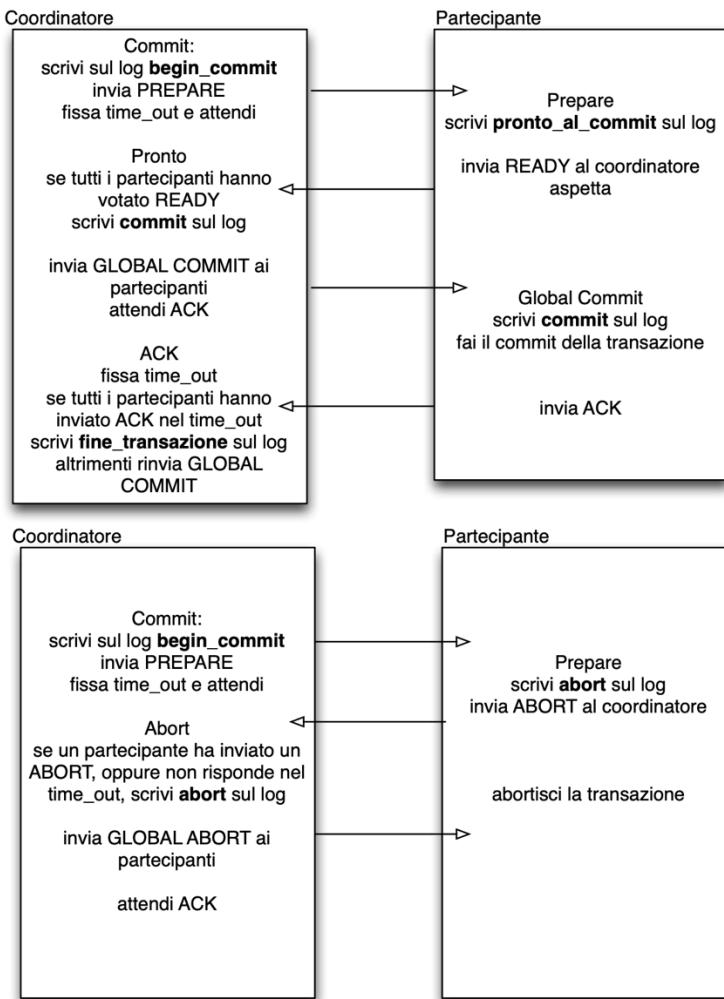
Per assicurare l'atomicità di una transazione globale, occorre che sia assicurato che ogni singola sottotransazione vada in abort o in commit, attraverso l'uso di opportuni protocolli di commit (Two Phase Commit, 2PC, ad esempio) che mettano il sistema a riparo da eventuali fallimenti. Di seguito, si supporrà che ogni transazione globale sia gestita da un server che agisce da coordinatore della transazione, che è generalmente il server che ha iniziato la transazione, e dagli altri server locali coinvolti, detti partecipanti. Il coordinatore conosce l'identità di tutti i partecipanti e viceversa. Il protocollo prevede una fase preliminare di "matrimonio" attraverso una procedura detta "Rito del Matrimonio":

- Il coordinatore chiede a tutti i partecipanti se sono preparati a fare il commit delle transazioni (fase di voting);
- Se esiste un partecipante che genera abort, oppure se fallisce nel rispondere entro un certo prestabilito intervallo di tempo, allora il coordinatore informa tutti i partecipanti che la transazione sarà abortita (fase di decisione) e tale decisione dovrà essere adottata da tutti i partecipanti;
- Se un partecipante ha votato per l'abort, può abortire la transazione in maniera unilaterale, se ha votato per il commit, deve attendere la decisione globale (global commit o global abort) e ad essa attenersi.

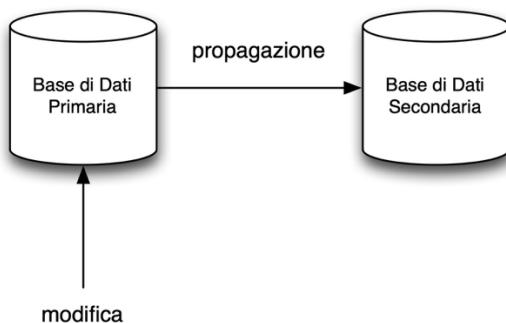


Ovviamente, il global commit avverrà da parte del coordinatore solo se tutti i partecipanti hanno votato in tempo per il commit; il coordinatore e ogni partecipante dovranno, poi, avere i propri log locali per poter effettuare un eventuale rollback o commit. I problemi di questo approccio sono la possibilità di deadlock e la caduta del coordinatore.

Un esempio di rito del matrimonio su transazioni globali è il seguente:



Un **modello di replicazione di basi di dati** (modello asimmetrico) consiste nel **copiare e mantenere le informazioni gestite da una base di dati in più server distribuiti**. La replicazione richiede che **una qualsiasi modifica fatta su un oggetto della base di dati su di un server sia applicata anche a tutte le altre copie fisicamente memorizzate su altri siti**:



Un **server master** controlla le **replicazioni** in modo che tutte le modifiche su di esso fatte si propaghino a tutti gli altri siti, detti **slaves**, i quali mantengono le copie del master che gli competono e vengono periodicamente modificati per mantenersi in linea con le informazioni presenti sul master. La **sincronizzazione** tra master e slaves può avvenire seguendo **due modalità**:

- **Sincrona**, prevede una modifica immediata degli oggetti modificati nel master anche in tutti gli slave correlati, facendo uso di opportuni protocolli (come 2PC);

- **Asincrona**, la modifica degli slave a seguito di modifica del master avviene solo in certi intervalli di tempo (entro qualche minuti, varie ore o anche giorni).

In questo modo viene garantita l'affidabilità del sistema: avere a disposizione copie multiple dello stesso dato permette, in caso di guasti di uno o di più siti, di effettuare il recovery a caldo in modo efficace ed efficiente; inoltre, invece che incidere su un unico server centralizzato, si può prevedere l'uso di più server remoti, migliorando complessivamente le prestazioni e il bilanciamento del carico di sistema e permettendo, allo stesso tempo, un numero sempre maggiore di utenti.