Appunti di “Basi di dati”

Indice

Database Management System (DBMS)

Definire una rappresentazione delle informazioni equivale a modellare la realtà. Un sistema informativo contiene informazioni sotto forma di dati, ossia un insieme di simboli memorizzati. Un sistema informativo deve organizzare le informazioni che contiene e può scegliere 2 approcci:

1. Considerare i dati come un elemento centrale e sviluppare attorno i vari applicativi;
2. Basarsi prima sui processi e poi sui dati che devono essere adoperati.

I sistemi salvano i dati in una base di dati (database), ossia una risorsa integrata unica condivisa dai vari applicativi del sistema. Perciò è fondamentale che una base di dati sia capace di gestire la concorrenza e il controllo (non tutte le applicazioni hanno il permesso di vedere tutti i dati) degli accessi.

Chiamiamo Database Management System, o DBMS, i vari sistemi di gestione delle basi di dati. Per usare un DBMS dobbiamo prima di tutto definire lo schema, o rappresentazione, della base di dati. Lo schema è diviso in tre livelli:

1. **Logico:** descrizione della base di dati nella sua interezza;
2. **Esterno:** descrizione dei dati accessibili dall'applicativo;
3. **Fisico:** descrizione delle strutture dati usate per rappresentare i dati.

Noi ci concentreremo solo sul livello logico. Grazie al modello a 3 livelli possiamo ottenere indipendenza fisica, in quanto le strutture dati sottostanti vengono completamente astratte, e logica, in quanto i vari schemi esterni sono indipendenti tra di loro.

I DBMS utilizzano 2 linguaggi per operare sui dati:

1. **DDL (Data Definition Language):** utilizzato per definire nuovi dati all'interno della base (comandi tipo `CREATE`, `DROP` e `ALTER`)
2. **DML (Data Manipulation Language):** utilizzato per manipolare o interrogare (effettuare *query*) la base (comandi tipo `SELECT`, `INSERT`, `UPDATE` e `DELETE`)

Il modello relazionale

I database usano diversi modelli logici per organizzare i dati. Uno dei possibili modelli, ed il più usato oggi, è il modello relazionale.

Il modello relazionale è stato inventato dall'informatico britannico E. Codd negli anni '70. È stato messo sul mercato dalla Oracle negli anni '80 e raggiunse successo commerciale nel 1985. A differenza dei due modelli precedenti, quello reticolare basato su grafo con archi e nodi e quello gerarchico della rivale IBM basato su strutture ad albero, il modello relazionale si basa sul concetto di relazione.

Struttura del modello relazionale

Definizione. ***Relazione*** Una relazione R su un insieme di domini $D_1, \dots, D_n$ è formata da due parti: una intestazione e un corpo.

1. Intestazione: formata da un insieme fissato e senza ordine di attributi. Ogni attributo è descritto dalla coppia $\langle nome\_attributo, nome\_dominio \rangle$ e corrisponde a esattamente uno solo dei domini D_i. Il nome di ciascun attributo deve essere distinto, i domini in sé non devono essere distinti però.
2. Corpo: formato da un insieme di tuple disordinate. Ogni tupla è un insieme unico di coppie $\langle nome\_attributo, valore \rangle$, una per ogni attributo dell'intestazione, dove *valore* è un elemento del relativo dominio.

Definizione. ***Grado di una relazione*** Diremo grado di una relazione il numero di attributi che contiene.
Definizione. ***Cardinalità di una relazione*** Diremo cardinalità di una relazione il numero di tuple contenuto nel corpo

Più informalmente una relazione può essere vista come una tabella dove l'intestazione corrisponde agli attributi e le varie righe alle tuple.

ID	Nome	Età	Corso
123	Mario Rossi	19	Informatica 1
456	Francesca Sgarbi	21	Fisica 3
789	Leonardo Bianconi	21	Elettronica

Per questo motivo confonderemo il termine "relazione" con "tabella". Una notazione utile che utilizzeremo è la seguente: sia t una tupla, $t[X] = valore$ indica indica il valore che la tupla assume per l'insieme di attributi X.

Solitamente una singola relazione non è sufficiente non è sufficiente a organizzare tutti i dati relativi ad una applicazione. Perciò i database sono solitamente formati da diverse relazioni che contengono valori comuni. Il modello relazionale viene detto *value-based* poiché i riferimenti tra le varie tabelle avviene tramite i valori che appaiono nelle tuple. I modelli precedenti rappresentavano queste relazioni tramite dei puntatori e venivano chiamati *pointer-based*. Rispetto ai sistemi *pointer-based*, il modello relazionale possiede numerosi vantaggi tra cui:

1. La rappresentazione logica non fa alcun riferimento a quella fisica rendendo le tabelle indipendenti dalla loro rappresentazione;
2. Vengono rappresentati solo i dati rilevanti agli applicativi;
3. Poiché tutta l'informazione è contenuta nei valori, è relativamente semplice trasferire i dati da un contesto all'altro.

Per rappresentare valori inesistenti o sconosciuti, arricchiamo ogni dominio delle relazioni del database con il valore `null`.

Ricapitoliamo le definizioni del modello relazionale distinguendo gli schemi, rappresentazione logica simile alle classi in OOP, dalle istanze, materializzazione degli schemi simile agli oggetti in OOP.

Definizione. ***Schema di una relazione*** Definiremo schema di una relazione un simbolo, chiamato nome della relazione, ed un insieme di attributi X. Il tutto viene solitamente indicato con $R(X)$. Il nome della relazione deve essere univoco.

Definizione. ***Schema di un database*** Definiremo schema di un database l'insieme di schemi di relazioni.

$$\mathbf{R} = \{R_1(X_1), \dots, R_n(X_n)\}$$

Definizione. ***Istanza di una relazione (relazione)*** Un'istanza di una relazione con schema $R(X)$ è un insieme r di tuple su X.

Definizione. ***Istanza di un database (database)*** Un'istanza di un database con schema $\mathbf{R} = \{R_1(X_1), \dots, R_n(X_n)\}$ è un set di relazioni $\mathbf{r} = \{r_1, \dots, r_n\}$ dove ogni r_i, $1 \leq i \leq n$, è una relazione con schema $R_i(X_i)$.

La normalità di una relazione è una proprietà che affronteremo in dettagli più avanti. Per ora limitiamoci a definire la prima forma normale poiché noi lavoreremo con tabelle che soddisfano questa forma.

Definizione. ***Relazione in prima forma normale*** Una relazione si dice in prima forma normale se ogni tupla contiene solo elementi atomici.

Vincoli di integrità

La struttura del modello relazione ci permette di organizzare le informazioni di interesse per le nostre applicazioni. In molti casi, però, non è vero che ogni insieme di tuple che rispetta lo schema rappresenta informazioni corrette. In un database, è necessario che l'informazione sia sempre corretta. Perciò è stato introdotto il concetto di vincolo. Ogni vincolo è un predicato che associa un valore booleano ad ogni istanza. Un database è considerato corretto, o legale, se rispetta tutti i vincoli specificati. Esistono diversi tipi di vincoli:

1. Vincoli sui valori nulli: presenza di campi necessari;
2. Vincoli di integrità referenziale;
3. Vincoli di chiave;
4. Vincoli generici sulle tuple.

Vincoli di chiave

Iniziamo con il definire il concetto di chiave. Intuitivamente, possiamo vedere la chiave come un insieme di attributi che identifica in modo univoco le varie tuple di una relazione. Più precisamente:

Definizione. ***Superchiave*** Un insieme di attributi K è detto superchiave di un relazione R se R non contiene due tuple distinte tali che $t_1[K] = t_2[K]$.

Definizione. ***Chiave*** Un insieme di attributi K è detto chiave per una relazione R se esso è la superchiave minima.

Teorema. Ogni relazione R possiede sempre almeno una chiave.

Il teorema sopra garantisce il fatto che si potrà sempre accedere a tutti i valori di una relazione legale in modo disambiguo. In più, permette l'effettiva realizzazione dei riferimenti tra più tabelle che stanno alla base del modello *value-based*.

Poiché possono esistere più di una chiave per relazione, solitamente viene definita una chiave primaria (indicata sottolineandone gli attributi). Valori nulli per le chiavi primarie sono vietati mentre generalmente sono permessi per le altre. Inoltre, la maggior parte dei riferimenti tra relazioni usa le chiavi primarie.

Linguaggi di interrogazione

I linguaggi di interrogazione, come abbiamo visto, ci permettono di accedere ai dati all'interno della base di dati. Esistono due tipi di linguaggi di interrogazione:

1. Linguaggi formali: algebra relazionale, calcolo relazionale, datalog
2. Linguaggi *commerciali*: SQL

Algebra relazionale

Proposta inizialmente da Codd, è un linguaggio formale funzionale per formulare query (interrogazioni). È basato su 5 operazioni fondamentali: selezione, proiezione, unione, differenza e prodotto cartesiano.

Definizione. ***Relazioni compatibili*** Due relazioni si dicono compatibili se sono dello stesso grado e (nei sistemi) i domini sono ordinatamente dello stesso tipo (anche il nome deve combaciare).

Definizione. ***Selezione*** Indicata $\sigma_{predicato} R$, la selezione restituisce una nuova tabella senza nome con schema identico alla tabella di partenza e istanza le tuple che soddisfano il predicato di selezione.

Definizione. ***Proiezione*** Indicata $\Pi_{a_i \dots} R$, la proiezione restituisce una tabella senza nome con schema contenente solo gli attributi specificati e istanza la restrizione delle tuple sugli attributi selezionati. Formalmente la proiezione elimina eventuali duplicati che si vanno a formare, in SQL invece è su richiesta.

Definizione. ***Unione*** Indicata $R_1 \cup R_2$, l'unione è definita solo tra relazioni compatibili e restituisce una tabella priva di nome con schema invariato e istanza l'unione delle istanze dei due operandi.

Definizione. ***Differenza*** Indicata $R_1 \setminus R_2$, la differenza è definita solo tra relazioni compatibili e restituisce una tabella priva di nome con schema invariato e istanza la differenza delle tuple delle due relazioni.

Definizione. ***Prodotto cartesiano*** Indicato $R \times S$, il prodotto cartesiano restituisce una tabella priva di nome con schema l'unione degli schemi degli operandi e istanza tutte le possibili coppie di tuple degli operandi. Per rendere non ambigui gli attributi con lo stesso nome, si può prefissare il nome della relazione, ad esempio $R.a_1$ e $S.a_1$.

Sono permessi anche i seguenti operatori non algebrici per rendere le espressioni algebriche più leggibili:

1. Assegnamento $=$ o $\leftarrow$: definisce una etichetta ad una selezione
2. Ridenominazione $\rho_{a_1 \leftarrow a_2} R$: permette di modificare i nome degli attributi.

Le operazioni sono associate a sinistra e la precedenza è analoga a quella dell'algebra elementare.

Operazioni derivate

Le operazioni derivate sono definite in base alle operazioni fondamentali. Esse sono abbreviazioni di operazioni molto comuni. Ne vederemo tre: intersezione, join (con le sue varianti) e divisione.

Definizione. ***Intersezione*** Indicata $R_1 \cap R_2$, equivale a:

$$R \setminus (R \setminus S)$$

L'intersezione è definita solo tra relazioni compatibili.

Definizione. ***Join*** Indicato $R \underset{predicato}{\bowtie} S$, equivale a:

$$\sigma_{predicato}(R \times S)$$

Esso restituisce una tabella priva di nome con schema pari alla concatenazione degli schemi di R e S e istanza pari all'insieme ottenuto concatenando le tuple di R e S che soddisfano il predicato.

Definizione. ***Equi-Join*** È un caso particolare di join, dove il predicato ammette solo confronti di uguaglianza.

Definizione. ***Join naturale*** Equivale ad un equi-join di tutti gli attributi omonimi. Nella notazione, si omette il predicato e nel risultato si eliminano le colonne ripetute.

Definizione. ***Semi-join*** Indicato con $R \underset{predicato}{\ltimes} S$, equivale a

$$\Pi_{R.\star}(R \underset{predicato}{\ltimes} S)$$

Definizione. ***Semi-join naturale*** Indicato $R \ltimes S$, equivale a:

$$\Pi_{R.\star}(R \bowtie S)$$

Definizione. ***Divisione*** Indicato $R \div S$, equivale a

$$\Pi_{R \setminus S} R \setminus \Pi_{R \setminus S}((\Pi_{R \setminus S} R \times S) \setminus S)$$

Con lo schema di S contenuto in quello di R. Seleziona le tuple di R che contengono tutte le tuple di S

Il join ha la stessa precedenza del prodotto cartesiano.

Ottimizzazione delle organizzazioni

Possiamo ridurre le espressioni applicando alcune semplificazioni. Un modo utile per visualizzare le espressioni e le relative semplificazioni è visualizzare il tutto come un albero delle operazioni (vedi LEA).

1. **Eliminazione dei prodotti cartesiani:** Possiamo sostituire un prodotto cartesiano seguito da selezione con join.
2. **Push della selezione rispetto al join:** Possiamo trasformare $\sigma_p(R \bowtie_q S)$ in $(\sigma_p R) \bowtie_q S$. Questa semplificazione si può fare se e solo se il predicato della selezione coinvolge solo attributi di R (o analogamente di S).
3. **Push della proiezione rispetto al join:** Analogo al precedente, trasformiamo $\Pi_L(R \bowtie_q S)$ in $\Pi_L(\Pi_{LR \cup JR} R \bowtie_q \Pi_{LS \cup JS} S)$ dove LR e LS sono gli attributi di R ed S rispettivamente e JR e JS gli attributi di R ed S rispettivamente coinvolti nel join.
4. **Idempotenza della selezione:** $\sigma_{p \wedge q} R = \sigma_p(\sigma_q R)$ Unendo questa regola con il push della selezione rispetto al join, possiamo rimuovere la restrizione sugli attributi.
5. **Idempotenza della proiezione:** $\Pi_L R = \Pi_L \Pi_{L'} R$ con $L \subseteq L'$
6. **Push della selezione rispetto all'unione:** $\sigma_p(R \cup S) = \sigma_p R \cup \sigma_p S$
7. **Push della selezione rispetto alla differenza:** $\sigma_p(R \setminus S) = \sigma_p R \setminus \sigma_p S$
8. **Push della proiezione con l'unione:** $\Pi_L(R \cup S) = \Pi_L R \cup \Pi_L S$. Attenzione: la proiezioni non commuta con le altre operazioni insiemistiche!
9. **Distributività della join rispetto all'unione**

Calcolo relazionale

È una famiglia di linguaggi formali dichiarativi per formulare interrogazioni. I due tipi principali sono il calcolo relazionale delle tuple (TRC) e il calcolo relazionale dei domini (DRC). A sua volta il TRC può avere dichiarazioni di range o tuple arbitrarie. Noi vedremo il TRC con tuple arbitrarie.

Le interrogazioni sono così strutturate: $\{t | p(t)\}$ con t le tuple per le quali è vera $p(t)$. La formula $p(t)$ è costituita tramite atomi:

- $t \in r$
- $t[A]\ COMP\ k$
- $t_1[A_1]\ COMP\ t_2[A_2]$

Dove per *COMP* intendiamo un connettivo logico. Valgono le solite equivalenze relative alle espressioni booleane.

Esistono delle formule, tipo $\{t|t \notin r\}$ producono un risultato infinito. Riduciamo perciò il calcolo relazionale ai domini attivi

Definizione. ***Restrizione ai domini attivi*** Le tupe che soddisfano una formula possono essere composte solamente da valori che compaiono esplicitamente nella formula o in tuple di relazioni menzionate nella formula.

Il calcolo relazione è equivalente all'algebra. Infatti è possibile tradurre tutte le operazioni fondamentali dalla prima al calcolo relazionale.

1. Selezione: $\sigma_{A=1}r \equiv \{t|\exists t_1 \in r(t_1[A] = 1) \land t = t_1\}$
2. Proiezione: $\Pi_{AC}r \equiv \{t|\exists t_1 \in r(t[A, C] = t_1[A, C])\}$
3. Prodotto cartesiano: $r(A, B, C) \times s(D, E, F) = \{t|\exists t_1 \in r, t_2 \in s(t[A, B, C] = t_1[A, B, C]) \land t[D, E, F] = t_2[D, E, F]\}$
4. Unione: $r \cup s = \{t|(\exists t_1 \in r(t = t1) \lor (\exists t_2 \in s(t = t_2)))\}$
5. Differenza: $r \setminus s = \{t|(\exists t_1 \in r(t = t1) \land \neg(\exists t_2 \in s(t = t_2)))\}$

L'algebra relazione e la TRC hanno delle limitazioni sul tipo di interrogazioni che possiamo fare:

1. Possiamo solo estrarre valori già esistenti, non calcolarne di nuovi;
2. Interrogazioni inerentemente ricorsive come la chiusura transitiva.

Datalog

Da "Prolog per basi di dati". È un linguaggio di programmazione logica dichiarativo basato su regole e fatti.

Sintassi

Il datalog si basa sulla definizione di fatti di base (o *ground facts*) che corrispondono alle tuple delle nostre tabelle.

```
genitore(a, b)
genitore(b, c)
```

Il significato delle espressioni sopra è "a è genitore di b" e "b è genitore di c" rispettivamente.

Le regole, invece, definiscono come nuovi fatti verranno dedotti dai fatti base. La seguente espressione definisce 2 regole con il seguente significato:

```
antenato(X,Y) :- genitore(X,Y)
antenato(X,Y) :- genitore(X,Z), antenato(Z, Y)
```

1. `X` è un antenato di `Y` se `X` è genitore di `Y`
2. `X` è un antenato di `Y` se `X` è genitore di un `Z` e questo `Z` è antenato di `Y`

Ogni regola è divisa in due parti: la testa (o LHS) e il corpo (o RHS). Una regola si può generalmente interpretare così:

1. LHS è vero se RHS è vero
2. THS è vero se, per ogni letterale di RHS, tutte le sue variabili sono unificabili, ovvero sostituibili , con valori costanti che rendono vero il letterale.

Nella definizione di regole si possono usare operatori di confronto e funzioni aritmetiche.

> Consideriamo il fatto: `padre(X,Y) :- Persona(X, _, 'M'), genitori(X,Y)`. Esso è traducibile in algebra relazione con: $padre = \Pi_{1,5}\sigma_3(persona \bowtie_{1=1} genitori)$.

Le query, o *goals*, sono così definite:

```
?- antenato(a, X)
```

E considerando i precedenti fatti e regole ritornerebbe `b, c`.

Valutazione delle query

La valutazione parte da dei fatti "conosciuti" pari ai fatti del programma. Si procede enumerando ogni singola regola: se ogni atomo nel corpo della regola è nell'insieme dei fatti conosciuti, allora la testa viene aggiunta nella lista dei fatti noti. Il processo è ripetuto finché non si possono generare nuovi fatti.

L'insieme dei fatti di base è detto database estensionale, mentre l'insieme dei predicati appartenenti al corpo delle regole è detto database intensionale.

La valutazione di una query consiste nel cercare una tupla sull'intera base di dati e una sostituzione che unifichi le variabili. Le query che non contengono variabili restituiscono `true` o `false`.

Il potere espressivo del datalog

Il datalog senza negazione permette di rappresentare gli operatori di selezione, proiezione, prodotto cartesiano e unione dell'algebra:

- **Selezione, Proiezione e prodotto cartesiano:** vedi precedente esempio
- **Unione:** scriveremo più regole con la stessa testa.

Per rappresentare la differenza serve la negazione:

```
p(X,Y) :- r(X,Y), -s(X,Y)
```

La negazione, però, è un'operazione delicata in quanto può generale regole infinite. Per metteremo la negazione con la seguenti restrizioni:

1. Tutte le variabili di un letterale negato devono comparire anche in un letterale positivo del corpo della regola.
2. Non ci devono essere cicli di dipendenza tra letterali negati.

Abbiamo quindi raggiunto un livello di espressività almeno pari all'algebra relazionale. Il datalog, permette un livello di espressività ancora maggiore in quanto permette query ricorsive.

```
antenato(X,Y) :- genitore(X,Y)
antenato(X,Y) :- antenato(X,Z), genitore(Z,Y)
```

SQL

Acronimo di *Structured Query Language*, si compone di due parti:

1. DDL: definizioni di domini, tabelle, indici ecc.
2. DML: linguaggio di query, di modifica e comandi transazionali.

Data definition

Definizione degli schemi

Uno schema è una collezione di oggetti: domini, tabelle, indici, asserzione ecc. Uno schema ha un nome e un proprietario.

```
create schema [NomeSchema] [[authorization] Autorizzazione] {DefinizioneElemento}
```

Domini

I domini specificano i valori ammissibili per gli attributi. La definizione dei domini è analoga alla definizione dei tipi nei linguaggi di programmazione. Si dividono in due categorie:

- Elementari: predefiniti dallo standard
    1. Caratteri singoli o stringhe di lunghezza fissa:

       ```
       character [varying] [(Lunghezza)] [character set SetCaratteri]
       ```

       Si possono usare anche abbreviazioni come `char` e `varchar`.

    2. Bit singoli o in sequenza:

```
bit [varying] [(Lunghezza)]
```

Anche qui, possiamo usare l'abbreviazione `varbit`.

3. Domini numerici:

```
numeric [(Precisione [, Scala])]
decimal [(Precisione [, Scala])]
integer
smallint
```

4. Domini numerici approssimati:

```
float [(Precisione)]
real
double precision
```

5. Istanti temporali:

```
date(month, day, year)
time[(Precisione)] [with time zone] : (hour, minute, second)
timestamp[(Precisione)] [with time zone]
```

Quando si specifica la timezone, si hanno due ulteriori campi: `timezone_hour` e `timezione_minute`

6. Intervalli temporali:

```
interval PrimaUnitàDiTempo [to UltimaUnitàDiTempo]
```

Le unità di tempo sono divise in due gruppi: `year` e `month`; `day`, `hour`, `minute`, `second`.

- Definiti dall'utente. Ogni dominio è caratterizzato da un nome, dominio elementare, valore di default e un insieme di vincoli

```
create domain Nome as DominioElementare [Default] [Constraints]
```

Per definire il valore di default usiamo:

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

Dove `user` rappresenta il nome utente dell'utente che effettua il comando

Il valore `null` appartiene a tutti i domini con il significato di valore non noto per vari motivi: il valore esiste ma è ignoto al database, il valore è inapplicabile o non si sa se il valore è inapplicabile o meno.

Definizione di tabelle

Una tabella SQL consiste di un insieme di attributi ed un insieme di vincoli:

```
create table NomeTabella
( NomeAttr Dominio [Default] [Constraint]
  {, NomeAttr Dominio [Default] [Constraint]}...)
```

I vincoli sono condizioni che devono essere verificate da ogni istanza della base di dati. I vincoli intra-relazionali sono:

- `not null`
- `primary key`: definisce una chiave primaria (implica `not null`). Nel caso più attributi siano chiave primaria si può usare la seguente abbreviazione:

```
primary key(Attributo {, Attributo})
```

Nota bene: `Attr1 numeric 0 primary key, Attr2 numeric 0 primary key` è diverso da `Attr1 numeric 0, Attr2 numeric 0, primary key(Attr1, Attr2)`

- `unique`: simile a `primary key`, non implica `not null`
- `check`: sarà affrontato dopo

Le reazioni, invece, operano su una tabella interna in seguito a modifiche su una tabella esterna. Le violazioni delle relazioni tra tabelle possono essere introdotti da aggiornamenti o cancellazione di tuple. Le reazioni disponibili sono:

- `cascade`: propaga la modifica
- `set null`
- `set default`
- `no action`

La reazione può dipendere dall'evento:

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

Gli attributi descritti come `foreign key` nella tabella figlia devono presentare valori presenti come valori di chiave nella tabella padre. Per mantenere l'integrità relazionali usiamo:

- `references`: per un solo attributo, da specificare dopo il dominio
- `foreign key(Attributo {, Attributo}) references...` per uno o più attributi

Per velocizzare l'accesso ai dati, possiamo creare degli indici:

```
create [unique] index Nome on Tabella(Attributo)
```

Modifica degli schemi

Gli schemi possono essere modificati dopo la loro creazione. Abbiamo a disposizione 2 comandi:

- `alter <domain | table | column...>` modifica oggetti persistenti
- `drop <schema | domain | table | view | assertion> Nome [restrict|cascade]` cancella oggetti dallo schema
  - `restrict`: impedisce il `drop` se gli oggetti comprendono istanze
  - `cascade`: applica gli oggetti a tutti gli oggetti collegati

I cataloghi

Un catalogo contiene il dizionario dei dati, ovvero la descrizione relazionali e della struttura dei dati contenuti nel database. Ogni sistema ha una differente struttura. Lo standard ha definito due livelli:

1. `Definition_schema`: composto da tabelle, non vincolante
2. `Information_schema`: composto dalle viste, vincolante

Nello `information_schema` compaiono viste come `domains`, `domains_constraints`, `tables`, `views`, `comlumns`. Ad esempio, la vista `columns` può avere uno schema con attributi `table_name`, `column_name` ecc.

Il catalogo è normalmente riflessivo, ossia le strutture del catalogo sono definite nel catalogo stesso. Ogni comando del DDL viene quindi realizzato da opportuni comandi DML che operano sullo schema del catalogo. Ciò, però, non rende il DDL inutile.

Query semplici

Le interrogazioni SQL sono dichiarative, l'utente specifica quale informazione è di interesse, ma non come estrarla dai dati. Le interrogazioni vengono tradotte dall'ottimizzatore nel linguaggio procedurale interno al DBMS.

Per esprimere le query, usiamo 3 clausole in congiunzione: la clausola `select`, la clausola `from` e la clausola `where`. Ognuna di queste ha una sua sintassi particolare nella quale non entreremo nel dettaglio. Unite insieme formano.

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

In SQL i duplicati vengono mantenuti, per rimuoverli bisogna usare `select distinct`.

Un confronto con `null` restituisce `unknown`. Per verificare se un attributo è nullo o no si usa `Attr is [not] null`.

Comandi di modifica

Tutte le istruzioni di modifica possono operare su un insieme di tuple. Il comando può contenere una condizione, nella quale è possibile fare accesso ad altre tabelle,

- `insert into Tabella [(Attributi...)] <values(Valori...) | select_query>`
- `delete from Tabella [where Confizione]`
- `update Tabella set Attr = <Expr | SQL | null | default>... [where Cond]`

Query con `join`

Una semplice query che utilizza il predicato di join è:

```
select distinct Nome
from Studente, Esame, Corso
where Studente.Matr = Esame.Matr
  and Corso.CodCorso = Esame.CodCorso
  and Titolo like 'ge%' and Voto = 30
```

Esiste anche una sintassi che specifica esplicitamente il join nella clausola `from`:

```
select Attr {, Attr}
from Tabella {[TipoJoin] join Tabella on Condizione}
[where Condizione]
```

Il `join` può essere:

- `Inner`: default, normalmente omesso;
- `right`, `left`, `full` sono i join esterni:
  - `right` preserva tutte le tupla del primo operando, estendendo con `null` le tuple del primo che non hanno corrispondenza nel secondo;
  - `left` è simmetrico al primo;
  - `full` è l'unione tra i comportamenti di `left` e `right`: le tuple del primo che non hanno corrispondenza nel secondo sono estese con `null` e viceversa.

Entrambe le sintassi, inoltre, ci permettono di ridefinire con un alias i membri del `join` aggiungendo `as Alias` dopo il nome. Ciò può tornare utile quando si deve usare la stessa tabella più di una volta all'interno della stessa query:

```
select X.Nome, X.MatrMgr, Y.Matr, Y.Nome
from Impiegato as X, Impiegato as Y
where X.MatrMgr = Y.Matr
  and Y.nome = 'Giorgio'
```

Query complesse

Abbiamo 4 tipi diversi di query complesse:

1. Query con ordinamento
2. Query con aggregazioni
3. Query con raggruppamento
4. Query binarie
5. Query nidificate

Query·con·ordinamento

L'ordinamento viene definito dalla clausola `order by` che compare in coda alla query. Come si può intuire, ordina le righe del risultato:

```
order by Attr [asc | desc] {, Attr·[asc·|·desc]}
```

Query con aggregazioni

Le interrogazioni con funzioni aggregate non possono essere rappresentate in algebra. Il risultato di una query con funzioni aggregate dipende dalla valutazione del contenuto di un insieme di righe. Sono offerti 5 operatori aggregati:

- `count`: restituisce il numero di righe o valori distinti escludendo i valori nulli. Esempio:

  ```
  select count(*) from Ordine
  ```

- `<sum, max, min, avg>([distinct | all] Attr)`: esegue la somma, minimo, massimo e media sugli attributi. L'opzione `distinct` considera una volta ciascun valore, mentre l'opzione `all` considera tutti i valori diversi da `null`.

Nota bene: non si possono mischiare query di tupla con quelle aggregate!

```
-- Errore!
select Data, max(Improto)
from Ordine, Dettaglio
where Ordine.CodOrd = Dettaglio.CodOrd and CodProd = 'ABC'

-- Corretto!
select max(Importo) as MaxImp, min(Importo) as MinImp from Ordine
```

Query con raggruppamento

Nelle interrogazioni si possono applicare gli operatori aggregati a sottoinsiemi di righe. Si aggiungono le clausole `group by` (raggruppamento) e `having` (selezione dei gruppi).

```
-- Estrarre la somma degli importi degli ordini successivi al 10-6-97 per quei
-- clienti che hanno emesso almeno 2 ordini
select CodCli, sum(Importo)
from Ordine
where Data > 10-6-97
group by CodCli
having count(*) >= 2
```

Soltanto predicati che richiedono la valutazione di funzioni aggregate possono comparire nell'argomento della clausola `having`.

Si può combinare ordinamento con raggruppamento:

```
select ... from ... [where ...] [group by ... [having ...]] [order by ...]
```

Query binarie (set queries)

Le query binaria sono costruite concatenando due query SQL tramite operatori insiemistici:

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

Dove `union` indica l'unione, `intersect` l'intersezione e `except` (anche `minus`) la differenza. I duplicati vengono eliminati a meno che non venga specificato `all`.

Esempi Consideriamo lo schema:

```
Cliente(_NumC_, Nome, NumPat, ScadPat)
Noleggio(_NumN_, NumC, Targa, DataI, OraI, DataF, OraF)
Auto(_Targa_, Cambio, Posti, Tipo, Marca)
```

- Estrarre i clienti cui scade la patente durante il noleggio

  ```
  select Nome
  from Cliente C join Noleggio N on C.NumC = N.NumC
  where ScadPat between DataI and DataF
  ```

- Estrarre i clienti che hanno noleggiato auto Fiat oppure/e/e non VW

```
select C.NumC, Nome
from Cliente C, Noleggio N, Auto A
where C.NumC = N.NumC and N.Targa = A.Targa and Marca = 'Fiat'
union -- intersect/except
select C.NumC, Nome
from Cliente C, Noleggio N, Auto A
where C.NumC = N.NumC and N.Targa = A.Targa and Marca = 'VW'
```

- Estrarre i clienti che hanno noleggiato più di 5 volte nel 2013

```
select C.NumC, Nome
from Cliente C join Noleggio N on C.NumC = N.NumC
where (DataI >= 1.1.2013 and DataI <= 31.12.2013)
  or (DataF·>=·1.1.2013·and·DataF·<=·31.12.2013)
group by NumC
having count(*) > 5
```

- Estrarre i clienti che hanno noleggiato più di 5 volte auto dello stesso tipo nel 2013

```
select C.NumC, Nome
from Cliente C join Noleggio N on C.NumC = N.NumC
where (DataI >= 1.1.2013 and DataI <= 31.12.2013)
  or (DataF·>=·1.1.2013·and·DataF·<=·31.12.2013)
group by NumC, Tipo
having count(*) > 5
```

Query nidificate

Nella clausola where possono comparire predicati che confrontano un attributo con il risultato di una query SQL:

`AttrExpr Operator <any | all> SelectSQL`

- `any`: il predicato è vero se almeno una riga restituita dalla query nidificata soddisfa il controllo
- `all`: il predicato è vero se tutte le righe restituite dalla query soddisfano il controllo
- `Operator`: operatore di confronto

Altri tipi di query nidificate sono:

- `AttrExpr <in | not in> SelectSQL` con
  - `in`: il predicato è vero se almeno una riga restituita dalla query è presente nell'espressione
  - `not in`: il predicato è vero se nessuna riga restituita dalla query è presente nell'espressione
- `<exists | not exists> SelectSQL` con:
  - `exists`: il predicato è vero se la query possiede tuple
  - `not exists`: il predicato è vero se la query non possiede tuple

Le query possono essere nidificate anche su più livelli. Le query nidificate hanno accesso alle variabili definite nella query esterna.

Il potere espressivo di alcune espressioni nidificate è equivalente a quello di alcuni operatori SQL:

- `in`, `= any` e `exists` possono essere rese tramite `join` a meno di duplicati
- `not in`, `<> all` e `not exists` possono essere rese tramite al differenza
- `<comparatore> any` può essere reso con un theta-join, a meno di duplicati
- `<comparatore> all` può essere reso con raggruppamenti e estrazione di minimo e massimo

Le query nidificate possono anche essere utilizzate nelle modifiche:

```
update Ordine o
  set TotPrezzi == (select sum(Qta) from Dettaglio D where D.CodOrd = O.CodOrd)
```

Viste

Le viste offrono la "visione" di tabelle virtuali, simile al database intensionale del datalog. La sintassi per la creazione di una vista è:

```
create view Nome [(Attributi)] as SelectSQL
[with [local | cascaded] check option]
```

Possono a loro volta contenere nella definizione altre viste precedentemente definite, ma non vie può essere muta dipendenza. Le viste possono essere usate per formulare query complesse e sono talvolta necessarie per esprimere alcune query.

Le viste possono essere:

- semplici, ossia riconducibili a selezione e proiezione su una singola tabella
- complesse, ossia viste che usino qualsiasi cosa vada oltre al selezione e la proiezione

Non è possibile modificare le tabelle di base tramite le viste complesse poiché l'interpretazione sarebbe ambigua. Le opzioni `check option` intervengono in caso di aggiornamento per verificare che la tupla inserita/modificata appartenga alla vista:

- `local`: il controllo viene fatto solo rispetto alla vista su cui viene invocato il comando
- `cascaded`: il controllo viene fatto su tutte le viste coinvolte

Le viste ricorsive sono permesse da SQL99 in poi:

```
create view recursive Raggiungibile (Orig, Dest, Costo) as
( select Orig, Dest, Costo
  from Volo where Orig = 'Milano'
  union
  select R.Orig, V.Dest, V.Costo + R.Costo
  from Raggiungibile R join Volo V on R.Dest = V.Orig
  where V.Dest not in select Dest from Raggiungibile )
```

Controllo dell'accesso

Ogni componente dello scehma può essere protetto. Il proprietario di una risorsa (il creatore) assegna privilegi (autorizzazioni) agli altri utenti. Un utente predefinito (`_system`) rappresenta l'amministratore di sistema e ha pieno accesso a tutte le risorse. Un privilegio è caratterizzato da: risorsa, l'utente che concede il privilegio, l'utente che riceve il privilegio, l'azione che viene consentita e la possibilità di passare il privilegio ad altri utenti. SQL offre 6 tipi di privilegi:

1. `insert`: inserire un nuovo oggetto nella risorsa
2. `update`: modificare il contenuto della risorsa
3. `delete`: cancellare un nuovo oggetto nella risorsa
4. `select`: accedere al contenuto della risorsa in una query
5. `references`: per costruire un vincolo di integrità referenziale che coinvolge la risorsa
6. `usage`: per usare la risorsa in una definizione di schema

Per specificare tutti i privilegi si può usare `all privileges`. Per concedere un privilegio ad un utente si fa con:

```
grant <Privilegi | all privileges> on Risorsa to Utenti [with grant option]
```

Con `grant option` che specifica se deve essere garantita la possibilità di propagare il privilegio ad altri utenti. Per revocare un privilegio si usa:

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

Tramite le viste è possibile gestire in modo ottimale la privacy. Per dimostrare ciò facciamo un esempio basato su un caso reale: gestione dei conti correnti.

Abbiamo le seguenti due tabelle:

```
ContoCorrente(_NumConto_, Filiale, Cliente, CodFisc, DataApertura, Saldo)
Transazione(_NumConto_, _Data_, _Progr_, Cusale, Ammontare)
```

Abbiamo 2 utenti: i funzionari e i cassieri. I funzionari hanno tutti i privilegi sui conti correnti e `select` sulle transazioni. Il cassiere può aggiornare e selezionare i conti e ha il controllo completo sulle transazioni. Inoltre, i cassieri delle altre filiali della banca hanno solo privilegio di `select`

```
create view Conto1 as
( select *
  from ContoCorrente
  where Filiale = 1 )
with check option
create view Transazione1 as
( select *
  from Transazione
  where Filiale = 1 )
with check option

grant all privileges on Conto1 to Funzionari1
grant update(Saldo) on Conto1 to Cassieri1
grant select on Conto1 to Cassieri1, Cassieri2, Cassieri3
grant select on Transazione1 to Funzionari1
grant all privileges on Transazione1 to Cassieri1
grant select on Transazione1 to Cassieri2, Cassieri3
```

Comandi transazionali

Una unità elementare di esecuzione è incapsulata all'interno di due comandi: `begin transaction` e `end transaction`. Al suo interno, ogni possibile esecuzione prevede che sia eseguito esattamente una volta uno solo dei due comandi `commit work ...` o `rollback work ...`.

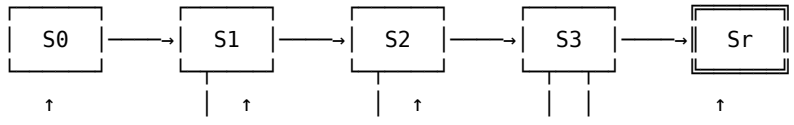
```
begin transaction;
update Account
  set Balance = Balance  10 where AccNum = 12202;
update Account
  set Balance = Balance  10 where AccNum = 42177;
select Balance into A from Account where Accnum = 42177
if (A >= 0) then
  commit work
else
  rollback work;
end transaction;
```

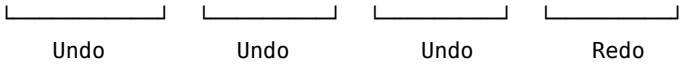
Una transazione ben formata ha la seguente forma:

```
begin transaction
SQL
commit work / rollback work
NO SQL
end transaction
```

La transazione è una operazione atomica che porta dallo stato iniziale a quello finale. Una transazione può assumere 3 comportamenti:

1. `commit work`: successo
2. `rollback work` o error precedente al commit: *undo*
3. errore dopo il commit: *redo*





Una transazione, inoltre, deve rispettare i vincoli di integrità e non è influenzata dal comportamento di altre transazioni concorrenti. L'effetto delle transazioni che hanno fatto `commit` non verrà mai perso.

Qualità dei dati

Un problema importante delle basi di dati è mantenere una alta qualità dei dati. In molte applicazioni reali i dati sono di scarsa qualità (tra il 5% e il 40% di dati scorretti). Per aumentare la qualità dei dati possiamo usare regole di integrità oppure modificare dati tramite programmi predefiniti (procedure e trigger).

Vincoli di integrità generici

I vincoli di integrità generici sono predicati che devono essere veri se valutati su istanze corrette (legali) della base di dati. Essi sono espressi in due modi: negli schemi delle tabelle o come asserzioni.

La clausola `check` può essere usata per esprimere vincoli arbitrari nella definizione dell schema. La sintassi è: `check(Condizione)`. Per `Condizione` si intende qualcosa che può essere messo in una clausola `where` (comprese el query nidificate.

```
create table Magazzino as
( CodProd   char(2) primary key,
  QtaDisp   integer not null
            check(QtaDisp > 0),
  QtaRiord  integer not null,
            check(QtaRiord > 10)
  check(QtaDisp > QtaRiord))
```

Le asserzioni permettono la definizione di vincoli al di fuori della definizione delle tabelle. Sono utili per esprimere vincoli inter-relazionali. Una asserzione associa un nome a una clausola check:

```
create assertion Nome check(Condizione)
```

La verifica dei vincoli può essere di due tipi:

1. `immediate`: la loro violazione annulla l'ultima modifica.
2. `deferred`: la loro violazione annulla l'intera transazione.

Ogni vincolo è definito di un tipo (normalmente `immediate`). L'applicazione può modificare a runtime il tipo dei vincoli tramite il comando `set constraint`

Procedure

Le procedure sono moduli di programma che svolgono una specifica attività di manipolazione dei dati. SQL-2 permette la definizione di procedure, ma solo in maniera molto limitata. La maggior parte dei sistemi offrono delle estensioni che permettono di scrivere procedure complesse (ad esempio Oracle PL/SQL) con strutture di controllo, variabili, eccezioni ecc.

Le procedure appaiono in due contesti: dichiarazione (DDL) e invocazione (DML). Nell'architettura client-server le procedure sono normalmente invocante dal client e memorizzate ed eseguite dal server.

```
procedure Prelievo (Prod integer, Quant integer) is
begin
  Q1, Q2 integer;
  X exception;

  select QtaDisp, QtaRiord into Q1, Q2
    from Magazzino
    where CodProd = Prod
  if Q1 < Quant then
    raise(X);
```

```
  update Magazzino
    set QtaDisp = QtaDisp - Quant
    where CodProd = Prof;
  if Q1 - Quant < Q2 then
    insert into riordino
      values(Prod, sysdate, Q2)
end;

-- Interfaccia
procedure Prelievo (Prod integer, Quant integer)

-- Invocazione
Prelievo(4, 150)
```

Le procedure permettono di modularizzare le applicazioni e aumentare l'efficienza, il controllo e il riuso. Le procedure però aumentalo la responsabilità dell'amministratore della base di dati in quanto si sposta conoscenza dalle applicazione allo schema della base di dati.

Trigger

Vengono dette basi di dati attive le basi di dati con componenti per la gestione di regole Evento-Condizione-Azione. Esse hanno un comportamento reattivo, in contrasto a quello passivo di default: esse non eseguono solo le transazione degli utenti ma anche le regole. Le regole sono simili alle procedure, ma l'invocazione è automatica. Nell'ambito dei DBMS commerciali si parla di trigger (standardizzati a partire da SQL-3).

Intuitivamente quando si verifica un evento se la una condizione è soddisfatta viene eseguita l'azione. I trigger sono definiti con istruzioni DDL (`create trigger`) e sono caratterizzati da un evento (`insert, delete, update`), una condizione specificata da un predicato SQL e un'azione, ossia una sequenza di istruzioni SQL (o estensioni com PL/SQL). Ogni trigger fa riferimento ad una tabella (`target`).

Esistono vari tipi di trigger:

1. `for each row`: scatta tupla per tupla, sono definite in automatico due variabili: `new` e `old` per rispettivamente la nuova e la vecchia tupla.
2. `for each statement`: scatta una volta per ogni comando sulla tabella; le variabili automatiche in questo caso daranno `new_table` e `old_table`.

```
create trigger GestioneRiordino
  after update of QtaDisp on Magazzino
  when (new.QtaDisp < new.QtaRiord)
  for each row
  X exception
  begin
    if new.QtaDisp < 0 then raise(X);
    insert into Riordino
      values(new.CodProd, sysdate, new.QtaRiord)
  end
```

I trigger godono di alcune importanti proprietà:

1. Terminazione: per ogni possibile stato iniziale e transazione, l'esecuzione termina
2. Confluenza: per ogni possibile stato iniziale e transazione, l'esecuzione termina nello stesso stato.
3. Identicità del comportamento: per ogni possible stato iniziale e transazione l'esecuzione termina nello stesso stato e con gli stessi *side-effect*.

I trigger possono risultare utili in molti ambiti. L'errata progettazione del trigger può portare ad effetti indesiderati e inoltre l'eccessiva proliferazione dei trigger rallenta il DBMS perché si devono controllare tutti i trigger che scattano sull'evento. Inoltre, poiché sono stati standardizzati così tardi (SQL-3), portano con sé limitazioni sulla portabilità tra vari DBMS.

Modellazione dei dati

Il modello *entity-relationship*

Il modello *entity-relationship* è stato introdotto da P. Chen e si è affermato come standard industriale di buona parte delle metodologie e degli strumenti per il progetto concettuale di basi di dati. Sono definite:

1. L'entità è una classe di oggetti o di fatti rilevanti per l'applicazione. Ogni entità è caratterizzata da un nome.
2. Le relazioni (o associazione) rappresenta una aggregazione di entità di interesse di interesse per l'applicazione. Ogni istanza di una associazione è una ennupla tra istanze di entità. Anche ogni relazione è caratterizzata da un nome. Le associazioni possono anche essere ternarie: 3 entità possono essere collegate da una sola associazione.
3. Gli attributi rappresentano caratteristiche delle entità e delle associazioni di interesse per l'applicazione. Ogni attributo è caratterizzato da un nome

Una linea guida è la seguente: se il concetto è significativo per il contesto è un'entità; se il concetto è descrivibile tramite un dato elementare è un attributo; se il concetto definisce un legame tra entità è un'associazione.

Per cardinalità si intende un vincolo sul numero di istanze di associazione cui ciascuna istanza di entità deve partecipare. Possiamo avere 3 tipi di associazioni in base alla cardinalità massima: uno-uno, uno-molti, molti-molti. Possiamo anche costruire delle auto-associazioni.

Gli attributi possono avere valori scalari o multipli. Un attributo può essere anche composto. Anche gli attributi composti possono essere scalari o vettoriali. Un attributo che identifica in modo univoco ciascuna singola istanza di entità si dice identificatore. Anche un identificatore può essere composto.

Le entità possono essere deboli, ossia l'esistenza delle loro istanze è dipendente dall'esistenza di istanze di un'altra entità forte.

Una gerarchia di generalizzazione è un legame tra un'entità padre ed alcune entità figlie. Le entità figlie ereditalo le proprietà (attributi, relazioni, identificatori) dell'entità padre. Una gerarchia può essere:

- Totale se ogni istanza dell'entità padre fa parte di una delle entità figlie.
- Parziale se le istanze dell'entità padre possono far parte di una delle entità figlie.
- Esclusiva se ogni istanza dell'entità padre non può far parte di più di una delle entità figlie.
- Overlapping se ogni istanza dell'entità padre può far parte di più entità figlie.

Il progetto di una base di dati

Il progetto di una base di dati si inserisce nel ciclo di vita del sistema informativo. Noi non ci concentriamo su tutte le fasi del progetto, ma solo sulla progettazione degli schemi. La progettazione degli schemi a sua volta si suddivide in 3 fasi:

1. La progettazione concettuale ha per scopo tradurre l'analisi dei requisiti in una descrizione formale indipendente dal DBMS. La descrizione formale è espressa tramite uno schema concettuale costruito utilizzando un modello concettuale dei dati (ad esempio *entity relationship*).
2. La progettazione logica ha per scopo tradurre lo schema concettuale in uno schema logico, scelto all'interno dei modelli logici dei dati (gerarchico, reticolare, relazionale, object-oriented o XML). Lo schema logico è dipendente dal DBMS, ma non dallo specifico prodotto utilizzato.
3. La progettazione fisica ha per scopo produrre un progetto fisico della base di dati, cioè un progetto che ottenga prestazioni ottimali tramite scelta e dimensionamento di strutture fisiche di accesso. Il progetto fisico viene eseguito in modo differente su ciascun prodotto.

Il progetto concettuale

Le strategie di progetto possono essere "top-down" o "bottom-up". Queste ultime a loro volta possono essere a "macchia d'olio" o "miste".

Nel progetto top-down si procede per raffinamenti a partire da una descrizione che comprende tutta la realtà d'interesse. Ciò rende il progetto più ordinato e razionale, ma la sua progettazione è più difficile poiché richiede visione d'insieme. Nella strategia top-down "pura" si costruisce uno schema iniziale e partire dalle specifiche. Da questo schema iniziale si arriva per raffinamenti successivi allo schema finale. I raffinamenti prevedono l'uso di trasformazioni elementari (primitive) che operano sul singolo concetto per descriverlo con maggio dettaglio.

Nel progetto bottom-up si disegnano separatamente aspetti della realtà e poi li si integra costruendo un unico schema. Ciò permette di prendere decisioni differenti nell'affrontare i sotto-problem. Queste decisioni diverse potranno portare a

conflitti.

- La strategia a macchia d'olio nasce per accomodare i cambiamenti dei requisiti. Le tecnica è adatta a tradurre pian piano una descrizione testuale in un diagramma. Pur essendo bottom-up, il progettista analizza le specifiche in modo "stratificato" e le aggiunge progressivamente ad un unico schema, perciò i conflitti sono meno probabili.
- La strategia mista è la più adatta di fronte a progetti ampi. Si suddividono le specifiche in parti e si realizzano in top-down le varie parti per poi integrarle con un approccio bottom-up. Questo "divide-and-conquer" permette a più progettisti di lavorare sullo stesso schema. Il problema, come nell'approccio top-down, sono i conflitti tra i vari sotto schemi. La soluzione è lo sviluppo di un piccolo schema scheletro di soli concetti principali in modo top-down e attenersi alle scelte presenti nello schema scheletro in tutti gli altri schemi.

La qualità di uno schema concettuale dipendono dalla sua completezza, correttezza, leggibilità, minimalità e auto-esplicatività:

- Completezza e correttezza significa, ovviamente, rappresentare il modo completo e corretto i requisiti. Per migliorare la completezza bisogna, ovviamente, assicurarsi che i dati consentano di eseguire tutte le applicazioni. Per la correttezza bisogna assicurarsi che sia possibile popolare la base di dati anche con informazioni incomplete durante le fasi iniziali.
- La leggibilità si divide in leggibilità concettuale e grafica. Un modo per aumentare la leggibilità è eliminare eventuali ridondanze rimuovendo relazioni superflue senza perdere informazione. Nota: non sempre le ridondanze sono inutili, alcune volte l'eliminazione di una ridondanza porta al'uso di join complesse e meno immediatezza dello schema!
- Auto-esplicatività significa fare in modo che lo schema rappresenti esplicitamente il massimo di conoscenza sulla realtà.

L'ultimo passo da svolgere dopo aver progettato lo schema concettuale ed essersi assicurato della sua qualità è il post-processing. Per post-processing si intende un insieme di "pulizia" dello schema:

1. Verificare che tutte le entità abbiano un identificatore;
2. Verificare che tutte le associazioni abbiano cardinalità ben definite;
3. Verificare che le entità siano significative;
4. Verificare che tutte le generalizzazioni siano utili.

Il progetto logico

Lo schema entità-relazione descrive un dominio applicativo ad un dato livello di astrazione. Lo schema entità-relazione fornisce una buona descrizione sintetica e visiva ed è utile per rappresentare buona parte della semantica e mantenere una buona documentazione. Nel progetto logico, dobbiamo tradurre un modello E/R in un insieme di tabelle con `primary key` e `foreign key`. Per fare ciò possiamo eseguire le seguenti operazioni:

1. **Eliminazione delle gerarchie**: possiamo seguire due approcci:
   - Orizzontale: la gerarchia collassa in un'unica entità in corrispondenza del padre (esempio: Persona - Maschio/Femmina)
   - Verticale: si incorpora l'antenato comune nei figli (esempio: Persona - Stendete/Professore)
2. **Selezione delle chiavi primarie**: si sceglie come chiave primaria l'identificatore più usato, purché semplice. Se la chiave primaria sarebbe troppo complessa si introduce un codice. Gli identificatori esterni vanno "importati".
3. **Normalizzazione di attributi composti/multipli**: gli attributi composti vanno resi semplici. Quelli multipli vanno gestiti con una entità separata.
4. **Eliminazione di relazioni**: tradurre relazioni m-n, 1-n e 1-1 nei relativi equivalenti utilizzabili nei DB.

Un altro approccio alla progettazione: la normalizzazione

Quest'altro approccio è basato sui concetti di dipendenza funzionale e di forma normale. Parte dall'osservazione delle cosiddette anomalie, causate dalla ridondanza, senza passare per lo schema ER.

Abbiamo visto nelle prime sezioni i vincoli di chiave, di integrità o di tupla. Essi fanno parte di una categoria più vasta di vincoli: le dipendenze funzionali.

Definizione. ***Dipendenza funzionale*** Dati due insiemi di attributi X e Y, si dice he X determina Y, o che Y dipende da X e si scrive $X \to Y$ se e solo se date due tuple distinte t_1 e t_2 se $t_1[X] = t_2[X]$ allora $t_1[Y] = t_2[Y]$.

Definiamo, quindi, 3 tipi di anomalie. Consideriamo un'unica tabella con schema `(_Employee_, Salary, _Project_, Budget, Function)`

1. Se lo stipendio di un dipendente cambia, dobbiamo modificare il valore di tutte le corrispondenti tuple. Questa è detta anomalia di aggiornamento.
2. Se un dipendente smette di lavorare su tutti i progetti ma non lascia la società, tutte le corrispondenti tuple sono eliminate e così anche le informazioni di base. Questo è nota come anomalia di cancellazione
3. Se abbiamo informazioni su un nuovo dipendente, non le possiamo inserire fino a che il dipendente sia assegnato ad un progetto, a meno di decidere di inserire valori nulli. Ma questo non è possibile poiché `Project` è parte della chiave. Questo è noto come anomalia di inserimento.

Osservando la tabella sopra possiamo osservare che la la chiave primaria determina tutti gli altri attributi, ma valgono anche $employee \rightarrow salary$ e $project \rightarrow budget$.

Definizione. ***Dipendenza funzionale non banale*** Diciamo che una dipendenza funzionale $Y \rightarrow Z$ non è banale se nessun attributo in Z compare come attributo in Y.

Nell'esempio sopra $employee \rightarrow salary$ e $employee, project \rightarrow project$ non sono banali. In generale, una dipendenza funzionale causa anomalie se la relazione contiene informazioni eterogenee, ossia alcune informazioni corrispondenti alla chiave e altre ad attributi che non formano una chiave. La relazione si può decomporre in modo da ottenere tabelle contenti informazioni omogenee ottenendo 3 tabelle: `(Employee, Salary)`, `(Project, Budget)` e `(Employee, Project, Function)`.

Definizione. ***Forma normale di Boyce e Codd*** Una relazione R è in forma normale di Boyce e Codd (BCNF) se per ogni dipendenza $X \rightarrow Y$ in R, X contiene la chiave K di R.

È possibile che una relazione non in forma normale venga decomposta in due o più relazioni in forma normale. La decomposizione si può attuare effettuando proiezioni in modo tale da ottenere che ciascuna dipendenza funzionale definisca la chiave di una relazione separata.

Quando la relazione originale è ricostruibile mediante `join` la decomposizione è corretta e si dice senza perdita. La decomposizione senza perdita è garantita se gli attributi comuni contengono una chiave per almeno una delle relazioni decomposte. Le decomposizioni possono essere con perdita se non si conservano le dipendenze. Una decomposizione conserva le dipendenze se ogni dipendenza funzionale dello schema originale coinvolge attributi che compaiono attributi che compaiono assieme negli schemi decomposti.

Possono esistere dei casi in cui la BCNF non sia raggiungibile. Definiamo quindi una nuova forma normale, più debole della BCNF.

Definizione. ***Terza forma normale*** Una relazione R è in terza forma normale se, per ogni dipendenza funzionale non banale $X \rightarrow Y$ definita su di essa almeno una delle seguenti condizioni è verificata: X contiene una chiave di R; Y è contenuto in almeno una chiave di R.

La terza forma normale è sempre ottenibile.

Esistono anche le la seconda e la prima forma normale:

Definizione. ***Prima forma normale*** Una relazione è in prima forma normale se e solo se ciascun attributo è definito su un dominio con valori atomici.

Definizione. ***Seconda forma normale*** Una relazione è in seconda forma normale se è in prima forma normale e tutti gli attributi non chiave dipendono funzionalmente dall'intera chiave. Questa normalizzazione elimina le dipendenze parziali.

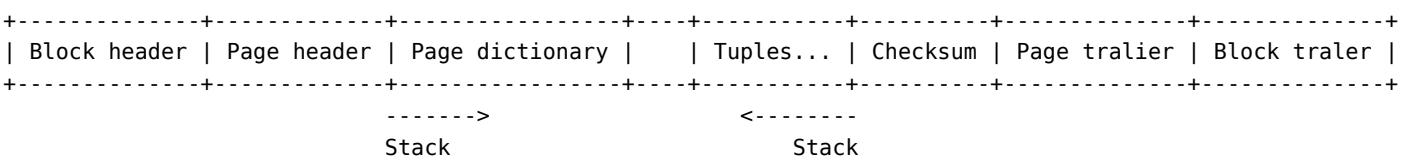
Definizione. ***Terza forma normale (definizione alternativa)*** Una relazione è in terza forma normale se è in seconda forma normale e inoltre non contiene dipendenze transitive dalla chiava, ciò tutti gli attributi non chiave dipendono direttamente dalla chiave.

La teoria della normalizzazione può essere usata per controllare la qualità degli schemi sia durante la progettazione concettuale che logica. L'analisi delle relazioni ottenute durante la progettazione logica può identificare problemi di inaccuratezza durante la progettazione concettuale. Le idee su cu si basa la normalizzazione possono essere utilizzate durante la progettazione concettuale per controllare la qualità degli elementi dello schema concettuale.

Il progetto fisico

Le basi di dati vengono memorizzate in file in memoria secondaria per motivi di dimensioni e persistenza. La pagina è l'unità di memorizzazione utilizzata per la memoria primaria, mentre il blocco per la secondaria.

La struttura di un blocco dipende dal database utilizzato, ma generalmente ha la seguente struttura



Le tuple di una stessa tabella saranno divise tra diversi blocchi. Le tuple vengono organizzate in modo sequenziale, solitamente ordinate in base alla data di inserimento.

Gli indici sono una struttura ausiliaria per permettere l'accesso efficiente alle tuple di un database. Ha come input i valori di un attributo, o di una lista di attributi, detti *chiave dell'indice*. Solitamente le strutture per gli indici sono basate su tabelle di hash o su alberi B+. Per creare e distruggere indici si possono usare i seguenti comandi:

```
create [unique] index IndexName on TableName (AtrtibuteList)
```

```
drop index IndexName
```

Delle linee guida per la creazione degli indici sono:

- Non cerare indici se la tabella è piccola
- Monti DBMS creano in modo automatico degli indici per le chiavi primarie e per le chiavi `unique`
- Può essere necessario aggiungere indici su attributi che compaiono in predicati di selezione o `join` oppure su operazioni che richiedono l'ordinamento
- Evitare di aggiungere indici se la tabella viene aggiornata frequentemente
- Evitare di aggiungere indici se in ogni caso per l'interrogazione occorre estrarre una parte significativa delle tuple della tabella
- Evitare di indicizzare attributi costituiti da lunghe stringhe di caratteri