Sistemi informativi, informazioni e dati

Nello svolgimento di qualsiasi attività è essenziale la disponibilità di informazioni e la capacità di gestirle in modo efficace; ogni organizzazione è dotata di un sistema informativo che gestisce le informazioni necessarie al perseguimento degli scopi della organizzazione stessa. Il concetto di sistema informativo è in parte indipendente dalla sua automatizzazione, perché può includere anche procedure non manuali. Per indicare la porzione automatizzata del sistema informativo viene di solito utilizzato il termine Sistema informatico.

Un sistema informativo (o informatico) è costituito da due parti principali:

- Insieme delle procedure (che lavorano su...)
- Dati

Le procedure sono fatte per ragionare sui dati. I dati sono informazioni grezze di cui non si conosce la rappresentazione e quindi a meno di una giusta interpretazione non sono di nessun significato. Una base di dati è una "collezione di dati".

Per gestire dei dati con un calcolatore esistono due diversi approcci; L'approccio convenzionale sfrutta la presenza di archivi o file immagazzinati sul filesystem per memorizzare i dati in modo persistente sulla memoria di massa, questo metodo alquanto semplice e intuitivo ha tutta una serie di svantaggi:
Le procedure scritte in un linguaggio di programmazione sono completamente autonome, definiscono e utilizzano uno o più file privati, quindi dati di interesse per più programmi sono replicati tante volte quanti sono i programmi che li utilizzano (ridondanza e possibilità di incoerenza).
Inoltre quando la quantità di dati è enorme, i dati diventano ingestibili, ed i file forniscono solo semplici meccanismi di accessi e condivisione con un problema sicurezza dei dati e di controllo e gestione degli accessi.

L'altro approccio per gestire dati all'interno di applicazioni è quello di interporre uno strato software a cui demandare l'interazione con il filesystem vero e proprio. Uno strato software che interagisce anche con le applicazioni mettendo a disposizione i dati.

Questi software vengono indicati con la sigla DBMS, che sta per Data Base Management System (sistema di gestione delle basi di dati), i più noti sono Microsoft Access, Microsoft SQL Server, PostgreSQL, MySQL, e il leader del mercato Oracle.
D'ora in poi considereremo una base di dati come una collezione di dati gestita tramite un DBMS.

Un sistema DBMS consente di gestire collezioni di dati che siano *grandi*, *condivise* e *persistenti*, assicurando loro *affidabilità* e *privatezza*.

- Le basi di dati possono essere di dimensioni enormi, di molti Gb, e in generale di dimensioni molto maggiori della memoria centrale disponibile.
- Le basi di dati sono *condivise*, applicazioni e utenti diversi debbono poter accedere, secondo opportune modalità, a dati comuni; in questo modo si riduce la ridondanza dei dati e la possibilità di inconsistenze.
- Le basi di dati sono *persistenti*, hanno cioè un tempo di vita che non è limitato a quello delle singole esecuzioni dei programmi che le utilizzano.
- I DBMS garantiscono la *privatezza* dei dati. Ciascun utente, viene abilitato a svolgere solo

determinate azioni sui dati, attraverso meccanismi di autorizzazione.
- I DBMS sono *efficienti*, capaci di svolgere le operazioni utilizzando un insieme di risorse che sia accettabile per gli utenti, ed *efficaci*, capaci cioè di rendere produttive, in ogni senso, le attività dei suoi utenti.

I DBMS mettono a disposizione dei livelli di indipendenza (dal livello logico e dal livello fisico):
L'*indipendenza fisica* consente di interagire con il DBMS in modo indipendente dalla struttura fisica dei dati.
L'*indipendenza logica* consente di interagire con il livello esterno della base di dati in modo indipendente dal livello logico.

Riassumendo i DBMS comportano una serie di vantaggi e alcuni svantaggi:
- DBMS permettono di considerare i dati come una risorsa comune di una organizzazione, a disposizione di tutte le sue componenti.
- La base di dati fornisce un modello unificato e preciso della parte del mondo reale di interesse per l'organizzazione.
- Con i DBMS è possibile un controllo centralizzato dei dati.
- La condivisione permette di ridurre ridondanze e inconsistenze.
- L'indipendenza dei dati favorisce lo sviluppo di applicazioni più flessibili e facilmente modificabili.
- Hanno come svantaggio tutta una serie di costi: il costo del software DBMS, il costo hardware, il costo di transizione da un approccio filesystem e soprattutto, più importante, il costo di know-how per l'amministrazione della banca di dati da parte di un DBA (Data Base Administrator).

I primi modelli di DBMS ad essere proposti furono:
- Il modello Gerarchico, basato sull'uso di strutture ad albero, tuttora ampiamente utilizzato.
- Il modello Reticolare, basato sull'uso di grafi.
- Il modello Relazionale, in assoluto il più diffuso, per il modo in cui i dati sono logicamente organizzati. Il modello relazionale ha come limite la gestione di dati di tipo multimediale (come suoni, immagini, video), e fu proposto un modello OO (Object Oriented), che estende alle basi di dati il paradigma di programmazione ad oggetti. Questo modello però non ha avuto successo in quanto il suo approccio scartava totalmente il modello relazionale.

Il Modello Relazionale

Il modello relazionale si basa su due concetti, relazione e tabella, di natura diversa ma facilmente riconducibili l'uno all'altro. La nozione di relazione proviene dalla teoria degli insiemi, mentr il concetto di tabella è semplice e intuitivo.
S1 { 1, 2, 3 }
S2 { A, B, C }
S1 x S3 { (1, A), (1, B), (1, C), (2, A), ..., (3, C) }
Una relazione matematica D è un qualsiasi sottoinsieme del prodotto cartesiano tra i due insiemi S1 ed S2 (es. D { (1, A), (2, B), (3, C) }) dove S1 ed S2 sono i domini della applicazione. Una tabella è una rappresentazione non posizionale nell'ordine delle righe e delle colonne (ciò si ottiene assegnando un nome ai domini, alle colonne) di una relazione. Il nome della colonna è detto attributo, che descrive il "ruolo" giocato dal dominio. Dovendo identificare univocamente le componenti, gli attributi di una relazione debbono essere diversi l'uno dall'altro.
Una relazione con attributi altro non è che una tabella alla quale abbiamo assegnato un nome alla colonna.

Un elemento della relazione con attributi si chiama tupla.

Scasa	***Sospite***	***G1***	***G2***
Juve	Napoli	0	4
Milan	Roma	0	4

Tupla T1.Scasa = Milan

Attraverso il concetto di tabella è anche possibile correlare le informazioni, che vengono collegate tra di loro attraverso valori comuni degli attributi.

Per rappresentare in modo semplice e comodo la non disponibilità di valori, il concetto di relazione viene esteso prevedendo che una tupla possa assumere, su ciascun attributo, o un valore del dominio, oppure un valore speciale, detto *valore nullo*, che denota la assenza di informazione, ma è un valore aggiuntivo rispetto a quelli del dominio, e ben distinto da essi.
Una banca dati relazionale è un insieme di relazioni e di vincoli che si applicano su di esse.
Un vincolo viene rappresentato da una condizione logica che deve essere soddisfatta affinché il vincolo sia rispettato.

È possibile classificare i vincoli a seconda degli elementi che ne sono coinvolti.

- *Intrarelazionali*: se il suo soddisfacimento è definito rispetto alle singole relazioni della base di dati, in particolare se il suo soddisfacimento può essere effettuato guardando al solo inserimento viene definito vincolo di tupla, perché valutato su ciascuna tupla indipendentemente dalle altre.
- *Interrelazionali*: se il suo soddisfacimento può essere effettuato solo guardando a tutte le relazioni e i riferimenti, coinvolge più relazioni. È molto importante il vincolo di integrità referenziale che correla informazioni in diverse tabelle:
  Un vincolo di integrità referenziale fra un insieme di attributi X di una relazione R1 ed una relazione R2 è soddisfatto se i valori su X di ciascuna tupla dell'istanza R1 compaiono come valori della chiave (primaria) dell'istanza di R2.

Quanto più forte saranno i vincoli più bassa sarà la probabilità che i dati possano essere errati.

Una chiave è un insieme di attributi utilizzato per identificare univocamente le tuple di una relazione (ogni relazione ha una chiave). Su una delle chiavi (detta *chiave primaria*) si vieta la presenza di valori nulli affinché sia garantita la univoca identificazione della tupla.
Se una chiave primaria è composta da più attributi l'assenza del NULL deve essere estesa a tutti gli attributi che compongono la chiave. Una super-chiave è l'insieme di una chiave e di altri attributi non strettamente identificativi (quindi una chiave ridondante).

SQL, Structured Query Language (Linguaggio Strutturato di Interrogazione)
Per quantificare in modo preciso l'aderenza allo standard sono stati definiti tre livelli di supporto dei costrutti del linguaggio, denominati rispettivamente Entry, Intermediate e Full. I sistemi possono essere così caratterizzati in base al livello cui aderiscono. Il livello Intermediate contiene le caratteristiche ritenute più importanti per rispondere alle esigenze del mercato, e viene attualmente offerto da diverse versioni recenti dei prodotti relazionali di maggior diffusione.

DDL: Data Definition Language

`Character`	Tipo Carattere
`Character(N)`	Tipo Stringa composta da N caratteri
`Varchar(N)`	Tipo Stringa variabile composta al massimo da N caratteri
`Bit`	Tipo Binario [0, 1]
`Bit(N)`	Tipo sequenza binaria composta da N bit
`Varbit(N)`	Tipo sequenza binaria variabile composta al più da N bit
`Date`	Tipo data (Anno-Mese-Giorno)
`Time`	Tipo ora (Ore Minuti Secondi)
`Timestamp`	Tipo timbro temporale (anno mese giorno ore minuti secondi)
`Interval`	Tipo intervallo temporale tra date
`Integer`	Tipo numerico intero (di solito 4 byte)
`Smallint`	Tipo numerico intero ridotto (byte <= byte integer)
`Numeric(I, J)`	Tipo numerico in virgola fissa (I numeri dei quali J decimali)
`Decimal(I, J)`	Tipo numerico decimale
`Float`	Tipo numerico adatto a rappresentare valori reali precisione a seconda della imp.
`Real`	Tipo numerico in virgola mobile con notazione scientifica con precisione fissa
`Double`	Tipo numerico in virgola mobile doppio (byte >= byte real)

Una *tabella* SQL è costituita da una collezione ordinata di attributi e da un insieme (eventualmente di vincoli). La sintassi per la definizione delle tabelle è:

```
create table NomeTabella
( NomeAttributo1 Dominio [Valore di Default] [ Vincoli ]
, NomeAtturbutoN Dominio [Valore di Default] [ Vincoli ]
)
```

Nella definizione delle tabelle si può far riferimento ai domini predefiniti del linguaggio o a *domini* definiti dall'utente a partire dai domini predefiniti

```
create domain NomeDominio as TipoDiDato [ Valore di Default ] [ Vincolo ]
```

La dichiarazione di nuovi domini permette di associare un insieme di vincoli ad un nome di dominio.

Nella sintassi per la definizione dei domini e delle tabelle è possibile specificare il valore di *default*, ovvero, il valore che deve assumere l'attributo quando viene inserita una riga nella tabella senza che sia specificato un valore per l'attributo stesso. Quando il valore di default non è specificato, si assume come default il valore nullo. La sintassi specifica dei valori di default è:

```
default < Generico Valore | user | null >
```

Sia nella definizione di domini che nella definizione delle tabelle è possibile definire dei vincoli, ovvero delle proprietà che devono essere verificate da ogni istanza della base di dati:

Il vincolo *not null* indica che il valore nullo non è ammesso come valore dell'attributo; in tal caso, l'attributo

deve sempre essere specificato, tipicamente in fase di inserimento. Il vincolo viene specificato facendo seguire alla definizione dell'attributo le parole chiave `not null`.
Qualche sistema propone di invertire le considerazioni sui valori nulli rendendo NOT NULL di default tutti gli attributi a meno che non sia specificato NULLABLE per specificare che la tupla può assumere in quel dominio dei valori nulli.
Un vincolo *unique* si applica ad un attributo o un insieme di attributi di una tabella e impone che i valori dell'attributo siano una (super)chiave, cioè che righe differenti della tabella non possono avere gli stessi valori; viene fatta un'eccezione per il valore nullo, il quale può comparire su diverse righe senza violare il vincolo. La definizione di questo vincolo può avvenire in due modi; il primo è usato quando bisogna definire il vincolo su un solo attributo; in questo caso si fa seguire la specifica dell'attributo dalla parola chiave `unique` (analogamente a come avviene per *not null*); il secondo è necessario quando il vincolo opera su un insieme di attributi. Dopo aver definito gli attributi della tabella, si usa la sintassi:

`unique` (*Attributo* { , *Attributo* })

Per ogni relazione è di norma necessario specificare la chiave primaria. SQL permette di specificare il vincolo *primary key* una sola volta per ogni tabella. Come il vincolo *unique*, il vincolo *primary key* può essere definito direttamente su di un singolo attributo, oppure essere definito elencando più attributi che costituiscono l'identificatore.
la scrittura *constraint* permette in seguito di modificare o eliminare il vincolo definito semplicemente agendo sul nome del vincolo. La stessa chiave primaria può essere definita come un vincolo constraint:

`constraint` *NomeVincolo* `primary key` (*Attributo* { , *Attributo* }).

I vincoli interrelazionali più diffusi e significativi sono i vincoli di integrità referenziale, per la loro definizione si usa l'apposito vincolo di *foreign key*, ovvero di *chiave esterna*.
Questo vincolo crea un legame tra i valori di un attributo della tabella corrente (interna) e i valori di un attributo di un'altra tabella (esterna). Il vincolo impone che per ogni riga della tabella il valore dell'attributo specificato, se diverso dal valore nullo, sia presente nelle righe della tabella esterna tra i valori del corrispondente attributo. Requisito è che l'attributo a cui si fa riferimento nella tabella esterna sia almeno *unique*, tipicamente l'attributo della tabella esterna cui si fa riferimento rappresenta in effetti la chiave primaria della tabella. Più attributi possono essere coinvolti nel vincolo, quando la chiave della tabella esterna è costituita da un insieme di attributi; in tal caso l'unica differenza è che bisognerà confrontare ennuple di valori invece che singoli valori.

`constraint` *VincoloFK* `foreign key` (*Attr_int* [, *Att_int*]) `references` *Tabella*(*Attr_est* [, *Attr_est*])

La corrispondenza tra gli attributi locali e quelli esterni avviene in base all'ordine: al primo attributo argomento di `foreign key` corrisponde il primo attributo argomento di `references`, e così per gli altri attributi.
Si possono introdurre violazioni modificando il contenuto della tabella interna inserendo una nuova riga, o modificando il valore dell'attributo referente, e queste operazioni vengono semplicemente impedite; vengono invece offerte diverse alternative per rispondere alle violazioni generate da modifiche alla tabella esterna. Le operazioni che possono introdurre violazioni sono le modifiche del valore dell'attributo riferito e la cancellazione delle righe. Per le operazioni di modifica, è possibile agire nei seguenti modi: `cascade` (il nuovo valore viene riportato su tutte le corrispondenti righe della tabella interna), `set null` (all'attributo referente viene assegnato il valore nullo), `set default` (all'attributo referente viene assegnato il valore di default, `no action` (l'azione di modifica non viene consentita).
Per le violazioni prodotte dalla cancellazione di un elemento della tabella esterna si ha a disposizione:

`cascade` (tutte le righe interna corrispondenti alla riga cancellata, vengono cancellate), `set null` (all'attributo referente viene assegnato il valore nullo), `set default` (all'attributo referente viene assegnato il valore di default), `no action` (la cancellazione non viene consentita).
La politica di reazione viene specificata immediatamente dopo il vincolo di integrità secondo la sintassi:

```
on < delete | update > < cascade | set null | set default | no action >
```

Il comando *alter* permette di modificare domini e schemi di tabelle. Il comando può assumere le forme:

```
alter domain NomeDominio < set default ValoreDefault |    drop default |
      add constraint NomeVincolo | drop constraint NomeVincolo>
alter table NomeTabella
      < alter column NomeAttributo < set default ValoreDefault | drop default > |
      add constraint NomeVincolo | drop constraint NomeVincolo |
      add column NomeAttributo | drop column NomeAttributo >
```

Invece il comando *drop* permette di rimuovere dei componenti:

```
drop < schema | domain | table | view | assertion > NomeElemento [ restrict | cascade ]
```

L'opzione `restrict` (di default) specifica che il comando non dev'essere eseguito in presenza di oggetti non vuoti.

Ogni DBMS conserva un insieme di tabelle interne predefinite dette *cataloghi relazionali* dove sono conservati i nomi e i valori di tutte le altre tabelle, attributi, permessi e proprietari delle altre tabelle.

DML: Data Manipulation Language

La parte SQL dedicata alla formulazione di interrogazioni fa parte del DML.
Le operazioni di interrogazione in SQL vengono specificate per mezzo dell'istruzione `select`:

```
select ListaAttributi
from ListaTabelle
[ where Condizione ]
```

Le tre parti di cui si compone un'istruzione `select` vengono spesso ordinatamente chiamate *clausola* `select` (detta anche *target list*), *clausola* `from` e *clausola* `where`. Più precisamente la sintassi è:

```
select AttrExpr [ [as] Alias ] {, AttrExpr [ [as] Alias ] }
from Tabella [ [as] Alias ] {, Tabella [ [as] Alias ] }
[ where Condizione ]
```

L'interrogazione SQL seleziona, tra le righe che appartengono al prodotto cartesiano delle tabelle elencate nella clausola `from`, quelle che soddisfano le condizioni espresse nell'argomento della clausola `where`. Il risultato dell'esecuzione di una interrogazione SQL è così una tabella con una riga per ogni riga selezionata dalla clausola `where` e le cui colonne si ottengono dalla valutazione delle espressioni *AttrExpr* che appaiono nella clausola `select`. Ogni colonna viene eventualmente ridenominata con l'*Alias*, se questo compare dopo l'espressione.

La clausola `select` specifica gli elementi dello schema della tabella risultato. Come argomento della clausola `select` può anche comparire il carattere speciale `*` (asterisco), che rappresenta la selezione di tutti gli attributi delle tabelle elencate nella clausola `from`. Nella clausola `select` possono comparire generiche espressioni sul valore degli attributi di ciascuna riga selezionata.

Quando si desidera formulare un'interrogazione che coinvolge righe appartenenti a più di una tabella, si pone come argomento della clausola `from` l'insieme di tabelle alle quali si vuole accedere.
La clausola `where` ammette come argomento una espressione booleana costruita combinando predicati semplici con gli operatori `and`, `or`, e `not`. Ciascun predicato semplice usa gli operatori =, <>, <, >, <= e >= per confrontare da un lato una espressione costruita a partire dai valori degli attributi per la riga, e dall'altro lato un valore costante o un'altra espressione.
Oltre ai normali predicati di confronto relazionali, SQL mette a disposizione un operatore `like` per il confronto di stringhe, che permette di effettuare confronti con stringhe in cui compaiono i caratteri speciali: '`_`' (underscore) e '`%`' (percentuale). Il primo carattere speciale può rappresentare nel confronto un carattere arbitrario, il secondo una stringa di un numero arbitrario (anche nullo) di caratteri arbitrari.
Per selezionare i termini con valori nulli SQL fornisce il predicato `is null` la cui sintassi è semplicemente:

Attributo `is` [`not`] `null`

Il predicato risulta vero solo se l'attributo ha valore nullo. Il predicato `is not null` è la sua negazione.
Una sintassi alternativa per la specifica dei join permette di distinguere, tra le condizioni che compaiono nell'interrogazione, quelle che rappresentano condizioni di join e quelle che rappresentano condizioni di selezione sulle righe. La sintassi è la seguente:

`select` *AttrExpr* [[`as`] Alias] {, AttrExpr [[`as`] Alias] }
`from` *Tabella* [[`as`] *Alias*] { [*TipoJoin*] `join` *Tabella* [[as] *Alias*] `on` *CondizioneDiJoin* }
[`where` *AltraCondizione*]

Il parametro *TipoJoin* specifica qual è il tipo di `join` da usare, e ad esso si possono sostituire i termini `inner` (interno, valore di default che può essere omesso), `right outer`, `left outer`, o `full outer` (il qualificatore `outer` è opzionale). Con il join interno le righe che vengono coinvolte nel join sono in generale un sottoinsieme delle righe di ciascuna tabella. Può infatti capitare che alcune righe non vengano considerate in quanto non esiste una corrispondente riga nell'altra tabella per cui la condizione sia soddisfatta. Il join esterno (`outer join`) esegue un join mantenendo però tutte le righe che fanno parte di una o entrambe le tabelle coinvolte. Esistono appunto tre varianti dei join esterni: `left`, `right` e `full`. Il `left join` fornisce come risultato il join interno esteso con le righe della tabella che compare a sinistra per le quali non esiste una corrispondente riga nella tabella di destra; il `right join` si comporta in modo simmetrico; infine, il `full join` restituisce il join interno esteso con le righe escluse di entrambe le tabelle.

Nelle interrogazioni SQL è possibile associare un nome alternativo, detto *alias*, alle tabelle che compaione come argomento della clausola `from`. Il nome viene usato per far riferimento alla tabella nel contesto della interrogazioni. Utilizzando gli alias è possibile far riferimento a più esemplari della stessa tabella. Tutte le volte che si introduce un alias per una tabella si dichiara in effetti una variabile che fa riferimento al contenuto della tabella di cui è alias. Quando una tabella compare una sola volta in una interrogazione, non c'è differenza tra l'interpretare l'alias come uno pseudonimo o come una nuova variabile. Quando una tabella compare invece più volte, è necessario considerare l'alias come una nuova variabile.

SQL permette di specificare un ordinamento delle righe del risultato di una interrogazione tramite la clausola order by, con la quale si chiude l'interrogazione. La clausola rispetta la seguente sintassi:

`order by` *AttrDiOrdinamento* [`asc` | `desc`] {, *AttrDiOrdinamento* [`asc` | `desc`] }

In una interrogazione SQL spesso si devono valutare delle proprietà che dipendono da insiemi di tuple.
Gli operatori aggregati vengono gestiti come un'estensione delle normali interrogazioni. Prima viene

normalmente eseguita l'interrogazione, considerando solo le parti `from` e `where`. L'operatore aggregato viene poi applicato alla tabella contenente il risultato dell'interrogazione.
Lo standard SQL, prevede cinque operatori aggregati: `count`, `sum`, `max`, `min` e `avg`.
L'operatore `count` usa la seguente sintassi:

`count (` < `*` | [`distinct` | `all`] *ListaAttributi* > `)`

La prima opzione (`*`) restituisce il numero di righe; l'opzione `distinct` restituisce il numero di diversi valori degli attributi in *ListaAttributi*; l'opzione `all` invece restituisce il numero di righe che possiedono valori diversi dal valore nullo per gli attributi in *ListaAttributi*. Se si specifica un attributo e si omette `distinct` o `all`, si assume `all` come default.
Gli altri quattro operatori aggregati invece ammettono come argomento un attributo o un'espressione, eventualmente preceduta dalle parole chiave `distinct` o `all`. Le funzioni aggregate `sum` e `avg` ammettono come argomento solo espressioni che rappresentano valori numerici o intervalli di tempo.
Le funzioni `max` e `min` richiedono solamente che sull'espressione sia definito un ordinamento, per cui si possono applicare ance su stringhe di caratteri o su istanti di tempo.

< `sum` | `max` | `min` | `avg` > `(` [`distinct` | `all`] *AttrExpr* `)`

Gli operatori si applicano sulle righe che soddisfano la condizione presente nella clausola `where` e hanno il seguente significato:

- `sum`: restituisce la somma dei valori posseduti dall'espressione;
- `max` e `min`: restituiscono rispettivamente il valore massimo e il minimo;
- `avg`: restituisce la media dei valori (ovvero, il risultato della divisione di sum per count).

`distinct` elimina i duplicati, `all` trascura i valori nulli; sugli operatori `max` e `min` non hanno effetti.
La sintassi SQL non ammette che nella stessa clausola `select` compaiano funzioni aggregate ed espressioni al livello di riga, a meno che non si faccia uso della clausola `group by`.

Molto spesso sorge l'esigenza di applicare l'operatore aggregato separatamente a sottoinsiemi di righe. Per poter utilizzare in questo modo l'operatore aggregato, SQL mette a disposizione la clausola `group by`, che permette di specificare come dividere le tabelle in sottoinsieme. La clausola ammette come argomento un insieme di attributi e la query raggrupperà le righe che possiedono gli stessi valori per questo insieme di attributi.
Dopo che le righe sono state raggruppate in sottoinsiemi, l'operatore aggregato viene applicato separatamente su ogni sottoinsieme. Il risultato dell'interrogazione è costituito da una tabella con righe che contengono l'esito della valutazione dell'operatore aggregato affiancato al valore dell'attributo che è stato usato per l'aggregazione.
La sintassi SQL impone che, in una interrogazione che fa uso della clausola `group by`, possa comparire come argomento della `select` solamente un sottoinsieme degli attributi usati nella clausola `group by`. Per questi attributi, infatti, ciascuna tupla del sottoinsieme sarà caratterizzata dallo stesso valore.

Se le condizioni che i sottoinsiemi devono soddisfare sono verificabili al livello delle singole righe, allora basta porre gli opportuni predicati come argomento della clausola `where`. Se invece le condizioni sono delle condizioni di tipo aggregato, sarà necessario utilizzare un nuovo costrutto, la clausola `having`.
La clausola `having` descrive le condizioni che si devono applicare al termine dell'esecuzione di una interrogazione che fa uso della clausola `group by`. Ogni sottoinsieme di righe costruito dalla `group by` fa

parte del risultato della interrogazione solo se il predicato argomento della `having` risulta soddisfatto.
La sintassi permette anche la definizione di interrogazioni con clausola `having` senza una corrispondente clausola `group by`. In questo caso, l'intero insieme di righe è trattato come un unico raggruppamento, ma questo ha in generale un limitato campo di applicabilità, perché se la condizione non è soddisfatta il risultato sarà vuoto. Come la clausola `where`, anche la clausola `having` ammette come argomento una espressione booleana su predicati semplici. I predicati semplici sono normalmente confronti tra il risultato della valutazione di un operatore aggregato e una generica espressione; sintatticamente è ammessa anche la presenza diretta degli attributi argomento della `group by`, ma è preferibile raccogliere tutte le condizioni su questi attributi nell'ambito della clausola `where`, solo i predicati in cui compaiono operatori aggregati devono essere argomento della clausola `having`.

SQL mette a disposizione anche degli operatori insiemistici, simili a quelli disponibili nell'algebra relazionale. Gli operatori disponibili sono gli operatori di union (unione), intersect (intersezione) ed except (chiamato anche minus, differenza), di significato analogo ai corrispondenti operatori dell'algebra relazionale.
La sintassi per l'uso degli operatori insiemistici è la seguente:

SelectSQL { < `union` | `intersect` | `except` > [`all`] *SelectSQL* }

Gli operatori insiemistici, al contrario del resto del linguaggio, assumono come default di eseguire una eliminazione dei duplicati. Qualora nell'interrogazione si voglia adottare una diversa interpretazione degli operatori e si vogliano utilizzare gli operatori insiemistici che preservano i duplicati, sarà sufficiente utilizzare l'operatore con la parola chiave `all`.

SQL ammette anche l'uso di predicati con una struttura più complessa, in cui si confronta un valore (ottenuto come risultato di una espressione valutata sulla singola riga) con il risultato dell'esecuzione di una interrogazione SQL. L'interrogazione che viene usata per il confronto viene definita direttamente nel predicato interno alla clausola `where`. Si parla in questo caso di *interrogazioni nidificate*.
La soluzione offerta da SQL consiste nell'estendere, con le parole chiave `all` o `any`, i normali operatori di confronto relazionale (=, <>, <, >, <= e >=). La parola chiave `any` specifica che la riga soddisfa la condizione se risulta vero il confronto (con l'operatore specificato) tra il valore dell'attributo per la riga ed almeno uno degli elementi restituiti dall'interrogazione. La parola chiave `all` invece specifica che la riga soddisfa la condizione solo se tutti gli elementi restituiti dall'interrogazione nidificata rendono vero il confronto. La sintassi richiede la compatibilità di dominio tra l'attributo restituito dalla interrogazione nidificata e l'attributo con cui avviene il confronto.

Per rappresentare il controllo di appartenenza e di esclusione rispetto a un insieme, SQL mette a disposizione due appositi operatori, `in` e `not in`, i quali risultano del tutto identici a `= any` e `<> all`.

Una interpretazione molto semplice ed intuitiva delle interrogazioni nidificate consiste nell'assumere che l'interrogazione nidificata venga eseguita prima di analizzare le righe della interrogazione esterna. Il risultato della interrogazione può essere salvato in una tabella temporanea e il controllo sulle righe della interrogazione esterna può essere fatto accedendo direttamente al risultato temporaneo.
Talvolta però l'interrogazione nidificata fa riferimento al contesto della interrogazione che la racchiude; tipicamente ciò accade tramite una variabile definita nell'ambito della query più esterna ed usata nell'ambito della query più interna (si parla di un *passaggio di binding* da un contesto all'altro). La presenza del meccanismo di passaggio dei binding arricchisce il potere espressivo di SQL . In questo caso l'interpretazione

semplice non vale più; Prima si costruisce il prodotto cartesiano delle tabelle e successivamente si applicano a ciascuna riga del prodotto le condizione che compaiono nella clausola `where`. L'interrogazione nidificata è un componente della clausola `where` e dovrà anch'essa essere valutata separatamente per ogni riga prodotta nella valutazione della query esterna.
Per quanto riguarda la visibilità (o *scope*) delle variabili SQL vale la restrizione che una variabile è usabile solo nell'ambito della query in cui è definita o nell'ambito di una query nidificata (a un qualsiasi livello) all'interno di essa. Se una interrogazione possiede interrogazioni nidificate allo stesso livello (su predicati distinti), le variabili introdotte nella clausola `from` di una query non potranno essere usate nell'ambito dell'altra query.
L'operatore logico `exists` ammette come parametro una interrogazione nidificata e restituisce il valore vero solo se l'interrogazione fornisce un risultato non vuoto (corrispondente al quantificatore esistenziale della logica). Questo operatore può essere usato in modo significativo solo quando si ha un passaggio di binding tra l'interrogazione esterna e quella nidificata.
La parte di Data Manipulation Language comprende i comandi per interrogare e modificare il contenuto della base di dati. I comandi che permettono di modificare la base di dati sono `insert`, `delete` ed `update`.

Il comando di inserimento di righe nella base di dati presenta due sintassi alternative:

`insert into` *NomeTabella* [*ListaAttributi*] < `values (` *ListaDiValori* `)` | *SelectSQL* >

La prima forma permette di inserire singole righe all'interno delle tabelle. L'argomento della clausola `values` rappresenta esplicitamente i valori degli attributi della singola riga. La seconda forma invece permette di aggiungere degli insiemi di righe, estratti dal contenuto della base di dati.
Se in un inserimento non vengono specificati i valori di tutti gli attributi della tabella, agli attributi mancanti viene assegnato il valore di default, o in assenza di questo, il valore nullo. Se l'inserimento viola un vincolo di *not null* definito sull'attributo, l'inserimento viene rifiutato. La corrispondenza tra gli attributi della tabella e i valori da inserire è data dall'ordine in cui compaiono i termini nella definizione della tabella.

Il comando `delete` elimina righe dalle tabelle della base di dati, seguendo la semplice sintassi:

`delete from` *NomeTabella* [`where` *Condizione*]

Quando la condizione argomento della clausola `where` non viene specificata, il comando cancella tutte le righe dalla tabella, altrimenti vengono rimosse solo le righe che soddisfano la condizione. Qualora esista un vincolo di integrità referenziale con politica di `cascade` in cui la tabella viene referenziata, allora la cancellazione di righe dalla tabella può comportare la cancellazione di righe appartenenti ad altre tabelle.
La *Condizione* rispetta la sintassi della `select`, per cui possono comparire al suo interno anche interrogazioni nidificate che fanno riferimento ad altre tabelle.

Il comando di `update` presenta una sintassi leggermente più complicata:

```
update NomeTabella
      set Attributo = < Espressione | SelectSQL | null | default >
      { , Attributo = < Espressione | SelectSQL | null | default > }
[ where Condizione ]
```

Il comando di `update` permette di aggiornare uno o più attributi delle righe di *NomeTabella* che soddisfano l'eventuale *Condizione*. Se la condizione non compare, come al solito si suppone di default un valore *vero* e si esegue la modifica su tutte le righe.

Per specificare ulteriori vincoli, SQL-2 ha introdotto la clausola `check`, con la seguente sintassi:

`check ( `*Condizione*` )`

Le condizioni che si possono usare sono quelle che possono apparire come argomento della clausola `where` di una interrogazione SQL. La condizione deve essere sempre verificata affinché la base di dati sia corretta.

Grazie alla clausola `check` è possibile definire anche un ulteriore componente dello schema di una base di dati, le *asserzioni*. Le asserzioni rappresentano dei vincoli che non sono però associati a nessun attributo o tabella in particolare, ma appartengono direttamente allo schema.
Mediante le asserzioni è possibile esprimere vincoli su più tabelle o vincoli che richiedono che una tabella abbia una cardinalità minima. Le asserzioni possiedono un nome, tramite il quale possono essere eliminate esplicitamente dallo schema con l'istruzione `drop`.
La sintassi per la definizione delle asserzioni è:

`create assertion` *NomeAsserzione* `check (` *Condizione* `)`

Ogni vincolo d'integrità, definito mediante `check` o tramite asserzione, è associato a una politica di controllo che specifica se il vincolo è immediato o differito.
I vincoli immediati sono verificati immediatamente dopo ogni modifica della base di dati, mentre i vincoli differiti sono verificati solo al termine dell'esecuzione di una serie di operazioni (che costituisce una transazione). Quando un vincolo immediato non è soddisfatto, l'operazione di modifica che ha causato la violazione è stata appena eseguita e il sistema può "disfarla"; questo modo di procedere è chiamato *rollback parziale*. Tutti i vincoli predefiniti sono verificati in modo immediato e la loro violazione causa un rollback parziale. Quando invece si rileva una violazione di un vincolo differito al termine della transazione, non c'è modo di individuare l'operazione che ha causato la violazione, e perciò diventa necessario disfare l'intera sequenza di operazioni che costituiscono la transazione; in questo caso si parla di *rollback*.
Grazie a questi meccanismi, l'esecuzione di un comando di modifica dell'istanza di una base di dati che soddisfa tutti i vincoli produrrà sempre un'istanza della base di dati che pure soddisfa tutti i vincoli (si dice anche che lo stato della base di dati è *consistente*).
È possibile cambiare il tipo di controllo associato al vincolo, assegnandogli la modalità immediata o differita. Ciò avviene tramite i comandi:

`set constraints [` *NomeVincolo* `] immediate`
`set constraints [` *NomeVincolo* `] deferred`

Le viste vengono definite in SQL associando un nome ed una lista di attributi al risultato dell'esecuzione di una interrogazione. Si definisce una vista utilizzando il comando:

`create view` *NomeVista* `[ (` *ListaAttributi* `) ] as` *SelectSQL*
`[ with [ local | cascaded ] check option ]`

L'interrogazione SQL deve restituire un insieme di attributi pari a quelli contenuti nello schema; l'ordine nella clausola *select* deve corrispondere all'ordine degli attributi dello schema della vista.

Su certe viste è permesso effettuare operazioni di modifica, che verranno tradotte negli opportuni comandi di modifica al livello delle tabelle di base da cui la vista dipende.
Lo standard SQL permette che una vista sia aggiornabile solo quando una sola riga di ciascuna tabella di base corrisponde a una riga della vista. I sistemi commerciali tipicamente considerano una vista aggiornabile solo se è definita su una sola tabella; qualche sistema chiede pure che l'insieme di attributi della vista contenga almeno una chiave primaria della tabella. La clausola `check option` può essere utilizzata solo

nel contesto di queste viste. La `check option` specifica che possono essere ammessi aggiornamenti solo sulle righe della vista e dopo gli aggiornamenti le righe devono continuare ad appartenere alla vista.
Nel caso in cui una vista sia definita in termini di altre viste, l'opzione `local` o `cascaded` specifica se il controllo sul fatto che le righe vengono rimosse dalla vista debba essere effettuato solo all'ultimo livello (per cui si controlla solo che la modifica non faccia violare la condizione della vista più esterna) o se deve essere propagato a tutti i livelli di definizione (per cui si controlla che le righe su cui si apportano le modifiche non scompaiano dalla vista, a causa della violazione di una qualsiasi delle condizioni delle viste coinvolte); l'opzione di default è quella di `cascaded`.

SQL prevede alcune famiglie di funzioni. I sistemi spesso arricchiscono l'insieme di funzioni di ogni famiglia.

- Funzioni temporali: `current_date`, `current_time`, `current_timestamp`
- Funzioni di manipolazione di stringhe: `char_length`, `lower`, `upper`, `substring`
- Funzioni di conversioni di dominio: `cast` (es. `cast (Data as char(10)`)
- Funzioni condizionali:
  - `coalesce`: accetta una sequenza di espressioni e restituise il primo diverso da null: ad esempio:
    ```
    select cognome, coalesce (Dip,'Ignoto') from Impiegato
    ```
  - `nullif`: accetta una espressione e un valore costante; se l'espressione è pari al valore costante restituisce NULL, ad esempio:
    ```
    select cognome, nullif (Dip,'Ignoto') from Impiegato
    ```
  - `case`: permette di specificare espressioni condizionali che generano i valori sulla base di generici predicati SQL, con la seguente sintassi:
    ```
    case when Condizione then Espressione
         { when Condizione then Espressione }
         else Espressione
    end
    ```
- Funzioni per la formattazione dell'output, funzioni matematiche e funzioni di accesso ai servizi del sistema operativo che non fanno parte di SQL-2 ma che sono spesso offerte dalle diverse implementazioni di SQL.

Controllo dell'accesso

La presenza di meccanismi di protezione dei dati riveste grande rilevanza in molte applicazioni.
SQL prevede innanzitutto che ogni utente sia identificato in modo univoco dal sistema.
L'identificazione dell'utente può sfruttare le funzionalità del sistema operativo o essere indipendente.
Le risorse che il sistema protegge sono normalmente tabelle, ma si può proteggere un qualsiasi componente del sistema, come gli attributi di una tabella, viste e domini. Di regola l'utente che crea la risorsa ne è il proprietario ed è autorizzato a compiere su di essa qualsiasi operazione.
Il sistema basa il controllo di accesso su un concetto di privilegio. Gli utenti possiedono dei privilegi di accesso alle risorse del sistema; ogni privilegio è caratterizzato dai seguenti parametri:

1. la risorsa cui si riferisce;
2. l'utente che concede il privilegio;
3. l'utente che riceve il privilegio;
4. l'azione che viene permessa sulla risorsa;
5. se il privilegio più essere trasmesso o meno ad altri utenti

I privilegi disponibili sono i seguenti:

- `insert`: permette di inserire un nuovo oggetto nella risorsa (si può applicare solo alle tabelle e alle

viste)

- `update`: permette di aggiornare il valore di un oggetto (vale per le tabelle, le viste e gli attributi)
- `delete`: permette di rimuovere oggetti dalla risorsa (vale solo per le tabelle e le viste)
- `select`: permette di leggere la risorsa, ovvero utilizzarla nell'ambito di una interrogazione (vale per le tabelle, le viste e gli attributi)
- `references`: permette che venga fatto un riferimento a una risorsa nell'ambito della definizione dello schema di una tabella. Può essere associato solo a tabelle ed a specifici attributi
- `usage`: permette che venga usata la risorsa, per esempio nell'ambito della definizione dello schema di una tabella, ma solo per risorse come i domini.

Il privilegio di effettuare un `drop` o un `alter` di un oggetto non può essere concesso, ma rimane di competenza del creatore dell'oggetto stesso.
I privilegi vengono concessi o revocati tramite le istruzioni `grant` e `revoke`.
La sintassi del comando `grant` è la seguente:

```
grant Privilegi on Risorsa to Utenti [ with grant option ]
```

Il comando permette di concedere i *Privilegi* sulla *Risorsa* agli *Utenti*.

Il comando `revoke` fa invece l'inverso: sottrae a un utente i privilegi che gli erano stati concessi:

```
revoke Privilegi on Risorsa from Utenti [ restrict | cascade ]
```

Tra i privilegi che possono essere rimossi, oltre a quelli che possono comparire come argomento del comando di `grant`, vi è pure il privilegio `grant option`, derivante dall'uso dell'opzione `with grant option`. L'unico utente che può sottrarre privilegi ad un altro utente è l'utente che aveva concesso i privilegi in primo luogo. Il comando `revoke` può eliminare tutti i privilegi che erano stati concessi, o limitarsi a revocarne un sottoinsieme.
L'opzione `restrict` è il default e specifica che il comando non deve essere eseguito qualora la revoca dei privilegi all'utente comporti qualche altra revoca di privilegi, come può capitare quando l'utente ha ricevuto i privilegi con la `grant option` ed ha propagato il privilegio ad altri utenti, o come capita quando il privilegio che si vuole revocare è stato usato per la definizione di una vista o di una tabella dell'utente. Con l'opzione `cascade`, si forza l'esecuzione del comando: così tutti i privilegi propagati vengono revocati e tutti gli elementi della base di dati che erano stati costruiti sfruttando questi privilegi vengono rimossi.

Transazioni
Una transazione identifica una unità elementare di lavoro svolta da una applicazione, cui si vogliono associare particolari caratteristiche di correttezza, robustezza e isolamento.

Un sistema che mette a disposizione un meccanismo per la definizione e l'esecuzione di transazioni viene detto *sistema transazionale*.

Una transazione può essere definita sintatticamente: ogni transazione, quale che sia il linguaggio di programmazione in cui essa è scritta è incapsulata all'interno di due comandi: `begin transaction` ed `end transaction`.
All'interno del codice delle transazioni, possono comparire due istruzioni particolari, `commit work` e `rollback work`, cui facciamo riferimento usando i due termini *commit* e *abort*, che indicano l'azione associata alla rispettiva istruzione.
L'effetto di questi due comandi è decisivo per l'esito della transazione la quale "va a buon fine" solo a

seguito di un commit, mentre non ha nessun effetto tangibile sulla base di dati quando viene eseguito l'abort.

Tutto il codice che viene eseguito all'interno di una coppia di comandi `begin transaction-end transaction` gode di proprietà particolari, le cosiddette *proprietà acide* delle transazioni: *atomicità*, *consistenza*, *isolamento* e *persistenza* (Atomicity, Consistency, Isolation, Durability).

- L'*atomicità* rappresenta il fatto che una transazione è una unità *indivisibile* di esecuzione: o vengono resi visibili tutti gli effetti di una transazione, oppure la transazione non deve avere alcun effetto sulla base di dati, con un approccio "tutto o niente"
- La *consistenza* richiede che l'esecuzione della transazione non violi i vincoli di integrità definiti sulla base di dati.
- L'*isolamento* richiede che l'esecuzione di una transazione sia indipendente dalla contemporanea esecuzione di altre transazioni.
- La *persistenza* invece richiede che l'effetto di una transazione che ha eseguito il commit correttamente non venga più perso.

Progettazione di Basi di Dati

PROGETTAZIONE CONCETTUALE

- Raccolta e analisi dei requisiti -> Modello Entity-Relationship
  - Verifica di *Correttezza*: uno schema è corretto se utilizza propriamente tutti i costrutti messi a disposizione dal modello concettuale di riferimento (errori sintattici o semantici).
  - Verifica di *Completezza*: se rappresenta tutti i dati di interesse e se tutte le operazioni possono essere eseguite a partire dai concetti descritti nello schema.
  - Verifica di *Leggibilità*: se rappresenta i requisiti in maniera naturale e facilmente comprensibile.
  - Verifica di *Minimalità*: se tutte le specifiche sui dati sono rappresentate una sola volta nello schema, e non esistono quindi delle ridondanze, dei concetti che possono essere derivati da altri.
- Documentazione di supporto
  - Glossario dei termini
  - Regole Aziendali
  - Operazioni sui dati

PROGETTAZIONE LOGICA

- Ristrutturazione Modello E-R
  - Analisi delle ridondanze: si decide se eliminare o mantenere eventuali ridondanze.
  - Eliminazione generalizzazioni: tutte le generalizzazioni presenti nello schema vengono analizzate e sostituite da altri costrutti.
  - Partizionamento / Accorpamento Entità e Associazioni: si decide se è opportuno partizionare concetti dello schema in più concetti, o viceversa accorpare concetti separati in un unico concetto (ad es: eliminazione attributi multivalore)
  - Scelta identificatori primari: si seleziona un identificatore per le entità che ne hanno più d'uno.
- Traduzione

PROGETTAZIONE FISICA

Normalizzazione

La *prima forma normale* stabilisce che gli attributi delle relazioni sono definiti su valori atomici e non su valori complessi quali insiemi e relazioni.

Una relazione *r* è in *forma normale di Boyce e Codd* se per ogni dipendenza funzionale (non banale) *X* -> *Y* definita su di essa, *X* contiene una chiave *K* di *r*, cioè *X* è una superchiave per *r*.

Data una relazione che non soddisfa la forma normale di Boyce e Codd è possibile, in molti casi, sostituirla con due o più relazioni normalizzate, attraverso un processo detto di *normalizzazione*: se una relazione rappresenta più concetti indipendenti, allora va decomposta in relazioni più piccole, una per ogni concetto.

Possiamo dire che *r* si decompone *senza perdita* su due relazioni se l'insieme degli attributi comuni alle due relazioni è chiave per almeno una delle relazioni decomposte. In ogni decomposizione, ciascuna delle dipendenze funzionali dello schema originario dovrebbe coinvolgere attributi che compaiono tutti insieme in uno degli schemi decomposti. In questo modo è possibile garantire sullo schema decomposto il soddisfacimento degli stessi vincoli il cui soddisfacimento è garantito dallo schema originario.

Esistono schemi che violano la forma normale di Boyce e Codd per i quali non esiste alcuna decomposizione che conservi le dipendenze. In questi casi, si ricorre a una condizione meno restrittiva che definisce una nuova forma normale (sempre ottenibile): una relazione *r* è in *terza forma normale* se, per ogni dipendenza funzionale (non banale) *X* -> *Y* definita su di essa, almeno una delle seguenti condizioni è verificata:

- *X* contiene una chiave *K* di *r*,
- ogni attributo in *Y* è contenuto in almeno una chiave di *r*.