Basi di Dati

Luca Seggiani

29 maggio 2024

1 Introduzione alla base di dati relazionale

Una base di dati (in inglese "database") è una mole di dati organizzata in modo da favorirne la consultazione attraverso determinate interrogazioni ("query"). Formalmente, una base di datiè un archivio di dati contenente informazioni ben strutturate organizzate secondo un modello logico (nel caso di MySQL un modello relazionale).

L'elemento fondamentale di una base di dati fondata sul modello relazionale è la **Tabella**, formata da colonne (associate a determinati attributi), e righe (che rappresentano record dei suddetti attributi). Una colonna particolare, che deve differire per ogni record della tabella, viene detta chiave primaria e serve a distinguere tra di loro record (che potrebbero essere altrimente stati tra di loro identici!).

L'interrogazione della base di dati si effettua attraverso le cosiddette query, ovvero richieste compilate in linguaggio SQL (Structured Query Language), un linguaggio dichiarativo sviluppato appositamente per questo scopo. La query più semplice si basa su tre istruzioni: SELECT, FROM e WHERE. Prendiamo l'esempio in lingua naturale di un'interrogazione a una base di dati contenente dati su diverse persone (nello specifico, nome, cognome, età e codice fiscale come chiave.):

Riportare nomi e cognomi di tutte le persone con eta maggiore di 40 anni.

in questo caso gli attributi di interesse sono i nomi e cognomi (in quest'ordine) delle persone valide, e la condizione è un età maggiore di 40 anni. Di fronte a questa richiesta, la base di dati risponderà fornendo un insieme risultato (result set) contenente una tabella formata esattamente dalle informazioni richieste. In SQL potremmo definire tale query come segue:

```
SELECT Cognome, Nome
FROM Persona
WHERE Eta > 40
```

in questo caso, cognome e nome sono gli attributi di interesse che verrano inseriti nel nostro insieme risultato (attraverso un processo chiamato proiezione). La tabella persona sarà la fonte dei nostri dati, e età > 40 la nostra condizione.

Le condizioni in SQL supportano anche i classici operatori logici (AND, OR, ecc...). Un operatore degno di nota, applicabile ai soli dato di tipo numerico o su cui è comunque stabilita una relazione d'ordine, è il BETWEEN, che permette di selezionare (estremi inclusi) tutti i valori in un dato range.

Duplicati
2 righe risultano uguali se tutti gli attributi del record hanno valori identici. Non si possono avere righe uguali in tabelle SQL (la chiave distingue), ma è possibile avere duplicati tra gli attributi non chiave, con quindi risultanti problemi dati dalla proiezione di tali attributi su insiemi risultato privi di attributo chiave. Nel caso si rendesse necessario eliminare i duplicati da una data query, l'SQL offre il costrutto SELECT DISTINCT. Si nota che il costo per l'eliminazione dei duplicaty è computazionalmente $O(n^2)$, e quindi da evitare quando possibile.

Valori NULL
I valori NULL corrispondono solitamente a informazioni mancanti, non disponibili o comunque sconosciute. Qualsiasi condizione che coinvolge valori NULL è sempre falsa, compreso il confronto fra stessi NULL. È possibile utilizzare i valori NULL assegnandoli un valore semanticamente definito (e.g. una data non definita perchè non ancora determinata, ecc...). In questo caso, l'SQL offre i costrutti IS NOT NULL e IS NULL, che restituiscono rispettivamente falso e vero posti accanto ad un valore NULL. Si nota inoltre che la chiave di un record non può essere NULL.

Gestione delle date in SQL
Come per tutti i sistemi UNIX, le date in SQL vengono rappresentate come una distanza in secondi dalla mezzanotte del primo gennaio 1970 (il cosiddetto tempo di riferimento EPOCH, *EPOCH reference*).

Formato di data Esistono 2 tipi di dato data in SQL: DATE (formato YYYY-MM-DD) e TIMESTAMP (formato YYYY-MM-DD HH:MM:SS). La manipolazione di date è possibile attraverso la funzione DATE_FORMAT, che consente di cambiare il formato o effettuare manipolazioni d'utilità. Ad esempio potrò avere:

```
SELECT Matricola, DATE_FORMAT(DataLaurea, '%d|%m|%Y, %w')
FROM Studente
WHERE DataLaurea > 12-7-2004
```

che prende tutti gli studenti laureati dopo il 12 luglio 2004 e ne riporta il nome di matricola e la data formattata come DD-MM-YYYY + settimana. ancora, per includere ad esempio gli studenti laureati di mercoledì:

```
SELECT Matricola
FROM Studente
WHERE DATE_FORMAT(DataLaurea, '%w') = 3
```

dove il '%w' restituisce il giorno della settimana di DataLaurea. Qualsiasi ulteriore informazione sui formati delle date può essere trovata al link:
`https://www.w3schools.com/sql/func_mysql_date_format.asp`

Poniamoci adesso il problema di restituire i cognomi di tutti gli studenti laureati 5 anni fa, ovvero con un offset di 5 anni rispetto alla data attuale. Servirà chiaramente un qualche riferimento alla data di oggi, che l'SQL fornisce con la parola chiave CURRENT_DATE. Potrò quindi avere:

```
SELECT DISTINCT Cognome
FROM Studente
WHERE DataLaurea IS NOT NULL
  AND YEAR(DataLaurea) = YEAR(CURRENT_DATE) - 5
```

si noti che abbiamo usato l'operatore "-": l'SQL fornisce anche la funzione DATEDIFF(dataRecente, dataRemota), che restituisce il numero di giorni che separano due date. Nota bene: somme e sottrazioni di date non hanno senso: tutto quello che possiamo fare è sommare i singoli valori (anno, giorno, ecc...) di date diverse. Per qualsiasi altro caso occorre usare DATEDIFF(). Possiamo avere ad esempio:

```
SELECT Matricola, DATEDIFF('2005-07-15', DataIscrizione)
FROM Studente
WHERE DataIscrizione < '2005-07-15'
  AND DataLaurea > '2005-07-15'
```

che restituirà la matricola e giorni da quando si erano iscritti degli studenti ad oggi laureati e che non si erano ancora laureati il 15 Luglio 2005.
Inoltre, per sommare e sottrarre lassi di tempo a date possiamo usare le funzioni DATE_ADD e DATE_SUB. I suddetti lassi andranno espressi con la parola chiave INTERVAL:

```
INTERVAL NumeroIntero [YEAR|MONTH|DAY]
```

riportiamo ad esempio matricola e mese di iscrizione degli studenti che si sono laureati dopo cinque anni esatti dal giorno dell'iscrizione:

```
SELECT Matricola, MONTH(DataIscrizione)
FROM Studente
WHERE DataLaurea = DATE_ADD(DataIscrizione, INTERVAL 5 YEAR)
```

quando la somma è definita fra date e intervalli, possiamo usare anche solo l'operatore "+":

```
  AND DataLaurea = DataIscrizione + INTERVAL 5 YEAR
```

esistono poi diverse altre funzioni di utilità sulle date, che possono essere trovate nel sito precedentemente citato.

Operatori di aggregazione

Gli operatori di aggregazione permettono di fare determinati calcoli i cui operandi sono i valori assunti da un attributo in un insieme di record (ovvero conteggio, somma, minimo / massimo, media), e di collassarli in un unico attributo numerico. Gli operatori di aggregazione sono disponibili solamente nel SELECT, in quanto il WHERE non ha alcuna visibilità a livello globale della tabella su cui lavora, ma solo a livello di record.

Conteggio

Iniziamo col contare il numero di righe di una tabella o di un suo sottoinsieme. Definita la tabella di una semplice realtà medica, abbiamo:

```
SELECT COUNT(*) AS VisitePrimoMarzo
FROM Visita
WHERE Data = '2013-03-01'
```

per trovare il numero di visite effettuate in data 1 marzo 2013. La parola chiave AS serve solamente per rinominare la tabella generata dalla funzione COUNT(). Possiamo effettuare anche, anzichè il conteggio delle righe, il conteggio dei valori diversi assunti da un attributo. Ad esempio, se cercassimo il numero di pazienti visitati nel mese di marzo 2013, visto che lo stesso paziente potrà essere stato visitato più volte in un solo mese, dovremmo fare:

```
SELECT COUNT(DISTINCT Paziente) as PazientiMarzo
FROM Visita
WHERE MONTH(Data) = '03'
  AND YEAR(Data) = '2013'
```

Somma

Posso sommare i valori numerici di degli attributi di più record usando SUM(). Supponiamo di voler calcolare, data una tabella del reddito di più persone, il reddito totale di una sola famiglia:

```
SELECT SUM(Reddito) AS RedditoTotale
FROM Paziente
WHERE Cognome = 'Lepre'
```

Media

Allo stesso modo, potrò calcora il reddito medio:

```sql
SELECT AVG(Reddito) AS RedditoMedio
FROM Paziente
WHERE Cognome = 'Lepre'
```

Minimo / Massimo

Potrò inoltre calcolare valori massimi e minimi:

```sql
SELECT MIN(Reddito)
FROM Paziente

SELECT MAX(Reddito)
FROM Paziente
```

A questo punto, se volessimo cercare il reddito massimo, e il nome e cognome di chi lo detiene, incontreremmo un'ostacolo: non si può infatti ottenere un qualsiasi altro attributo da un insieme risultato ormai collassato ad un solo valore numerico. Non otterremmo nulla dal codice:

```sql
SELECT MAX(Reddito), Nome, Cognome
FROM Paziente
```

Riassumendo: non si possono affiancare agli operatori di aggregazione i nomi di attributi ormai collassati.

Query su più tabelle

In un database le informazioni sono spesso frammentate su più tabelle. Questo aiuta a evitare ridondanze, anomalie, ed a vere la possibilità di distribuire i dati. Ad esempio, immaginiamo il database di una clinica medica, che dovrà quindi immagazinare i dati di pazienti, dottori, visite mediche ecc... Potremo allora definire più tabelle separate per ciascuna di queste categorie di dati, ognuna con i propri specifici attributi. A questo punto, ogni tabella potrà contenere come attributi altre tabelle, o meglio record provienienti da altre tabelle in qualche modo "collassati" nel singolo attributo della tabella. Questo meccanismo si concrettizza immagazzinando nella tabella l'informazione minimale per ritrovare il record desiderato nella tabella d'appartenenza, ovvero riportandone soltanto la chiave (che sappiamo essere diversa per ogni record archviato). La successiva esplosione della chiave fino al record completo nella tabella d'appartenenza viene effettutato in SQL attraverso le operazioni di unione (join).

Inner join

L'inner join trova il record nella tabella d'appartenenza che corrisponde alla chiave nella tabella su cui lavoriamo, e semplicemente lo affianca. Ad esempio, magari vogliamo indicare nome e cognome dei medici che hanno effettuato, nella nostra clinica, almeno una visita. Dovremmmo allora prendere la nostra tabella visita, che conterrà la chiave di un medico nella tabella medico, che

andremo quindi a sostituire con il record completo del medico corrispondente. In codice:

```
SELECT DISTINCT M.Nome, M.Cognome
FROM Visita V
  INNER JOIN
  Medico M ON V.MEDICO = M.Matricola
```

Riprendiamo la trattazione delle operazioni che permettono l'unione (join) di più tabelle.

Natural join

Il join naturale combina i record della prima tabella con i record della seconda tabella aventi valori uguali su tutti gli attributi omonimi. Ad esempio, in pseudocodice, ammettendo che la tabella visita abbia un attributo ominimo ad un'altro attributo sulla tabella medico:

```
  SELECT M.Nome, M.Cognome
  FROM Visita V
    NATURAL JOIN
    Medico M
```

Nota bene: **tutti** gli attributi omonimi dovranno avere valori identici, ed a quel punto il record della tabella di provenienza verrà unito alla tabella su cui lavoriamo una sola volta.

External join

Il join esterno combina tutti i record della prima tabella con tutti i record della seconda che soddisfano una condizione, mantenendo tutti i record in una delle tabelle, a differenza del join esterno che invece scarterebbe i record non appaiati con nessuno dei record della tabella madre. Il join esterno può essere pensato come una sorta di prodotto cartesiano fra tabelle. I record spaiati che verrano uniti avranno tutti valori NULL sugli attributi della tabella di provenienza (inesistenti). Poniamo ad esempio di voler indicare le visite effettuate da medici che non lavorano più presso la clinica. Supponiamo quindi che loro anagrafica sia stata quindi eliminata dal database.

```
SELECT V.*
FROM Visita V
  LEFT OUTER JOIN
  Medico M ON V.Medico = M.Matricola
WHERE M.Matricola IS NULL;
```

Così facendo potremo unire tutti i record di tutti i medici, anche quelli che non lavorano più nella mia clinica. Controllando a questo punto chi di questi medici ha elemento chiave (si controlla, per convenzione, proprio l'elemento chiave) NULL, otterremo le visite svolte dai suddetti. Notiamo infine la differenza fra join esterno destro e join esterno sinistro: il join esterno sinistro mantiene tutti i record della tabella di sinistra mettendo NULL a destra

per i record non appaiati. Il join esterno destro fa la cosa opposta, quindi mantenendo i record a destra e inserendo NULL a sinistra.

Query con join e condizioni sui record

Possiamo a questo punto combinare il FROM con instruzione di join allo WHERE con condizioni di selezione dei record d'interesse applicate alla tabella *dopo* il join. Ad esempio, se voglio ottenere matricola, cognome e specializzazione dei medici che hanno visitato almeno un paziente il giorno 1 Marzo 2013

```
SELECT DISTINCT M.Matricola, M.Cognome, M.Specializzazione
FROM Medico M
  INNER JOIN
  Visita V ON M.Matricola = V.Medico
WHERE V.Data = '2013-01-03'
```

Unioni multiple

Il join multiplo può essere effettuato su più di 2 tabelle. Diciamo che oltre al medico, la nostra tabella visita contiene un'attributo corrispondente al paziente visitato. Vogliamo allora ottenere nomi e cognomi di tutti i pazienti visitati da un certo medico. Notiamo che il nome del medico sta nella sottotabella corrispondente al medico, e il nome del paziente sta nella sottotabella corrispondente al paziente, ovvero dovremmo unire non una ma due tabelle alla nostra tabella madre. In codice:

```
SELECT DISTINCT P.Nome, P.Cognome
FROM Paziente P
  INNER JOIN
  Visita V ON P.CodFiscale = V.Paziente
  INNER JOIN
  Medico M ON V.Medico = M.Matricola
WHERE M.Nome = 'Rino'
  AND M.Cognome = 'Neri';
```

notiamo che in questo caso, per assicurare la sicurezza dell'operazione, nome e cognome del medico dovrebbero essere una cosiddetta *chiave candidata*, ovvero un insieme di attributi sempre diversi in tutti i record (come la chiave primaria ma senza lo status di chiave primaria).

Si noti inoltre di come gli alias (P V e M) nell'esempio precedente hanno il compito di evitare ambiguità su attributi con lo stesso nome ma appartenenti a tabelle diverse. Ulteriore esempio, supponiamo di voler indicare nome e cognome dei pazienti visitati nel mese di dicembre 2013, e il nome e cognome dei medici che li hanno visitati:

```
SELECT DISTINCT P.Nome AS NomePaziente, P.Cognome AS
    CognomePaziente
                M.Nome AS NomeMedico, M.Cognome AS
    CognomeMedico
```

```
FROM Paziente P
  INNER JOIN
  Visita V ON P.CodFiscale = V.Paziente
  INNER JOIN
  Medico M ON V.Medico = M.Matricola
WHERE YEAR(V.Data) = 2013
  AND MONTH(V.Data) = 12
```

Self join

Il self join combina le righe di una tabella con righe della stessa tabella. Più propriamente, non esiste una keyword SELF JOIN, ma possiamo sfruttare le possibilità dell'INNER JOIN. Diciamo ad esempio di volere il codice fiscale dei pazienti che sono stati visitati più di una volta da uno stesso medico della clinica, nel mese corrente:

```
SELECT DISTINCT V1.Paziente
FROM Visita V1
  INNER JOIN
  Visita v2 ON (
                V2.Medico = V1.Medico
                AND V2.Paziente = V1.Paziente
                AND V2.Data<> V1.Data
                )
WHERE MONTH(V1.Data) = MONTH(CURRENT_DATE)
  AND YEAR(V1.Data) = MONTH(CURRENT_DATE)
  AND MONTH(V2.Data) = MONTH(CURRENT_DATE)
  AND YEAR(V2.Date) = YEAR(CURRENT_DATE);
```

qui unisco alla mia tabella visita un record proveniente dalla stessa tabella quando medico, paziente corrispondono e la data differisce. A questo punto seleziono soltanto i record della nuova tabella creata che hanno date diverse fra le due visite. Si noti che per n visite, il paziente comparirà nel result set n - 1 volte (da cui il DISTINCT), ovvero il join accade in una sola direzione.

Common Table Expression

Le common table expression (CTE) sono result set dotati di identificatori che possono essere usati prima di una query per costruire risultati intermedi. Si scrivono, separate da virgole, prima della query che li usa, tramite la parola chiave WITH e separati da virgole. Sono parti di codice i cui risultati vengono stoccati e poi restituiti alle query che nli usano. Ad esempio:

```
WITH
  name1 AS (query1)
  name2 AS (query2)
  ...
  nameN AS (queryN)
query finale;
```

Diciamo per esempio, di volere indicare il numero di pazienti di Siena mai visitati da ortopedici. Avrò allora la CTE che trova tutti gli ortopedici:

```
WITH ortopedici AS
  (
  SELECT M.Matricola AS Medico
  FROM Medico M
  WHERE M.Specializzazione = 'Ortopedia'
  )
```

poi la CTE che trova tutti i pazienti visitati da tali ortpedici:

```
paz_visitati_ortopedici AS
  (
  SELECT V.Paziente AS CodFiscale
  FROM Visita V NATURAL JOIN Ortopedici O
  )
```

infine ottengo, usando le CTE precedenti, tutti i pazienti senesi:

```
SELECT COUNT(*)
FROM Paziente P
  NATURAL LEFT OUTER JOIN
  paz_visitati_ortopedici PVO
WHERE Citta = 'Siena'
  AND PVO.Matricola IS NULL
```

2 Subquery

Le subquery (query annidate) permettono di incapsulare query in altre query, in modo correlato o non correlato. Presentano un modo alternativo di risolvere i problemi su cui normalmente si userebbe il join: infatti per ogni problema che si può risolvere usando il join esiste sempre un corrispettivo che usa le subquery. Chiamiamo outer query la query più esterna, cioè quella che incapsula, e inner query la subquery incapsulata.

Subquery non correlate

Le subquery non correlate (*noncorrelated*) sono incapsulate nel WHERE, e permettono di ottenere un certo result set intermedio, che verrà poi usato dall'outer query per calcolare il result set finale. Il result set della subquery non correlata non dipende dall'outer query. Poniamo ad esempio di voler indicare nome, cognome e parcella degli ortopedici che hanno effettuato almeno una visita nell'anno 2013. Definiremo allora la subquery:

```
SELECT V.Medico
FROM Visita V
WHERE YEAR(V.Data) = 2013
```

che innesteremo poi in:

```
SELECT M.Nome, M.Cognome, M.Parcella
FROM Medico M
WHERE M.Specializzazione = 'Ortopedia'
  AND M.Matricola IN  (
                      SELECT V.Medico
                      FROM VIsita V
                      WHERE YEAR(V.Data) = 2013
                      );
```

La parola chiave IN permette di controllare la presenza dell'attributo M.Matricola nel result set della subquery. La subquery calcolerà quindi un result set composto da tutte le visite fatte nel 2013. L'outer query userà la subquery, confrontandola con tutti i medici specializzati in ortopedia la cui matricola trova un riscontro nel result set calcolato dalla subquery. A questo punto si restituiscono gli attributi richiesti. Il meccanismo è del tutto analogo a quello di un join naturale fra le tabelle "Medico" e "Visita". Notiamo che non è necessario alcuna istruzione DISTINCT, in quanto ogni record della tabella Medico viene considerato comunque una volta sola, e cercato nel result set delle visite.
Avremmo potuto porre la stessa query usando un comune join, ad esempio come:

```
SELECT M.Nome, M.Cognome, M.Parcella
FROM Medico M
  INNER JOIN Visita V ON V.Medico = M.Matricola
WHERE YEAR(V.Data) = 2013
  AND Medico.Specializzazione = 'Ortopedico';
```

Si può anche usare la negazione di quanto appena fatto, usando la parola chiave NOT IN. Ad esempio, se vogliamo indicare i cognomi dei pazienti che non appartengono anche a un medico, abbiamo la subquery:

```
SELECT M.Cognome
FROM Medico M
```

innestata in:

```
SELECT DISTINCT P.Cognome
FROM Paziente P
WHERE P.Cognome NOT IN  (
                        SELECT M.Cognome
                        FROM Medico M
                        );
```

Analogamente a prima, esisterà una versione che usa il join:

```
SELECT DISTINCT P.Cognome
FROM Paziente P
```

```
  LEFT OUTER JOIN Medico M ON
    M.Cognome = P.Cognome
WHERE M.Cognome IS NULL;
```

Annidamento multiplo

Non c'è teoricamente limite al numero di subquery che si possono innestare l'una dentro l'altra. A volte può infatti tornare utile avere innesti multipli (e frontali!). Vediamo come ottenere il numero di tutti i pazienti di Siena mai visitati da pazienti. Innanzitutto si trovano i medici specializzati in ortopedia:

```
SELECT M.Matricola
FROM Medico M
WHERE M.Specializzazione = 'Ortopedia'
```

poi i pazienti visitati da suddetti medici:

```
SELECT V.Paziente
FROM Visita V
WHERE V.Medico IN (...)
```

e si rimuovono da una tabella dei soli pazienti senesi:

```
SELECT COUNT(*)
FROM Paziente P
WHERE P.Citta = 'Siena'
  AND P.CodFiscale NOT IN (
    SELECT V.Paziente
    FROM Visita V
    WHERE V.Medico IN (
      SELECT M.Matricola
      FROM Medico M
      WHERE M.Specializzazione = 'Ortopedia'
    )
  );
```

Subquery scalari

Una subquery scalare produce, invece che un insieme di record che poi verrà controllato con IN e NOT IN, un singolo record di un singolo attributo, come ad esempio farebbe la funzione COUNT(). A questo punto, il valore scalare prodotto può essere controllato con i vari operatori di confronto. Vogliamo ad esempio indicare il numero degli otorini aventi parcella più alta della media delle parcelle della loro specializzazione. Troviamo innanzitutto il la parcella media degli otorini (un valore scalare):

```
SELECT AVG(Medico.Parcella)
FROM Medico M
WHERE M.Specializzazione = 'Otorinolaringoiatria'
```

e la useremo in un'outer query:

```
SELECT COUNT(*)
FROM Medico M1
WHERE M1.Parcella >  (
                      SELECT AVG(Medico.Parcella)
                      FROM Medico M
          WHERE M.Specializzazione = 'Otorinolaringoiatria'
                      );
```

Un caso interessante: mettiamo di voler trovare le entrate complessive generate dai cardiologi della clinica negli ultimi 2 anni. Ciò risulta semplice con un join:

```
SELECT SUM(M.Parcella) AS IncassoTotale
FROM Visita V
```

```
INNER JOIN Medico M ON V.Medico = M.Matricola
WHERE V.Data > CURRENT_DATE - INTERVAL 12 YEAR
  AND M.Specializzazione = 'Cardiologia'
```

La versione con subquery è invece più complessa, richiedendo una subquery nel SELECT:

```
SELECT SUM((
  SELECT M.Parcella
  FROM Medico M
  WHERE M.Matricola = V.Medico
     AND M.Specializzazione = 'Cardiologia'
)) AS IncassoTotale
FROM Visita V
WHERE V.Data > CURRENT_DATE - INTERVAL 12 YEAR
```

Notiamo che la subquery è correlata, come vedremo nella prossima lezione.

3 Raggruppamento

Il raggruppamento (o aggregazione) divide in gruppi i record risultanti dalle clausole FROM e WHERE (detti record target), sulla base di un certo attributo particolare in comune, che resta quindi costante in un determinato gruppo. Poniamo di voler calcolare, nello schema clinica, per ogni specializzazione, la parcella media dei medici:

```
SELECT Specializzazione, AVG(Parcella) AS ParcellaMedia
FROM Medico
GROUP BY Specializzazione;
```

Dove il select di Specializzazione e AVG(Parcella) ha senso solo a seguito del GROUP BY, con Specializzazione proprio come attributo di raggruppamento (!).

Condizioni sui gruppi

Le condizioni sui gruppi sono controllate gruppo per gruppo (non record per record) e permettono di scartarne alcuni. Si applicano attraverso l'operatore HAVING. Poniamo di voler indicare le specializzazioni della clinica con più di due medici:

```
SELECT Specializzazione
FROM Medico
GROUP BY Specializzazione
HAVING COUNT(*) > 2
```

Si nota che il DISTINCT non è necessario: ogni specializzazione viene presa comunque una volta sola. Vediamo adesso un caso un attimo più complesso: ottenere le specializzazioni con la più alta parcella media. Dovremo allora

calcolare tutte le medie delle parcelle dei medici di ciascuna specializzazione, calcolare tra queste la più alta, e selezionare quindi le specializzazioni la cui media corrisponde a questa media massima.

Il massimo si troverà con:

```
SELECT MAX(D.MediaParcelle)
FROM (
  SELECT M2.Specializzazione, AVG(M2.Parcella) AS
   MediaParcella
  FROM Medico M2
  GROUP BY M2.Specializzazione
)
AS D;
```

e la query completa sarà:

```
SELECT M.Specializzazione
FROM Medico M
GROUP BY Specializzazione
HAVING AVG(Parcella) =
(
  SELECT MAX(D.MediaParcelle)
  FROM (
    SELECT M2.Specializzazione, AVG(M2.Parcella) AS
   MediaParcella
    FROM Medico M2
    GROUP BY M2.Specializzazione
  )
  AS D;
)
```

4 Subquery correlate

Abbiamo già visto le subquery non correlate (*non-correlated*): il result set di tali subquery viene calcolato una sola volta ed è indipendente dalle specifiche della query esterna (*outer query*). Nelle subquery correlate invece il result set dipende da ciascuna tupla della tupla esterna. L'ordine di esecuzione è: prima si esegue il WHERE della query esterna, e poi la subquery. Poniamo ad esempio di voler indicare matricola e parcella dei medici che hanno visitato per la prima volta almeno un paziente nel mese di ottobre 2013. Avremo bisogno di selezionare pazienti che non erano mai stati visitati da un determinato medico prima di ottobre:

```
SELECT DISTINCT V1.Medico
FROM Visita V1
WHERE YEAR(V1.Data) = 2013
  AND MONTH (V1.Data) = 10
  AND V1.Paziente NOT IN (
                            SELECT V2.Paziente
                            FROM Visita V2
```

```
                                WHERE V2.Medico = V1.Medico
                                  AND V2.Data < V1.Data
                              )
```

Notiamo la particolarità della subquery correlata: la relazione V1, appartenente alla query esterna, viene usata nella clausola WHERE della correlata. Ricordiamo che una subquery correlata viene eseguita nuovamente per ogni tupla, ed è quindi particolarmente inefficiente dal punto di vista della complessità. Notiamo inoltre che una subquery può essere usata anche nella clausola SELECT della query, per calcolare velocemnte un valore da inserire nel risultato. Ad esempio, vogliamo trovare tutti i pazienti di sesso maschile, indicandone nome e numero di visite effettuate. Quest'ultima informazione non sarà un dato da proiettare (nel WHERE), ma da calcolare nel SELECT.

```
SELECT Nome, (
  SELECT COUNT(*)
  FROM Visite V
  WHERE V.Paziente = P.CodFiscale
) AS NumeroVisite
FROM Paziente P
WHERE P.Sesso = 'M'
```

Correlated subquery nel SELECT

L'inserimento di subquery correlate all'interno del SELECT permette di calcolare valori da inserire in un attributo dell'insieme risultato. Proprio per questo motivo le subquery di questo tipo dovranno obbligatoriamente essere scalari. Poniamo ad esempio di voler considerare il nome e il numero di visite effettuate da tutti i pazienti di sesso maschile. Attraverso il raggruppamento, si potrà semplicemente dire:

```
SELECT P.Nome, COUNT(*) AS NumeroVisite
FROM Visita V INNER JOIN Paziente P ON V.Paziente = P.
   CodFiscale
WHERE P.Sesso = "M"
GROUP BY P.CodFiscale
```

Lo stesso comportamento può essere ottenuto attraverso le subquery correlate, ad esempio con:

```
SELECT P.Nome, (
  SELECT COUNT(*)
  FROM Visite V
  WHERE V.Paziente = P.CodFiscale
) AS NumeroVisite
FROM Paziente P
WHERE P.Sesso = "M"
```

Si può assumere che il modulo del DBMS detto *query optimizer*, che ha il compito di ottimizzare le query, tradurrà una query simile nell'equivalente che utilizza il raggruppamento.

5 Costrutto EXISTS

Il costrutto EXISTS ci permette di verificare che il result set di una subquery correlata contenga almeno un record. La sua negazione controlla che il result set sia vuoto. Poniamo ad esempio di voler trovare medico, paziente e data delle visite di controllo del mese di gennaio 2016. Definiamo una visita di controllo come una visita in cui un medico visita un paziente già visitato precedentemente almeno una volta.

```
SELECT V_1.Medico, V_1.Paziente, V_1.Data
FROM Visita V_1
WHERE MONTH(V_1.Data) == 1 AND YEAR(V_1.Data) = 2016
  AND EXISTS
    (
    SELECT *
    FROM Visita V_2
    WHERE V_2.Paziente = V_1.Paziente
      AND V_2.Medico = V_1.Medico
      AND V_2.Data < V_1.Data
    )
```

Diciamo invece di voler indicare le matricole dei medici che hanno visitato per la primma volta almeno un paziente nel mese di ottobre 2013.

```
SELECT DISTINCT V_1.Medico
FROM Visita V_1
WHERE MONTH(V_1.Data) = 10 AND YEAR(V_1.Data) = 2013
  AND NOT EXISTS
  (
  SELECT *
  FROM Visita V_2
  WHERE V_2.Paziente = V_1.Paziente
    AND V_2.Medico AND V_1.Medico
    AND V_1.Data < V_2.Data
  )
```

6 Divisione insiemistica

L'operatore di divisionie insiemistica permette di trovare tutti i record che rispettano condizioni esaustive. Ad esempio, possiamo usarlo per indicare i

pazienti visitati da tutti i medici. L'SQL non fornisce un'operatore di divisione insiemistica di per sé, ma possiamo crearne uno attraverso altri costrutti. Esistono due metodi:

- Possiamo usare il costrutto EXISTS e le subquery correlate per ottenere la seguente query:

```
    SELECT P.CodFiscale
    FROM Paziente P
    WHERE NOT EXISTS
    (
      SELECT *
      FROM Medico M
      WHERE NOT EXISTS
      (
        SELECT *
        FROM Visita V
        WHERE V.Medico = M.Matricola
        AND V.Paziente = P.CodFiscale
      )
    )

```

  La query si traduce pressapoco con "un paziente è stato visitato da tutti i medici se non esiste un medico per cui non esiste una visita di quel medico con quel paziente".

- Possiamo usare il raggruppamento e una subquery non correlata:

```
    SELECT V.Paziente
    FROM Visita V
    GROUP BY V.Paziente
    HAVING COUNT(DISTINCT V.Medico) = (SELECT COUNT(*)
                                       FROM Medico)

```

  Che è considerevolmente più semplice.

7 SQL Procedurale

Introduciamo adesso il PL/SQL, un linguaggio di programmazione procedurale che estende l'SQL.

Stored procedure

Le stored procedure (procedure "stoccate") sono dei programmi dichiarativo-procedurali che possono essere memorizzati nel DBMS e invocati proprio come funzioni nei linguaggi di programmazione tradizionali, restituendo valori.

Procedure possono essere dichiarate e poi chiamate all'interno di query, o altre procedure.

Interfacce con stored procedure

Il meccanismo delle stored procedure può essere usato per implementare delle interfacce: all'utente non si mette a disposizione l'SQL, ma soltanto le procedure definite su quel DBMS. In questo modo l'utente non ha mai effettivamente accesso al database stesso, ma può comunque interagirci con procedure già definite. Questo sistema assicura il mascheramento dei dati e del codice, aumentando la sicurezza e la protezione da eventuali attacchi. Meccanismi di questo tipo sono alla base di architetture **multilivello** (*multi-tier*) usati ad esempio nei siti internet: l'uso di procedure predefinite per l'interazione fra server e database, e quindi fra utente e server, permette di aumentare la sicurezza del sistema. L'invocazione di procedure richiede, da parte del chiamante, il possesso di un particolare **grant** ("permesso"), che garantisce l'accesso del suddetto a tali procedure. E' ad esempio possibile che un'applicazione abbia accesso alle procedure ma non alle tabelle del database. Vediamo un'esempio. Vogliamo scrivere una stored procedure che restituisca tuttte le specializzazioni mediche offerte dalla clinica. Dovremo usare la seguente sintassi:

```
DROP PROCEDURE IF EXISTS mostra_specializzazioni
DELIMITER $$
CREATE PROCEDURE mostra_specializzazioni()
  BEGIN
    SELECT DISTINCT Specializzazione
    FROM Medico
  END $$
DELIMITER;
```

Il pasticcio di delimitatori serve a far ignorare all'SQL l'uso del ; all'interno della procedura, che altrimenti significherebbe "compila quanto hai letto fino ad ora". Alla fine della dichiarazione reimpostiamo il delimitatore a quello di default.
Potremo adesso chiamare la procedure con:

```
CALL mostra_specializzazioni()
```

Variabili locali

All'interne delle stored procedure è possibile memorizzante informazioni intermedie di ausilio, attraverso variabili locali. La dichiarazione di variabili locali va fatta contestualmente alla dichiarazione della procedura (cioè subito) con la sintassi:

```
DECLARE nome_variabile tipo(size) DEFAULT valore_default;
```

La tipizzazione è forte, e sono presenti i classici tipi (int, double, char, date...). Il valore size rappresenta la dimensione della variabile: ad esempio, per il tipo char, potremo specificare char(50) per indicare una stringa di 50 caratteri. Il tipo varchar è un tipo particolare che può variare la sua dimensione, occupando *al massimo* il valore fornito alla dichiarazione. Si può fare assegnamento sulle variabili attraverso la parola chiave SET:

```
DECLARE min_visite_mensili INT DEFAULT 0;

% il corpo della procedura...

SET min_visite_mensili = 20;
```

Facciamo un'esempio: supponiamo di essere nel body di una stored procedure e creare una variabile contenente il numero minimo di visite effettuate questo mese, avremo:

```
DECLARE visite_mese_attuale INT DEFAULT 0;

% il corpo della procedura...

SET visite_mese_attuale = (
  SELECT COUNT(*)
  FROM Visita V
  WHERE MONTH(V.Data) = MONTH(CURRENT_DATE)
    AND YEAR(V.Data) = YEAR(CURRENT_DATE)
)
```

oppure, analogamente:

```
DECLARE visite_mese_attuale INT DEFAULT 0;

% il corpo della procedura...

SELECT COUNT(*)
FROM Visita V
WHERE MONTH(V.Data) = MONTH(CURRENT_DATE)
  AND YEAR(V.Data) = YEAR(CURRENT_DATE)
INTO visite_mese_attuale
```

Notiamo l'errore (di impedenza) in cui potremmo incorrere facendo qualcosa del tipo:

```
DECLARE visite_mese_attuale VARCHAR(255)
SELECT * INTO visite_mese_attuale
FROM Visita V
WHERE MONTH(V.Data) = MONTH(CURRENT_DATE)
  AND YEAR(V.Data) = YEAR(CURRENT_DATE)
```

Non possiamo infatti immagazzinare (anzi, "collassare") un'intero record in una sola variabile di tipo VARCHAR.

Variabili user-defined

Le variabili user-defined sono variabili definite dall'utente senza necessità di dichiarazione, che hanno tempo di vita uguale alla durata della connessione dell'utente al server. A differenza delle variabili locali, le variabili user-defined hanno tipizzazione debole e dinamica: potremo memorizzarvici qualsiasi tipo di valore, e anche valori di tipi diversi in momenti diversi. Non possono contenere result set, ma solo valori scalari. Si dichiarano con @.

Parametri di una stored procedure

Una stored procedure MySQL accetta parametri di tipo:

- **Ingresso**
  Un parametro di ingresso può essere letto, ma non modificato. Equivale al passaggio per valore del C++. Si dichiara con la parola chiave IN. Ad esempio, poniamo di voler scrivere una stored procedure che stampi la parcella media di una specializzazione specificata come parametro:

```
DROP PROCEDURE IF EXISTS parcella_media_spec;
DELIMITER $$
CREATE PROCEDURE parcella_media_spec(IN _specializzazione
    VARCHAR(100))
% resto del codice...
DELIMITER;
```

  a questo punto potremo chiamare la procedura con:

```
CALL parcella_media_spec('Ortopedia');
```

- **Uscita**
  Un parametro di uscita può essere modificato dalla procedura, e viene fornito in fase di chiamata dal chiamante. Nella chiamata si possono usare, nei parametri di uscita, variabili user defined (@variabile).
  Facciamo un'esempio: vogliamo scrivere una stored porcedure che restituisca *come parametro di uscita* il numero di pazienti visitati da medici di una data specializzazione, ricevuta come parametro di ingresso:

```
DROP PROCEDURE IF EXISTS parcella_media_spec;
DELIMITER $$
CREATE PROCEDURE parcella_media_spec(IN _specializzazione
    VARCHAR(100), OUT totale_pazienti INT)
% resto del codice...
DELIMITER;
```

  come prima, potremo adesso chiamare la procedura con la sintassi:

```
CALL tot_pazienti_visitati_spec('Neurologia',
   @quantiPazienti);
SELECT @quantiPazienti
```

dove @quantiPazienti è una variabile user-defined.
Si noti che si possono avere più parametri di uscita in una procedura SQL.

- **Ingresso-uscita**
  Non vengono trattate da questo corso. In altre parole t'attacchi.

Istruzioni condizionali

Le espressioni condizionali permettono di esprimere condizioni, modificando il flusso di esecuzioni. Possono contenere letterali, variabili e funzioni. Nell'SQL si riducono alle due parole chiave: IF e CASE.

- **Istruzione IF**
  L'IF è analogo a quello di altri linguaggi, con l'inclusione del THEN:

```
IF if_condition THEN
  -- blocco di istruzioni
ELSEIF elseif_1_condition THEN
 ...
ELSEIF elseif_N_condition THEN
ELSE
  -- blocco di else
END IF;

```

- **Istruzione CASE**
  Vediamo il CASE:

```
CASE
WHEN condition_1 THEN
  -- blocco 1
 ...
WHEN condition_n THEN
  --blocco n
END CASE
```

Istruzioni iterative

Le struzioni iterative permettono di ripetere blocchi di codice. L'SQL fornisce il WHILE, il REPAT, e il LOOP.

- **Istruzione WHILE**
  Corrisponde al semplice while.

```
WHILE condition DO
  -- blocco istruzioni
END WHIILE
```

- **Istruzione REPEAT**

```
REPEAT
  -- blocco istruzioni
UNTIL condition
END REPEAT
```

- **Istruzione LOOP**
  Definisce un LOOP, che andrà chiuso con un'istruzione LEAVE.

```
loop_label: LOOP
  -- blocco di istruzioni, check di condizioni
END LOOP
```

Istruzioni di salto

Le istruzioni di salto permettono di interrompere cicli o passare a iterazioni successive. Nell'SQL sono rispettivamente la LEAVE e ITERATE.

Cursore

Un cursore scorre i record su un result set, solo in avanti, per effettuare delle azioni all'interno di istruzioni iterative. Vediamo la sintessi:

```
DECLARE NomeCursore CURSOR FOR
SQL query;
```

I cursori si possono dichiarare immediatamente dopo la dichiarazione di tutte le variabili, contestualmente alla dichiarazione della procedura. Su un cursore valgono le seguenti operazioni:

```
OPEN NomeCursore;
FETCH NomeCursore INTO ListaVariabili;
CLOSE NomeCursore;
```

La prima e l'ultima operazione sono banali. Il FETCH si limita a restituire il prossimo record e avanzare il cursore.

Handler

Gli handler sono gestori di situazioni, utili per eseguire codice quando l'istruzione fetch porta il cursore in fondo alla tabella. Possono essere definiti dopo le dichiarazioni di variabili e cursori, sempre contestualmente alla dichiarazione della procedura. Ad esempio, esiste il NOT FOUND HANDLER:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND
SET finito = 1;
```

Questo handler viene eseguito quando si arriva a "fine corsa", ovvero in fondo alla tabella che si stava scorrendo. La parola chiave CONTINUE rappresenta

il fatto che l'handler non interrompe l'esecuzione, a differenza di un handler EXIT.

Ciclo di fetch

Il meccanismo del FETCH e degli HANDLER ci permette di stabilire un ciclo di fetch:

```
DECLARE cur CURSOR FOR tabella
DECLARE CONTINUE HANDLER FOR NOT FOUND
SET finito = 1;

scan: LOOP
FETCH cur -- prelieva il record
IF finito = 0
  -- processo il record
ELSE LEAVE scan
END LOOP scan;
```

8 Manipolazione dei dati

Vediamo adesso i costrutti della parte **DML** (*Data Manipulation Language*), dell'SQL.

Inserimento

L'inserimento consiste nel, considerata una nuova tabella, inserire un nuovo record i cui valori degli attributi possono essere sia statici che ricavati. Inseriamo un nuovo paziente nel database, notiamo i dati (statici):

- Nome: Elvira
- Cognome: Passerotti
- Sesso: F
- Data di nascita: 27 Ottobre 1964
- Città: Pisa
- Reddito: 1500 Euro
- Codice Fiscale: PSSLVR65R67G702U

```
INSERT INTO Paziente
VALUES ('PSSLVR65R67G702U', 'Passerotti', 'Elvira', 'F', '
   1965-10-27', 'Pisa', 1500);
```

Dove è importante rispettare l'ordine di definizione degli attributi della tabella. Facciamo un'altro esempio: vogliamo inserire il paziente Edoardo Lepre, visitato tre giorni fa, di codice fiscale "slq6". Poniamo però di non sapere se la visita era stata mutuata o meno:

```
INSERT INTO Visita(Medico, Paziente, Data)
VALUES(010, 'slq_6', CURRENT_DATE - INTERVAL 3 DAY);
```

Questo mi permette di specificare solo alcuni valori, lasciando gli altri a NULL o al loro valore di default. Notiamo che, in generale, la sintassi:

```
INSERT INTO Tabella [(Attributo_1, Attributo_2, ...)]
VALUES (Valore_1, Valore_2, ...);
```

Ci permette di definire qualsiasi attributo in qualsiasi ordine, anche diverso da quello di definizione.

Inserimento con valori ricavati

E' possibile definire un'inserimento del tipo:

```
INSERT INTO Tabella [(Attributo_1, Attributo_2, ...,
   Attributo_N)]
Query_di_selezione;
```

Attraverso una query "Query_di_selezione" arbitraria che restituisca valori per tutti gli attributi. Poniamo di voler archiviare tutte i record delle visite effettuate prima di due anni fa in una nuova tabella, VisitaVecchia(CognomePaziente, CognomeMedico, DataVisita, Specializzazione):

```
INSERT INTO VisitaVecchia
SELECT P.Cognome AS CognomePaziente,
  M.Cognome AS CognomeMedico,
  V.Data AS DataVisita,
  M.Specializzazione
FROM Visita V INNER JOIN Medico M ON M.Matricola = V.Medico
  INNER JOIN Paziente P ON P.CodFiscale = V.Paziente
WHERE YEAR(V.Data) <= YAER(CURRENT_DATE) - 2;
```

Come vediamo, basterà definire una query che restituisce tutti i record da inserire nella tabella nell'ordine di definizione (dato nel SELECT).

Aggiornamento

Possiamo anche aggiornare record già definiti. Dovremo però usare una clausola WHERE per specificare quali record aggiornare:

```
UPDATE Tabella
SET Attributo_i = Valore_i, ecc...
WHERE Condizione
```

Mutuiamo tutte le visite del mese corrente effettuate da pazienti nati prima del 1925:

```sql
UPDATE Visita
SET Mutuata = 1
WHERE MONTH(Data) = MONTH(CURRENT_DATE)
  AND YEAR(Data) = YEAR(CURRENT_DATE)
  AND Paziente IN (
    SELECT CodFiscale
    FROM Paziente
    WHERE YEAR(DataNascita) < 1925
  );
```

Cancellazione

Vediamo come cancellare record dalla tabella. La sintassi è analoga a quella dell'aggiornamento:

```sql
DELETE FROM Tabella
WHERE Condizione
```

Poniamo di voler licenziare tutti i medici di Pisa che non hanno effettuato visite mutate il mese scorso. Scriviamo la query:

```sql
DELETE FROM Medico
WHERE Matricola IN (
  SELECT M_1.Matricola
  FROM Medico M_1 LEFT OUTER JOIN (
    SELECT V_2.Medico AS MedicoMutuato
    FROM Visita V_2 INNER JOIN Medico M_2 ON M_2.Matricola =
   V_2.Medico
    WHERE M_2.Citta = 'Pisa'
      AND V_2.Mutuata = TRUE
      AND V_2.Data > CURRENT_DATE - INTERVAL 1 MONTH
    ) AS D
    ON M_1.Matricola = D.MedicoMutuato
    WHERE D.MedicoMutuato IS NULL
);
```

Il DBMS riceve la query e restituisce un errore. Questo perchè, nel momento in cui dichiariamo l'operazione di cancellazione con DELETE FROM Medico; stiamo effettivamente bloccando la tabella. Non possiamo quindi definire una query che legga tale tabella: il DBMS ce lo impedirà in quanto potremmo generare loop infiniti. Possiamo risolvere questa situazione in più modi.

- Trasformare la query in una derived table:

```sql
DELETE FROM Medico
WHERE Matricola IN (
  SELECT * FROM (
    SELECT M_1.Matricola
    FROM Medico M_1 LEFT OUTER JOIN (
     SELECT V_2.Medico AS MedicoMutuato
```

```
      FROM Visita V_2 INNER JOIN Medico M_2 ON M_2.
    Matricola = V_2.Medico
      WHERE M_2.Citta = 'Pisa'
        AND V_2.Mutuata = TRUE
        AND V_2.Data > CURRENT_DATE - INTERVAL 1 MONTH
      ) AS D
      ON M_1.Matricola = D.MedicoMutuato
      WHERE D.MedicoMutuato IS NULL
  ) AS D_2
);
```

Questo assicura che il valore della derived table venga calcolato prima dell'operazione di cancellazione, eludendo quindi il blocco imposto dal DBMS. La stessa cosa potrà essere fatta, anziché con una derived table, usando una CTE.

- Un'altro approccio, meno intuitivo, è quello di usare un join:

```
DELETE M_1.*
FROM Medico M_1 LEFT OUTER JOIN (
  SELECT V_2.Medico AS MedicoMutuato
  FROM Visita V_2 INNER JOIN Medico M_2 ON M_2.Matricola
   = V_2.Medico
  WHERE M_2.Citta = 'Pisa'
    AND V_2.Mutuata = TRUE
    AND V_2.Data > CURRENT_DATE - INTERVAL 1 MONTH
 ) AS D
ON M_1.Matricola = D.MedicoMutuato
WHERE D.MedicoMutuato IS NULL;
```

9 Database attivi

Abbiamo visto finora le metodistiche dall'SQL procedurale, che permette di implementare funzionalità in linguaggio, appunto, procedurale, sebbene il paradigma dell'SQL sia in teoria dichiarativo. Adesso vedremo alcune caratteristiche dell'SQL che permettono di implementare comportamenti "attivi" del database: per la precisione, **trigger** e **event**.

Trigger

Un trigger è una procedura che viene eseguita sulla base di eventi di istruzione DML (*data manipulation language*). Un trigger è fornito di una **parte reattiva** che reagisce a eventi DML, che causa l'esecuzione di un'azione (operazione) sui dati. Gli eventi DML scatenanti possono ad esempio essere modifiche, cancellazioni, inserzioni.

La **condizione** è un predicato booleano che viene valutato dopo l'evento. Un risultato positivo della condizione comporta l'esecuzione dell'azione. Abbiamo quindi:

1. Evento;
2. Condizione;
3. Azione.

Facciamo un'esempio: voglliamo gestire un'attributo ridondante nella tabella Paziente contenente la data nella quale un paziente è stato visitato l'ultima volta. Sarà un'attributo ridondante perché lo potremo già tovare nella tabella Visita. D'altronde, ad ogni successiva visita, l'attributo sulla tabella paziente andrà aggiornato. In ogni caso, il comportamento che vogliamo è che, all'inserzione dell'ultima visita nella tabella visita da parte di un'operatore, il database possa automaticamente (*database attivo*) inserire l'informazione necessaria nella tabella paziente attraverso un trigger.
Individuiamo evento, condizione ed azione:

- **Evento:** l'evento sarà l'inserimento di una nuova visita nella tabella Visita;
- **Condizione:** qui la condizione non c'é: ogni nuova visita è un'ultima visita.
- **Azione:** rintraccia il paziente della visita, e aggiorna la sua ultima visita.

in SQL, la sintassi di un trigger sarà:

```
DROP TRIGGER IF EXISTS nome_trigger
CREATE TRIGGER nome_trigger
[BEFORE | AFTER][INSERT |UPDATE|DELETE] ON target
FOR EACH ROW blocco_istruzioni
```

Vediamo i dettagli. Con BEFORE si indica un'azione di *preprocessing* quindi di esecuzione prima dell'evento, mentre con AFTER si indica un'azione *a posteriori*, o "collaterale", che viene eseguita dopo l'evento. La sintassi che vorremo nell'esempio appena sopra sarà quindi:

```
DROP TRIGGER IF EXISTS aggiorna_ultima_visita;
CREATE TRIGGER aggiorna_ultima_visita
AFTER INSERT ON Visita FOR EACH ROW
  UPDATE Paziente
  SET UltimaVisita = CURRENT_DATE
  WHERE CodFiscale = NEW.Paziente
```

dove NEW indica il record che è stato già inserito (in un trigger AFTER; in un trigger BEFORE sarebbe stato quello che sta per essere inserito)..

Trigger multi-statement

Un trigger *multi-statement* ("multi-blocco") è formato da più blocchi separati da punti e virgola. Si rende quindi necessario, come avevamo già visto, l'uso di un delimitatore ausiliario:

```
DROP TRIGGER IF EXISTS nome_trigger;
DELIMITER $$
CREATE TRIGGER nome_trigger
BEFORE ... ON ...
FOR EACH ROW

BEGIN
-- blocchi di istruzioni
END $$
DELIMITER ;
```

Regole aziendali

Le regole aziendali (*business rule*) sono determinate regole della realtà d'interesse da modelizzare nel DBMS. Il meccanismo dei trigger fornisce un modo per assicurarsi che tali regole vengano rispettate. Poniamo ad esempio la regola aziendale: "Ogni mese, le visite mutuate di un medico non possono essere più di 10".

```
DROP TRIGGER IF EXISTS blocca_non_mutuate;
DELIMITER $$
CREATE TRIGGER blocca_non_mutuate
BEFORE INSERT ON Visite
FOR EACH ROW
BEGIN

  DECLARE non_mutuate_mese INTEGER DEFAULT 0;

  SET non_mutuate_mese = (
    SELECT COUNT(*)
    FROM Visita V
    WHERE M.Medico = NEW.Medico
      AND MONTH(V.Data) = MONTH(CURRENT_DATE)
      AND YEAR(V.Data) = YEAR(CURRENT_DATE)
      AND V.Mutuata = 1
  );
  IF non_mutuate_mese = 10 THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Visita non inseribile';
  END IF;

END $$
```

```
24 DELIMITER ;
```

Il comando "SIGNAL SQLSTATE '45000'" serve a lanciare un'errore (segnale), di cui impostiamo subito dopo il messaggio d'errore, e impedire l'inserimento. Nello specifico, il codice 45000 sta per "errore generico". In ogni caso, l'*exception handling* è al di fuori degli argomenti del corso.

Event

Gli **event** sono procedure simili ai trigger, ma la loro causa scatenante è il raggiungimento di uno specifico istante temporale. Ad esempio possono essere usati per eseguire azioni periodice (giornalmente, mensilmente, ecc...). Poniamo per esempio di voler creare e mantenere giornalmente aggiornata una ridondanza nella tabella Medico contenente il totale delle visite effettuate. Potremo dire:

```
1  CREATE EVENT AggiornaTotaliVisite
2  ON SCHEDULE EVERY 1 DAY
3  STARTS '2023-05-25 23:55:00'
4  DO
5    UPDATE Medico
6    SET TotaleVisite = TotaleVisite +
7                       (SELECT COUNT(*)
8                        FROM Visita V
9                        WHERE V.Medico = Matricola
10                        AND V.Data = CURRENT_DATE);
```

Notiamo la parola chiave STARTS per impostare la data di inizio. Esiste inoltre la variante STARTS - ENDS per impostare anche la data di fine della schedulazione.

Attivazione dello scheduler

L'attivazione dello scheduler si ottiene impostando la varibile di sistema event_scheduler, come:

```
1 SET GLOBAL event_scheduler = ON;
```

10 Materialized view

Le materialized view sono viste ridondanti, cioè ricavabili dai dati nel database (detti *raw data*), che scegliamo di precalcolare. Sostanzialmente, una materialized view è il risultato precalcolato di una query (spesso formata da un join molto corposo). A differenza delle normali view (viste), la materialized view non viene calcolata ad ogni accesso, ma precalcolata periodicamente, accumulando inevitabilmente un certo scarto temporale. Si usano quando il risultato deve essere ottenuto velocemente, ma la query necessaria a calco-

larla richiede molte risorse (e tempo).
La dichiarazione di una materialized view corrisponde a quella di una tabella:

```
CREATE TABLE MATERIALIZED_VIEW(
  Paziente CHAR(100) NOT NULL,
  NumVisite INT(11) NOT NULL DEFAULT 0,
  UltimaVisita DATE,
  PRIMARY KEY(Paziente)
) ENGINE = InnoDB DEFAULT CHARSET=latin1;
```

Politiche di refresh

Esistono più politiche di refresh (ricalcolo) delle materialized view:

- **Immediate:** dopo ogni evento, quindi equiparabili ad un **trigger**;
- **Deferred:** su base temporale, quindi equiparabili ad un **event**;
- **On demand:** avviate manualmente, quindi equiparabili ad una **stored procedure**.

Esistono inoltre due modalità di aggiornamento della materialized view:

- **Full refresh:** si aggiorna tutta la vista in blocco, da zero.
- **Incremental refresh:** si aggiornano solamente le componenti non più aggiornate. Può quindi essere un'aggiornamento sia totale che parziale.

Vediamo alcuni esempi:

- **Immediate refresh** (sync):
  Una vista materializzata ad aggiornamento immediato dopo ogni aggiornamento si può implementare come:

```
DELIMITER $$
DROP TRIGGER IF EXISTS immediate_refresh_visita $$
CREATE TRIGGER immediate_refresh_visita
AFTER INSERT ON Visita
FOR EACH ROW
BEGIN
  UPDATE MATERIALIZED_VIEW
  SET NumVisite = NumVisite + 1
    UltimaVisita = CURRENT_DATE
  WHERE Paziente = NEW.Paziente
END $$
```

  questo assumendo che ad ogni paziente corrisponderà un record in MATERIALIZED_VIEW, creato con il trigger:

```
DROP TRIGGER IF EXISTS immediate_refresh_paziente $$
CREATE TRIGGER immediate_refresh_paziente
AFTER INSERT ON Paziente
FOR EACH ROW
BEGIN
  INSERT INTO MATERIALIZED_VIEW(Paziente)
  VALUES(NEW.CodFiscale)
END $$
DELMITER ;
```

- **On demand refresh** (full):
  Vediamo una vista materializzata on demand (ad aggiornamento manuale, con procedura) di tipo full:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS on_demand_refresh $$
CREATE PROCEDURE on_demand_refresh
BEGIN
  TRUNCATE MATERIALIZED_VIEW;
  INSERT INTO MATERIALIZED_VIEW
  SELECT P.CodFiscale,
    IF(V.Paziente IS NOT NULL, COUNT(*), 0),
    IF(V.Paziente IS NOT NULL, MAX(V.Data), NULL)
  FROM Visita V
    RIGHT OUTER JOIN
    Paziente P ON P.CodFiscale = V.Paziente
  GROUP BY P.CodFiscale
END $$
DELMITER ;
```

- **Deferred refresh** (full):
  Vediamo infine come implementare il refresh in differita, appoggiandoci alla procedura on demand appena creata:

```
DELIMITER $$
DROP EVENT IF EXISTS deferred_refresh $$
CREATE EVENT deferred_refresh
ON SCHEDULE EVERY 1 WEEK
BEGIN
  CALL on_demand_refresh;
END $$
DELIMITER ;
```

Refresh incrementale

Vediamo adesso tutte quelle metodistiche che permettono di effettuare un refresh "incrementale", che va opportunamente ad effettuare una sincronizzazione nel, attraverso le modalità già viste, su una tabella che decidiamo di

mantenere aggiornata fino ad un certo istante nel tempo $t^{\star}$. L'incremental refresh si basa sul non accedere mai (o almeno, sul farlo in modo molto limitato) alle tabelle contenente i dati raw, ma di usare invece il meccanismo delle **log table**. Le log table, *"tabelle dei log"*, sono apposite strutture create per tenerre traccia delle modifiche effettuate su una o più tabelle. Una log table conterrà quindi, ad esempio, tutte le modifiche fatte ad una certa relazione, la data in cui tali modifiche sono state eventuate, ecc... Una log table è libera di contenere valori contenenti nelle nuove entrate aggiunte alle tabelle di interesse, come valori derivati ricavati da tali entrate. Sulla base della log table, si potrà poi eseguire:

- **Partial refresh**: un trasferimento parziale del log, cioè solamente delle modifiche che effettivamente comportano una variazione dei contenuti della materialized view;
- **Complete refresh**: un trasferimento totale e imparziale delle modifiche riportate sulla log table;
- **Rebuild**: una ricostruzione da zero della materialized view, sulla base delle informazioni contenute nella log table.

Assicurandosi che la ricostruzione della materialized view sulla base della log table non implichi accessi alle tabelle d'interesse, si può creare un sistema molto performante, in quanto i record della log table sono molto meno di quelli delle tabelle grezze. Questo significa tenere traccia di tutte le modifiche effettuate al database, e che queste siano sufficienti ad eseguire il refresh della vista materializzata. Nel caso la vista si basi su più tabelle, possono tornare utili più log table.

Partial refresh

Vediamo nel dettaglio il partial refresh. Il partial refresh si basa sul fatto che non tutti i record che vengono inseriti nella tabella d'interessa dopo il tempo $t^{\star}$ di aggiornamento della materialized view sono effettivamente importanti per il calcolo della materialized view. Possiamo quindi restringere le operazioni di push sulla materialized view alle sole modifiche effettivamente interessanti la materialized view. Vediamo un'esempio. Si potrà dichiarare una log table come qualsiasi altra tabella:

```
CREATE TABLE LOG_TABLE (
  Paziente CHAR(100) NOT NULL,
  DataVisita DATE
) ENGINE = InnoDB DEFAULT CHARSET=latin1;
```

Possiamo quindi definire trigger di aggiornamento della stessa, sia in caso di inserzione su paziente che su visita:

```
DELIMITER $$
DROP TRIGGER IF EXISTS push_visita $$
CREATE TRIGGER push_visita
AFTER INSERT ON Visita
FOR EACH ROW
BEGIN
  INSERT INTO LOG_TABLE
  VALUES(NEW.Paziente, CURRENT_DATE);
END $$

DROP TRIGGER IF EXISTS push_paziente $$
CREATE TRIGGER push_paziente
AFTER INSERT ON Paziente
FOR EACH ROW
BEGIN
  INSERT INTO LOG_TABLE
  VALUES(NEW.CodFiscale, NULL);
END $$
```

A questo punto saranno stabilite le procedure che manterranno in ordine la log table, e si potrà quindi procedere con l'implementazione di procedure che aggiornino la materialized view sulla base delle informazioni contenute nella stessa, magari fino a una certa data dopo $t^\star$ (si parla di refresh parziale). Notiamo adesso che è importare scegliere se concentrarsi sull'ottimizzazione delle procedure di push dalla tabella raw alla log table, o sulle procedure di refresh della materialized view. In generale, conviene ottimizzare le operazioni di push (in quanto più frequenti), e lasciare che siano le operazioni di refresh a prendere la maggior parte del tempo e delle risorse, in quanto quest'ultime verranno comunque eseguite in momenti di basso carico sul database (ad esempio di notte). Vediamo quindi la procedura d'aggiornamento parziale:

```
DROP PROCEDURE IF EXISTS on_demand_partial $$
CREATE PROCEDURE on_demand_partial(_up_to DATE)
BEGIN
  REPLACE INTO MATERIALIZED_VIEW
  WITH aggregated_log AS (
    SELECT LT.Paziente, SUM(IF(LT.DataVisita IS NULL, 0, 1))
   AS NuoveVisite,
      MAX(LT.DataVisita) AS NuovaUltima
    FROM LOG_TABLE LT
    WHERE LT.DataVisita <= _up_to OR LT.DataVisita IS NULL
    GROUP BY LT.Paziente )
  SELECT COALESCE(MV.Paziente, AL.Paziente) AS Paziente,
    IF(MV.Paziente IS NULL,
      AL.NuoveVisite,
      MV.NumVisite + IF(AL.NuovaUltima IS NULL, 0, AL.
   NuoveVisite)),
    IF(MV.Paziente IS NULL,
```

```
      IFNULL(Al.NuovaUltima, NULL),
      Al.NuovaUltima
    )
  FROM MATERIALIZED_VIEW MV
    RIGHT OUTER JOIN
    aggregated_log AL USING(Paziente);

  DELETE FROM LOG_TABLE LT
  WHERE LT.DataVisita <= _up_to OR (LT.Paziente IS NULL AND
   LT.Paziente IN (SELECT Paziente

                   FROM MATERIALIZED_VIEW));
END $$
```

La procedura usa una CTE che aggrega le entrate del log in una comoda tabella che contiene, per ogni paziente, il nome, il numero di visite inserite nel log e la più recente fra queste. Questa tabella fa quindi join con la materialized view, in modo da poter inviduare i record da aggiornare e quelli da creare da zero, attraverso le funzioni COALESCE (che sceglie il primo valore non NULL che trova) e le funzioni IF. Infine, si fa il flush delle tabelle ormai inserite della log table.

11 Data analytics

Window function

Una window function (ovvero funzione analitica) e una funzione che associa ad ogni record r un valore ottenuto da un'operazione su insieme di record logicamente connessi ad r, come ad esempio media, rank, ecc... Vediamo un'esempio di applicazione delle window function. Vogliamo scrivere una query che indichi, per ogni cardiologo, la matricola, la parcella, e la parcella media della sua specializzazione. Senza usare correlated subquery nel select, si può invece usare la clausola OVER. La clausola OVER applica una funzione di tipo aggregato o non-aggregato a un'insieme di record associati a un record di result set, detto *partition*. L'esempio sarebbe quindi:

```
SELECT M.Matricola,
  M.Parcella,
  M.Parcella
  AVG(M.Parcella) OVER()
FROM Medico M
WHERE M.Specializzazione = 'Cardiologia';
```

Si può generalizzare questa query a tutte le specializzazioni attraverso la partizione. La partizione è l'insieme di record a cui si applica la funzione

aggregata / non aggregata, e dipende dal record processato attualmente. L'esempio può essere:

```
SELECT M.Matricola,
  M.Parcella,
  M.Specializzazione,
  M.Parcella
  AVG(M.Parcella) OVER( PARTITION BY
                        M.Specializzazione)
FROM Medico M
```

Dove la clausola PARTITION BY M.Specializzazione richiede alla query di effettuare l'operazione di aggregazione AVG su sottoinsiemi formati da una partizione della tabella Medico contenente solo medici con la stessa specializzazione del medico d'interesse. ll funzionamente è simile a quello di una GROUP BY, o di una semplice aggregazione, ma mantiene l'informazione rigaurdante i singoli medici (da cui ricaviamo matricola e parcella). Con OVER si possono usare diverse funzioni aggregate, una cui lista non viene riportata qua. Di queste abbiamo già visto le AVG, COUNT, COUNT(DISTINCT), MAX, MIN e SUM. Notiamo poi che la definizione di WINDOW è concessa fuori dalla OVER:

```
SELECT M.Matricola,
  AVG(...) OVER w
FROM Medico M
WINDOW w AS ( ... )
```

Funzioni non aggregate

Le funzioni non aggregate si possono dividere in due sottocategorie: quelle che usano l'intera partizione, e quelle che usano un suo sottoinsieme (*frame*). Le window functions non aggregate associano a ciascun record di un result set un valore ottenuto senza l'aggregazione del result set stesso, ma calcolato su tutti i record a partire dall'inizio della partizione fino al record attuale se si specifica un ORDER BY nella partizione, e tutti i record della partizione in caso contrario. Alcuni esempi sono ad esempio la FIRST_VALUE, LAST_VALUE, ecc... Possiamo fare un'esempio: poniamo di voler associare un numero ad ogni medico nella sua specializzazione. Potremo usare la funzione ROW_NUMBER, che restituisce un'indice all'interno della partizinoe per ogni record:

```
SELECT M.Matricola,
  M.Cognome,
  M.Specializzazione,
  ROW_NUMBER() OVER( PARTITION BY
                     M.Specializzazione)
FROM Medico M;
```

Classificazione

Vediamo l'uso della funzione RANK per stilare classifiche. La funzione RANK permette di classificare record (banalmente, ordinare) associando ad ognuno un certo valore SCORE (punteggio). Poniamo per esempio di voler classificare i medici in base alla loro convenienza (quindi in base a parcelle più basse):

```
SELECT M.Matricola,
  M.Cognome,
  M.Specializzazione,
  M.Parcella
  RANK() OVER( ORDER BY M.Parcella )
FROM Medico M;
```

Questo query, per ogni medico, ottiene il sottoinsieme della partizione del resultset ordinato fino a quel medico (con ORDER BY), e restituisce la posizione (con RANK) di quel medico in tale partizione. Di default, la ORDER BY effettua un'ordinamento ascendente (quindi parcelle più basse in cima, parcelle più alte in fondo). Vediamo la gestione dei pareggi. In questa forma, la funzione rank salta, per ogni parimerito, un numero uguale al numero dei pareggi, di posizioni. Ergo, se due medici ottengono lo stesso punteggio, si avrà:

Nome	Parcella	Posizione
Fabio Rossi	150	1
Nicola Tonini	250	2
Tullio Nomefalso	250	2
Michele Poraccio	300	4

Per ottenere un comportamento diverso posso usare il DENSE_RANK. Il DENSE_RANK non scala la posizione nel caso di un parimerito, e dà quindi una classifica senza "buchi":

```
SELECT M.Matricola,
  M.Cognome,
  M.Specializzazione,
  M.Parcellam
  DENSE_RANK() OVER( ORDER BY M.Parcella )
FROM Medico M;
```

Nome	Parcella	Posizione
Fabio Rossi	150	1
Nicola Tonini	250	2
Tullio Nomefalso	250	2
Michele Poraccio	300	3

Si possono effettuare classifche anche in base su più sottinsiemi (i cosiddetti **rank multipli**). Stiliamo, ad esempio, la classifica delle visite effettuate sia riguardo alla totalità dei medici, sia riguardo alla specializzazione di appartenenza del medico:

```
WITH visite AS (
  SELECT V.Medico,
    M.Cognome,
    M.Specializzazione,
    COUNT(*) AS Visite
  FROM Visita V
    INNER JOIN Medico M ON M.Matricola = V.Medico
  GROUP BY V.Medico
)
SELECT VV.Cognome,
  VV.Specializzazione,
  VV.Visite,
  RANK() OVER(ORDER BY VV.Visite DESCS) AS GlobalRank,
  RANK() OVER(PARTITION BY VV.Specializzazione
              ORDER BY VV.Visite DESC) AS SpecRank
FROM visite VV;
```

Funzioni LEAD e LAG

Le funzioni LEAD e LAG ricavano il valore dell'attributo (che può esssere anche ricavato) di una row posta k posizioni prima o dopo un dato record. Chiaramente, per applicare le LEAD e LAG dev'essere stabilito una relazione di ordinamento sensata nella partizione considerata. La funzione LAG, ad esempio, prende due argomenti: il primo rappresenta l'attributo su cui effettuare il "salto", mentre il secondo rappresenta il numero di posizioni da saltare. Poniamo ad esempio di voler considerare, per tutte le visite otorinolaringoiatriche dal 2010 al 2019, la matricola del medico, il codice fiscale del paziente, la data e la data della visita precedente effettuata da un medico con la stessa specializzazione. Avremo:

```
SELECT V.Medico,
  V.Paziente,
  V.Data,
  LAG(V.Data, 1) OVER ( PARTITION BY V.Paziente
                        ORDER BY V.Data)
FROM Visita V
  INNER JOIN Medico M ON V.Medico = M.Matricola
WHERE M.Specializzazione = "Otorinolaringoiatria"
  AND YEAR(V.Data) BEWEEN 2010 AND 2019
```

Frame

Un frame è un sottoinsieme di una partizione. Per la precisione, a partire dal record corrente in una partizione, ci si può muovere N posizioni **preceding**

(precedenti) e N posizioni **following** (seguenti) da considerare. I record al di fuori delle soglie preceding e following si chiamano preceding o following **unbounded**. Una window function su un frame processa un result set e calcola valori aggregati e non sulla base di record adiacenti alla current row, ovvero su un frame della current row. Esistono alcune funzioni aggregate che lavorano solo su frame: ad esempio FIRST_VALUE e LAST_VALUE, che restituiscono rispettivamente il primo e l'ultimo valore del frame. Se non si specifica un frame, viene impostato automaticamente un default frame (che non corrisponde necessariamente alla partizione intera). Il default frame equivale al frame che contiene tutti valori precedenti nel caso di OVER con ORDER BY, e all'intera partizione nel caso di OVER semplici. Vediamo un'esempio dell'uso di FIRST_VALUE. Poniamo di voler ottenere le prime visite cardiologiche fatte da 3 specifici pazienti nel triennio 2012-2014 con ciascun medico:

```
SELECT V.Medico,
  V.Paziente,
  V.Data,
  FIRST_VALUE(V.Data) OVER w
FROM Visita V
WHERE V.Paziente IN ('aaa_1', 'bbc_4', 'ccc_2')
  AND YEAR(V.Data) BETWEEN 2012 AND 2014
WINDOW w AS ( PARTITION BY V.Medico, V.Paziente
              ORDER BY V.Data)
```

In questo caso il default frame è corretto, in quanto vogliamo il primo record del result set.

12 Introduzione alla teoria delle basi di dati

Che cos'è l'informatica?

- L'informatica è la scienza del trattamento razionale, spesso attraverso macchine automatiche, dell'informazione.

possiamo fare una distinzione fra:

- metodologica
- tecnologica

13 Sistema informativo

- Il sistema organizzativo è costituito da risorse e regole per lo svolgimento coordinato di attività di una certa organizzazione (azienda, ente, ecc...) Noi ci concentreremo sul
- Sistema informativo, ovvero la parte del sistema organizzativo che acquisice, conserva, elabora e produce informazioni d'interesse per l'organizzazione. Inoltre esegue e gestisce i processi informativi (che coinvolgono informazioni).

possiamo analizzare più nel dettaglio i tipo di attività svolte dal sistema informativo:

- Raccolta e acquisizione di informazioni
- Archiviazione e conservazione di suddette informazioni
- Elaborazione, trasformazione e ancora produzione di nuove informazioni sulla base di quelle ottenute
- Distribuzione e comunicazione delle informazioni così elaborate.

Occorre fare attenzione: il sistema informativo non è di per se in alcun modo legato all'informatica (esistono sistemi informativi, si pensi ai servizi anagrifici o alle banche, che non usano alcuna automatizzazione).
La parte del sistema informativo che usa la tecnologia informatica è il sistema informativo automatizzato, o semplicemente **sistema informatico**.
Ricapitolando, possiamo stabilire la seguente relazione:

Azienda → Sistema organizzativo → Sistema informativo → sistema informatico

14 Gestione delle informazioni

L'informazione può essere gestita secondo modalità diverse, e su supporti diversi. Ad esempio, abbiamo:

- Idee informali
- Linguaggio naturale
- Disegni, schemi, grafici

- Numeri e codici

su vari supporti:

- Mente umana
- Carta
- Dispositivi elettronici (e.g. hard disk, ecc...)

Definiamo ora la differenza fra informazioni e dati:

- **Informazione**
  Notizia, dato o elemento che consente di avere una conoscenza dei fatti, situazioni o modi di essere.
- **Dato**
  Ciò che è immediatamente presente alla conoscenza prima dell'elaborazione. In informatica, elementi di informazione costituiti da *simboli* che devono essere elaborati.

I dati sono spesso codifiche particolari di informazioni, che vanno quindi da essi estrapolate. Ad esempio, il codice fiscale, i cartelli stradali, ecc...

15 La base di dati

Il cuore di un sistema informativo automatizzato è la base di dati (database), cioè un insieme organizzato di dati rappresentanti informazioni di interesse. Nelle due accezioni (metodologica e tecnologica), possiamo dire:

- Metodologica: insieme organizzato di dati utilizzati come supporto per lo svolgimento di attività
- Tecnologica: insieme di dati gestito da un DBMS (Database Management System)

Le basi di dati hanno solitamente:

- Dimensioni molto maggiori della memoria centrale dei sistemi di calcolo utilizzati
- Tempo di vita indipendente dalle signole istanze dei programmi che li utilizzano (persistenza dei dati)
- Supporto per gestione di collezioni di dati condivise fra più dispositivi

- Capacità di garantire privacy, affidabilità, efficienza ed efficacia

Vediamo nel dettaglio dell'aspetto di condivisione:

- Ogni organizzazione è divisa in settori o almeno svolge disparate attività
- Ciascun settore potrebbe essere fornito di un sottosistema informativo, magari disgiunto a quello principale.

Questo può chiaramente portare a problemi di:

- **Ridondanza**: ripetizione dell'informazione
- **Incoerenza**: più versioni dell'informazione che non coincidono.

Nota: perchè non usare un semplice archivio di file invece che di un database?
Un archivio non fornisce alcuna gestione dell'interdipendenza fra informazioni, ed è quindi poco portato agli eventuali controlli sulla coerenza e la correttezza dell'informazione. Inoltre, in un comune filesystem abbiamo a disposizione solamente le operazioni rudimentali di scrittura/lettura, senza particolari controlli su concorrenza o funzionalità particolari.

Per ovviare a tutta questa serie di problemi, introduciamo:

- **Autorizzazione**: gestione dell'accesso a date risorse
- **Concorrenza**: gestione dell'accesso *simultaneo* (!) a date risorse
- **Affidabilità**: resistenza a malfunzionamenti hardware e software. Una tecnica fondamentale in questo campo è la corretta gestione delle **transazioni**
- **Efficienza**: gestione ottimale delle risorse in termini di memoria e tempo
- **Efficacia**: offerta di funzionalità articolate, potenti e flessibili.

Vediamo un'attimo nel dettaglio l'aspetto delle transazioni:

Transazioni

Una transazione è un insieme di operazioni da considerare indivisibili (*"atomiche"*), corretto anche in presenza di concorrenza, e con effetti definitivi a fine esecuzione. Una transazione deve essere eseguita *per intero* o *per niente*. La corretta gestione della concorrenza deve invece assicurarsi che transazioni

concorrenti vengano gestite correttamente (o in serie, e. g. transazioni bancarie, o in maniera mutualmente esclusiva, e. g. prenotazione biglietto aereo). La transazione dovrà poi essere permanente, ovvero la conclusione positiva di una transazione corrisponde ad un impegno (commit) a mantenere traccia della versione ormai aggiornata dei dati a seguito della stessa.

Descrizione dei dati
Nei programmi tradizionali che accedono a file, ogni programma contiene una descrizione della struttura del file stesso, con i conseguenti rischi di incoerenza fra informazioni e file stessa. Ecco perchè nei DBMS è opportuno dedicare una porzione della base di dati alla descrizione centralizzata dei dati. Introduciamo il concetto di **modello dei dati**: un insieme di costrutti utilizzati per organizzare i dati di interesse e descriverne le varie dinamiche, fornendoci quindi una vista astratta dei suddetti. Occorre quindi definire la differenza fra:

- **Schema**: descrizione della struttura della dei dati, solitamente invariante nel tempo (attributi)
- **Istanza**: i valori attuali immagazzinati nella base di dati, che variano nel tempo (record).

introdiciamo l'elemento fondamentale della base di dati: la **tabella**. Una comune tabella di una base di dati contiene come intestazioni delle sue colonne gli attributi (schema) dei dati immagazinati, mentre le successive righe presentano vari record (istanze) dei dati definiti dalle colonne.

16 Modelli dei Dati

Modelli logici
Adottati nei DBMS esistenti per l'organizzazione dei dati, indipendenti dalle strutture fisiche. Esempi: relazionale, reticolare, gerarchico, a oggetti, basato su XML.

Modelli concettuali
Permettono di rappresentare i dati in modo indipendente da ogni sistema, cercando di descrivere i concetti del mondo reale. Sono utilizzati nelle fasi preliminari di progettazione.
Esempio: Entity-Relationship (ER).

Possiamo dire che l'utente si interfaccia con lo schema logico di un database, dettato dal modello logico adottato. A livello fisico accade quanto definito nello schema interno, ovvero la parte di effettiva implementazione del DBMS.

Schema logico
Lo schema logico è la descrizione della base di dati nel modello logico, quindi la struttura di tabelle, ecc... L'utente che sviluppa il database si interfaccia con lo schema logico, usufruendo delle sue astrazioni.

Schema interno
Lo schema interno è l'implementazione dello schema logico, ed è noto solo a chi implementa il DBMS.

17 Linguaggi per Basi di Dati

I DBMS dispongono di vari linguaggi e interfacce per la definizione di schemi, modifica e lettura dei dati, e formmulazione di query. Possiamo avere:

- Linguaggi testuali interattivi (SQL)
- Comandi (SQL) immersi in un linguaggio ospite (Java, C++, ...)
- Interfaccie grafiche (Access)

Si può fare l'ulteriore distinzione:

- **Data definition language** (DDL): per la definizione di schemi (logici e fisici)
- **Data manipulation language** (DML): per l'interrogazione e l'aggiornamento di istanze di basi di dati.

18 Architettura a tre livelli per DBMS

Possiamo complicare le cose introducendo, oltre allo schema interno e lo schema logico, un'ulteriore schema, lo schema esterno, attraverso cui permetteremo agli utenti di interfacciarsi con la base di dati a livello ancora più alto (astratto).
Lo schema esterno immplementa fondamentalmente viste parziali (magari solo alcune tabelle) di database. Sarà in ogni caso importante stabilire:

- Indipendenza fisica: il livello logico e quello esterno sono indipendenti da quello fisico, come nel paradigma dell'astrazione procedurale, la realizzazione dello schema fisico può variare senza che debbano essere attuate modifiche dello schema logico.

- Indipendenza logica: il livello esterno è poi indipendente da quello logico, aggiunte o modifiche alle viste non richiedono modifiche al livello logico, e viceversa le modifiche dello schema logico lasciano inalterato lo schema esterno.

19 Attori

Possiamo adesso stabilire quali sono gli "attori" che implementeranno, lavoreranno su ed interagiranno con il DBMS:

- Progettisti e realizzatori di DBMS
- Progettisti della base di dati e suoi amministratori
- Progettisti e programmatori di applicazioni
- Utenti:
  - Utenti finali: eseguono applicazioni predefinite:
  - Utenti casuali: eseguono operazioni noon previste a priori, usando linguaggi interattivi (SQL)

20 Modello relazionale

Vale la pena riportare i 3 modelli logici storici, tra cui figura il modello relazionale che andremo a studiare:

gerarchico, reticolare, relazionale

Più recentemente si è poi diffuso il paradigma ad oggetti, basato su XML, anche detto NoSQL (viene principalmente usato su database di dimensioni particolarmente grandi).

I modelli gerarchici e reticolari non sono molto utilizzati per un particolare difetto: la gestione di relazioni fra dati non viene gestita in maniera efficiente (si usa il meccanismo dei riferimenti, che sono però fin troppo dipendenti dalla struttura fisica adottata). Nel modello relazionale, invece, tutto è basato sui valori, completamente slegati alle specificità della struttura fisica. Gli stessi riferimenti, o più propriamente relazioni, sono basati su valori condivisi fra più tabelle.

Tabelle

La tabella è l'entità fondamentale di un database relazionale. Come già visto

prima, una tabella contiene righe (record) e colonne (attributi), che rappresentano in un qualsiasi momento un istanza dello schema logico adottato.
Relazioni
Le relazioni del database relazionale si basano sul concetto matematico di relazione. Poniamo ad esempio due insiemi:

$$D_1 = (a, b)$$

$$D_2 = (x, y, z)$$

e il loro prodotto cartesiano:

$$D_1 \times D_2 = ((a, x), (a, y), (a, z), (b, x), ...)$$

una certa relazione R è:

$$R \subseteq D_1 \times D_2$$

sottoinsieme del loro prodotto cartesiano. Formalmente:
Dati n insiemi (anche non distinti) $D_1, D_2, ..., D_n$, e definito su di essi un prodotto cartesiano $D_1 \times D_2 \times ... \times D_n$, ovvero l'insieme di n-uple ordinate $(d_1, d_2, ..., d_n)$ con $d_1 \in D_1, d_2 \in D_2, ..., d_n \in D_n$, una relazione è un sottoinsieme del prodotto cartesiano fra gli insiemi.
Non esiste ordinamento fra le n-uple, e non esistono n-uple uguali. Gli insiemi $D_1, D_2, ..., D_n$ sono detti domini della relazione.

Struttura non posizionale
Dando un certo nome ai domini della relazione, possiamo passare da quella che è effettivamente una struttura posizionale (le n-uple sono ordinate), ad una struttura dove le posizioni dei domini (attributi) non sono più rilevanti.

Riferimenti fra relazioni
I riferimenti fra relazioni diverse sono rappresentati da valori dei domini che compaiono nelle n-uple.

Definizioni formali
Abbiamo adesso tutti gli strumenti necessari a definire formalmente il modello relazionale:

- **Schema di relazione**: simbolo R detto nome della relazione e un insieme di attributi $X = (A_1, ..., A_2)$ solitamente indicato con $R(X)$. A ciascun attributo X è associato un dominio. Gli attributi non possono essere relazioni.

- **Schema di base di dati**: un insieme di schemi di relazione con nomi diversi, solitamente indicato come $R = (R_1(X_1), ..., R_n(X_m))$.

N-uple
Una n-upla o tupla su insieme di attributi X è una funzione che associa a ciascun attributo $A \in X$ un elemento, o valore, nel dominio di A. Il simbolo $t[A]$ denota il valore della n-upla t sull'attributo A. Il simbolo $t[Y]$, con $Y \subseteq X$, denota il valore della n-upla t sull'insieme di attributi Y. N.B.: le n-uple non sono ordinate!

Definiamo allora le istanze di relazioni (e quindi di database):

- **Istanza di relazione**: su uno schema $R(X)$, un insieme r di n-uple su X
- **Istanza di base di dati**: su uno schema $R = (R_1(X_1), ..., R_n(X_m))$, un insieme di relazioni $r = (r_1, ..., r_2)$ dove ogni r_i è una relazione sullo schema $R_i(X_i)$

Graficamente, una n-upla è una riga della tabella, e un istanza l'insieme di tutte le sue righe.

Informazione incompleta
Il modello relazionale impone ai dati una struttura rigida, ma non è detto che i dati del mondo reale aderiscano sempre a questa struttura. Si introduce allora la parola chiave NULL, per evitare di usare valori particolari del dominio (0, ecc...), che potrebbero diventare significativi, o la cui gestione è comunque abbastanza complicata. Il valore di un certo attributo A, ovvero $t[A]$, potrà quindi appartenere al dominio D_A oppure essere il valore NULL. Si possono inoltre imporre determinate restrizioni sulla presenza dei valori nulli in una data relazione. Un valore NULL può modellizzare almeno 3 casi differenti:

- valore sconosciuto
- valore inesistente
- valore senza informazione

N.B.: Il DBMS non fa alcuna distinzione fra valori NULL!
La limitazione sui valori che possono o non possono essere NULL torna utile ad esempio nel caso della definizione di chiavi primarie di record, che devono essere fra di loro diverse ma comunque mai nulle.

21 Correttezza di basi di dati

Una base di dati può essere scorretta, ovvero immagazinare dati incompatibili con la realtà dei fatti. Ad esempio, sarebbe impensabile avere una base di

dati contenente voti relativi ad esami universitari, e trovare in tale base di dati la valutazione "27 e lode". La correttezza dei dati può essere assicurata grazie al sistema dei vincoli:

22 Vincoli di integrità

I vincoli di integrità sono sostanzialmente funzioni booleane (predicati) che associano alla istanza completa di basi dati un valore vero o falso. Se il vincolo di una base di dati restituisce vero, significa che le sue proprietà sono soddisfatte. Possiamo distinguere i vincoli in:

- **Vincoli intra-relazionali**: i vincoli intra-relazionali sono definiti rispetto ad una singola relazione. Possiamo portare gli esempi:
  - Vincolo di n-upla: può essere valutata su ciascuna n-upla indipendentemente dalle altre
  - Vincolo di dominio: vincolo di n-upla che coinvolge un solo attributo.
- **Vincoli inter-relazionali**: i vincoli inter-relazionali sono definiti rispetto a più relazioni diverse fra loro.

23 Chiavi

Una chiave è un insieme di attributi che identifica univocamente le n-uple. Formalmente: una **superchiave** è un insieme K di attributi di un insieme r se r non contiene due n-uple distinte t_1 e t_2 tali che $t_1[K] = t_2[K]$. Una chiave K è tale per un insieme r se è una superchiave minimale in R (cioè non contiene un'altra superchiave). N.B.: non per forza una chiave è unica.

Qualsiasi relazione ammette sempre almeno una chiave, in quanto una relazione contiene n-uple tutte diverse fra loro, ergo tutto l'insieme degli attributi forma sempre una chiave.
L'esistenza di una chiave garantisce l'accessibilità a ciascun dato della base di dati, e permette di correlare fra loro dati in relazoni diverse. Le chiavi formano inoltre un vincolo di integrità, detto appunto vincolo di chiave.
Nel caso una relazione abbia più di una chiave, ne scegliamo una in particolare, la chiave primaria (*primary key*). Le altre chiavi verranno allora dette chiavi candidate.

Chiavi e valori nulli
La presenza di valori nulli nelle chiavi può rappresentare un grande problema:

rende impossibile il riferimento da parte di altre relazioni e non permette l'identificazione univoca di ogni record. Va dunque evitato quando possibile.

Dipendenze Funzionali
I vincoli di chiave sono particolari tipo di vincoli, che fanno parte di una categoria può vasta: le **dipendenze funzionali**. Formalmente: dati due insiemi di attributi X e Y sulla relazione R, si dice che X determina Y, e si scrive $X \to Y$, se e solo se per ogni coppia di n-uple distinte t_1 e t_2, se $t_1[X] = t_2[X]$, allora $t_1[Y] = t_2[Y]$.

24 Integrità referenziali

Informazioni in relazioni diverse possono essere correlate attraverso valori comuni. Definiamo allora il concetto di chiave esterna (*foreign key*), ovvero un'insieme di attributi di una relazione che corrispondono ad una chiave di un'altra relazione.

Vincolo di integrità referenziale
Un vincolo di integrità referenziale fra gli attributi X di una relazione R_1 (chiave esterna di R_1) e un'altra relazione R_2 impone ai valori su X di R_1 di comparire come valori della chiave primaria di R_2. N.B.: in questo caso l'ordine dei valori è significativo.

Integrità referenziale e valori nulli
In presenza di valori nulli i vincoli possono essere resi meno restrittivi: scegliamo da R_1 solamente i valori di X diversi da NULL.

25 Operazioni di aggiornamento

Le operazioni di aggiornamento di una relazione consistono in sostanza di:

- Operazioni di inserimento, può violare:
  - vincoli intra-relazionali
  - integrità referenziale
- Operazione di cancellazione, può violare:
  - integrità referenziale
- Operazione di modifica (cancellazione + inserimento), di norma non viola nulla.

La correttezza di queste operazioni è assicurata dal DMBS, che reagirà alle violazioni dei vincoli attraverso azioni compensative.

Azioni compensative Nel caso si provi ad eliminare una n-upla causando violazionzi del vincolo di chiave, il comportamento standard potrebbe essere quello del rifiuto dell operazione. Si potrebbero sennò adottare altre azioni compensative, come ad esempio:

- Eliminazione in cascata
- Introduzione di valori nulli

26 Linguaggi per basi di dati

I linguaggi per basi di dati appartengono a 2 categorie distinte:

- Operazioni sullo schema: **Data Definition Language (DDL)**;
- Operazioni sui dati: **Data Manipulation Lanaguage (DML)**
  - Interrogazione (query)
  - Aggiornamento (update).

Interrogazioni di basi di dati
Un operazione di interrogazione è un operazione di lettura che richiede l'accessso a una o più tabelle. Per specificare interrogazioni si possono seguire due formalismi:

- **Modo dichiarativo**: si specificano le proprietà del risultato (che cosa)
- **Modo procedurale**: si specificano le modalità di generazione del risultato (come).

L'**algebra relazionale** permette di specificare delle interrogazioni secondo il modello procedurale: cioè elencando i passi "primitivi" necessari alla generazione di una risposta. Il **calcolo relazionale** invece ci permette di definire in modo dichiarativo quello che sarà il risultato dell'interrogazione. Come vedremo, sarà il calcolo relazionale a definire la semantica del linguaggio relazionale, perchè permette di fornire un'implementazione slegata dai dettagli procedurali.

27 Algebra relazionale

Un'algebra è una struttura matematica dotata di un'insieme di dati e una serie di operatori che manipolano suddetti dati. Nell'algebra relazionale abbiamo che:

- **Dati**: relazioni
- **Operatori**:
  - su relazioni
  - che producono relazioni
  - che possono essere composti.

  essi sono divisi fra:
  - **Operatori su insiemi**:
    unione, intersezione, differenza
  - **Operatori su relazioni**:
    ridenominazione, selezione, proiezione,
    join:
    - naturale, prodotto cartesiano, theta

Notazione dell'algebra relazionale
Specifichiamo adesso la notazione usata:

- $R, R_1, ...$ indicano nomi di relazionie
- $A, B, C, A_1...$ indicano nomi di attributo
- XY è un'abbreviazione di $X \cup Y$
- Una relazione con n-uple $t_1, t_2, ...$ è indicata con l'insieme $\{t_1, t_2, ...\}$
- $t_j[A_j]$ indica il valore della n-upla t_j sull'attributo A_j
- $t[X]$ indica l'n-upla ottenuta da considerando solo gli elementi di X.

Operatori su insiemi
Le relazioni sono insiemi di tuple, e non possono avere elementi duplicati (sarebbero altrimenti multiinsiemi). I risultati di operazioni fra relazioni sono a loro volta relazioni, ovvero insiemi di tuple. Gli operatori fra relazioni possono applcarsi solo e soltanto fra relazioni definite sullo stesso insieme

di attributi X, e il risultato sarà a sua volta definito sullo stesso insieme di attributi X.

Unione
L'operatore di unione fra due relazioni sull'insieme di attributi X comporta la formazione di una nuova relazione che ha tutte le n-uple delle relazioni unite. Eventuali n-uple identiche fra le relazioni appariranno nel risultato una volta sola. Il suo simbolo è $\cup$.

Intersezione
L'operatore di intersezione fra due relazioni sull'insieme di attributi X restituisce una nuova relazione che contiene soltanto gli elementi appartenenti sia alla prima che alla seconda relazione. Il suo simbolo è $\cap$.

Ðifferenza
L'operatore di differenza fra due relazioni sull'insieme di attributi X restituisce una nuova relazione contenente tutte le n-uple che appartengono alla prima relazione ma non alla seconda. L'operatore di differenza è l'unico fra gli operatori insiemistici a non essere commutativo. Il suo simbolo è il meno $(-)$, o più propriamente $\setminus$.

Notiamo che l'unione è l'unico operatore in grado di creare relazioni con un numero di elementi maggiore di quello degli operandi. Questa caratteristica tornerà poi utile nel calcolo relazionale.

Descriviamo adesso gli operatori non insiemistici (sulle relazioni):
Ridenominazione
L'operatore di ridenominazione è un'operatore monadico che modifica lo schema dell'operando, lasciando inalterata l'istanza. Si scrive, data una relazione R, come:

$$\rho_{B_1 B_2 \ldots} \leftarrow_{A_1 A_2 \ldots (R)}$$

I pedici a sinistra e a destra della freccia sono insiemi di attributi. Avremo che:

- L'attributo A_1 viene sostituito dall'attributo B_1
- L'attributo A_2 viene sostituito dall'attributo B_2
- ecc...

Selezione
L'operatore di selezione è un operatore monadico che produce un un risultato con lo stesso schema dell'operando, e un sottoinsieme di n-uple che rispettano una determinata condizione. Si scrive come:

$$\sigma_F(R)$$

sulla relazione R, dove F è un'espressione Booleana (predicato) ottenuta componendo con gli operatori logici AND, OR e NOT le condizioni atomiche:

- $A \star B$, dove A e B sono attributi di X con domini compatibili e $\star$ un operatore di confronto.
- $A \star k$, dove A è un attributo di X e k una costante con dominio compatibile con A e $\star$ sempre un operatore di confronto.

Notiamo che la condizione atomica è vera solo per valori non nulli in ogni attributo.

Selezione valori NULL

Per riferirsi ai valori NULL occorre usare le apposite condizioni: IS NULL e IS NOT NULL.

Proiezione

La proiezione è un'operatore monadico che produce un sottoinsieme degli attributi dell'operando contenente tutte le sue n-uple ristrette soltanto ad alcuni attributi. Si scrive, data una relazione $R(X)$ e un insieme di attributi $Y \subseteq X$, come:

$$\pi_Y(R)$$

Il risultato è una relazione su Y che contiene l'insieme delle n-uple di R ristrette ai soli attributi di Y. Notare che, di nuovo, non possono esserci righe ripetute. Ogni n-upla ripetuta nell'insieme risultato verrà inserita una volta sola.

Cardinalità di proiezioni

Una proiezione contiene al massimo tante n-uple quante ne contiene l'operando. Inolre, se l'insieme di attributi X è superchiave della relazione R, allora la proiezione $\pi_X(R)$ avrà tante n-uple quante ne ha l'operando.

Join

Non possiamo, usando gli operatori di selezione e proiezione, estrarre e combinare informazioni da più relazioni diverse fra loro, e non possiamo nemmeno combinare informazioni presenti in n-uple diverse di una stessa relazione. Avremo allora bisogno di un'ulteriore serie di operazioni, le cosiddette operazioni di join. I join ci permettono di unire sulla stessa riga più righe di relazioni diverse. Vediamo nel dettaglio:

Join naturale

Il join naturale è un'operatore con due operandi (generalizzabile), che produce come risultato l'unione degli attributi degli operandi, contente le n-uple costruite ciascuna a partire da un n-upla di ognuno degli operandi. Formalmente: Date due relazioni R_1X_1 e R_2X_2, il loro join naturale si scrive

come:

$$R_1 \bowtie R_2$$

Il risultato è una relazione $R(X_1 \cup X_2)$ definita come:

$$R(X_1 \cup X_2) = R_1(X_1) \bowtie R_2(X_2) = \{t | \exists t_1 \in R_1, \quad t_2 \in R_2$$

$$\text{con} \quad t[X_1] = t_1, \quad t[X_2] = t_2\}$$

N.B.: è perfettamente plausibile voler fare il join naturale tra due relazioni senza attributi in comune, e in questo caso si ottiene il prodotto cartesiano fra le due.

Cardinalità del join

Il join di due relazioni R_1 e R_2 contiene un numero di n-uple compreso fra 0 e il prodotto di $|R_1|$ e $|R_2|$. Se il join coinvolge una chiave di R_2 allora il numero di n-uple è compreso fra 0 e $|R_1|$. Se il join coinvolge una chiave di R_2 e un vincolo di integrità referenziale allora il numero di n-uple è uguale a $|R_1|$. Più formalmente:
Il join di $R_1(A, B)$ e $R_2(B, C)$ contiene un un numero di n-uple:

$$0 \leq |R_1 \bowtie R_2| \leq |R_1| \times |R_2|$$

Se B è una chiava di R_2 allora il numero di n-uple è

$$0 \leq |R_1 \bowtie R_2| \leq |R_1|$$

Se B è una chiave di R_2 ed esiste un vincolo di integrità referenziale fra B (in R_1) e R_2 allora il numero delle n-uple è:

$$|R_1 \bowtie R_2| = |R_1|$$

Una problematica del join è il fatto che le n-uple che non contribuiscono al risultato ("che non fanno join") restano tagliate fuori. Introduciamo allora altri tipi di operatori di join, i cosiddetti:

Join esterno

Il join esterno estende, con valori NULL, le n-uple che verrebbero altrimenti scartate dal join (interno). Ne esistono tre versioni:

- **Sinistro**: mantiene tutte le n-uple del primo operando ($\bowtie_{LEFT}$);
- **Destro**: mantiene tutte le n-uple del secondo operando ($\bowtie_{DESTRO}$);
- **Completo**: mantiene tutte le n-uple di entrambi gli operandi ($\bowtie_{FULL}$).

Join e proiezioni
Date due relazioni $R_1(X_1)$ e $R_2(X_2)$:

$$\pi_{X_1}(R_1 \bowtie R_2) \subseteq R_1$$

Date una relazione $R(X)$ con $X = X_1 \cup X_2$

$$(\pi_{X_1}(R) \bowtie \pi_{X_2}(R)) \supseteq R$$

Prodotto cartesiano
Come detto prima, date due relazioni $R_1(X_1)$ e $R_2(X_2)$, senza attributi in comune, cioè con $X_1 \cap X_2 = 0$, la definizione di join naturale e comunque senso e restituisce il prodotto cartesiano delle due relazioni:

$$R = R_1 \bowtie R_2 = R_1 \times R_2$$

la cardinalità del prodotto cartesiano è uguale al prodotto delle cardinalità delle due relazioni.

Theta join
Nella pratica, il prodotto cartesiano ha senso quasi solamente se seguito da una selezione $\sigma_F(R_1 \times R_2)$. Questa combinazione prende il nome di theta join (è un operatore derivato) ed è indicato come:

$$R_1 \bowtie_F R_2$$

dove F è un certo rpedicato. Spesso F è una congiunzione di atomi di confronto $A_1 \sigma A_2$ dove σ è un operatore di confronto ($<=$, $<$, $=$, ecc...) e A_1 e A_2 attributi di relazioni diverse. Quando l'operatore di confronto è l'uguaglianza parliamo di equi-join.
Tutti gli operatori visti finora, ovvero gli operatori insiemistici e quelli relazionali, bastano a realizzare qualsiasi possibile interrogazione. Ogni altro operatore sarà un' operatore derivato dei 5 operatori relazionali e i 3 poeratori insiemistici. Un particolare operatore derivato è:

28 Divisione

Dati due insiemi di attributi disgiunti X_1 e X_2, una relazione r sulla loro unione e una relazione r_2 su X_2, la divisione $r \div r_2$ è una relazione su X_1 che contiene le n-uple ottenute come "proiezione" di n-uple di r che si combinano con tutte le n-uple di r_2 per formare n-uple di r, in una sorta di prodotto cartesiano inverso. In simboli:

$$r \div r_2 = \{t_1[X_1] \quad | \quad \forall t_2 \in r_2 \quad \exists t \in r : t[X_1] = t_1, \quad t[X_2] = t_2\}$$

possiamo dimostrare che è un operatore derivato definendolo come composizione di operatori fondamentali, in questo modo:

$$r \div r_2 = \pi_{X_1}(r) - \pi_{X_1}((\pi_{X_1}(r) \times r_2) - r)$$

Ovvero:

- $\pi_{X_1}(r) \times r_2$ contiene tutte le n-uple di $\pi_{X_1}(r)$ "estese" con tutti i possibili valori di r_2.
- $(\pi_{X_1}(r) \times r_2) - r$ contiene le "estensioni" di $\pi_{X_1}(r)$ che non compaiono in r.
- $\pi_{X_1}((\pi_{X_1}(r) \times r_2) - r)$ contiene le n-uple di $\pi_{X_1}(r)$ per le quali un qualche completamento con r_2 non compare in r.
- Togliendo queste ultime n-uple a $\pi_{X_1}(r)$ otteniamo tutte le n-uple di $\pi_{X_1}(r)$ che si combinano con tutte le n-uple di r_2.

29 Chiusura transitiva

Poniamoci il problema di dover trovare, in un'opportuna tabella supervisione formata da matricole di impiegati e supervisori di tali impiegati, le matricole di tutti i supervisori di un dato impiegato (ammettendo che i supervisori siano impiegati e possano a loro volta avere supervisori). Tale richiesta sarebbe perfettamente valida, ma impossibile da esprimere attraverso gli operatori dell'algebra relazionale.
Nell'algebra relazionale non esiste la possibilità di esprimere interrogazioni che calcolino la chiusura transitiva di una relazione arbitraria. Tale operazione potrebbe infatti richiedere un numero infinito di di join (join illimitato).

30 Espressioni equivalenti

Due espressioni sono equivalenti se producono lo stesso risultato qualunque sia l'istanza fornitagli. In questo caso, sarà opportuno scegliere espressioni di costo minore, dove il loro "costo" è determinato dalle dimensioni delle istanze intermedie che la loro esecuzione genera. Vediamo alcune equivalenze fondamentali:

- **Atomizzazione delle selezioni**: una congiunzione di selezioni può essere sostituita da una sequenza di selezioni atomiche:

$$\sigma_{F_1 \wedge F_2}(E) = \sigma_{F_1}(\sigma F_2(E))$$

- **Idempotenza delle proiezoni**: una proiezione può essere trasformata in una sequenza di proiezioni:

$$\pi_X(E) = \pi_X(\pi_{XY}(E))$$

- **Push selections down**: se una condizione F coinvolge solo attributi dell'espressione E_2:

$$\sigma_F(E_1 \bowtie E_2) = E_1 \bowtie \sigma_F(E_2)$$

- **Push projections down**: se un'espressione E_1 ha attributi X_1, un'espressione E_2 ha attributi X_2, $Y_2 \subseteq X_2$ e gli attributi $X_2 - Y_2$ non sono coinvolti nel join ($X_1 \cap X_2 \subseteq Y_2$):

$$\pi_{X_1Y_2}(E_1 \bowtie E_2) = E_1 \bowtie \pi_{Y_2}(E_2)$$

31 Ottimizzazione delle interrogazioni

Un modulo presente nel DBMS è il **query processor** (od ottimizzatore). L'ottimizzatore si occupa di scegliere la strategia realizzativa a partire dall'istruzione in linguaggio dichiarativo di alto livello, tenendo conto del costo di implementazioni diverse. Le fasi di eseecuzione di una query saranno:

- Analisi lessicale, sintattica e semantica della query in linguaggio di alto livello (SQL). Al termine di questa analisi, la query verrà tradotta in un'espressione dell'algebra relazionale.

- Ottimizzazione algebraica: a questo punto verranno calcolate una o più nuove espressioni algebriche equivalenti a quella di partenza, sfruttando le equivalenza sopra descritte.

- Ottimizzazoine basata sui costi: fra le alternative calcolate prima, viene selezionata la più efficiente, che viene poi utilizzata per effettuare l'interrogazione effettiva sulla base di dati.

Nella prima e l'ultima di queste fasi, il DMBS interagisce con un componente detto catalogo, che contiene informazioni sugli schemi contenuti nel database, la cardinalità delle loro istanze, ecc..

Profili delle relazioni

Tra le informazioni quantitative memorizzate nel catalogo troviamo:

- Cardinalità di ciascuna relazione;

- Dimensioni delle n-uple;
- Dimensioni dei valori;
- Numero di valori distinti degli attributi;
- Valore minimo e massimo degli attributi.

Queste informazioni vengono usate nella fase di ottimizzazione basata sui costi per stimare le dimensioni dei risultati intermedi di più espressioni algebriche alternative generate dall'ottimizzazione algebrica.

Ottimizzazione algebrica
In verità, il termine ottimizzazione non è completamente accurato: il processo utilizza infatti delle euristiche per trovare risultati migliori. Si basa sul concetto di equivalenza per trovare query che restituiscono lo stesso risultato generando dimensioni d'istanza intermedie minori. Ad esempio, un'ottimizzazione tipica è quella di eseguire selezioni e proiezioni il più presto possibile, ovvero le cosiddette "push selections down" e "push projections down".

32 Grafi

Un grafo $G = (V, E)$ consiste in un insieme V di vertici (o nodi) e un insieme E di coppie di vertici, detti archi. Ogni arco ovviamente connetter fra loro due vertici. Possiamo allora distinguere:

- **Grafi orientati**: detti anche grafi diretti, dove ogni arco è orientato è rappresenta relazioni ordinate fra oggetti;
- **Grafi non orientati**: detti anche grafi indiretti, dove ogni arco non è orientato e rappresenta relazioni simmetriche fra oggetti.

Sui grafi si possono definire **cammini** da un vertice x ad un vertice y. Formalmente, un cammino è:

$$(v_0, ..., v_k) \quad \text{di} \quad V, \quad v_0 = x, \quad v_k = y \quad | \quad 1 \leq i \leq k : (v_{i-1}, v_i) \in E$$

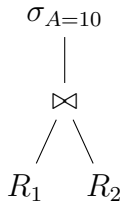
Un cammino che torna da dove parte, ovvero dove $(v_0, ..., v_k) : v_0 = v_k$, è detto ciclico. Si dice che un grafo diretto privo di cicli è **aciclico**.

Alberi
Un grafo non orientato si dice connesso se esiste un cammino fra ogni coppia di vertici. Un'albero è un grafo non orientato nel quale due vertici qualsiasi sono connessi da uno e un solo cammino.

Un'interrogazione può essere rappresentata da un'albero, dove le foglie (i nodi finali) sono dati (relazioni, file, ecc...), e i nodi intermedi sono operatori (operatori algebrici, poi effettivi operatori di accesso ai dati). Ad esempio, l'albero corrispondente all'espressione:

$$\sigma_{A=10}(R_1 \bowtie R_2)$$

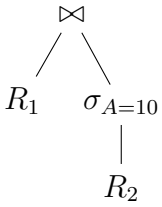
sarà:



La stessa interrogazione dopo il push down della selezione $\sigma_{A=10}$ sarà allora:

$$R_1 \bowtie \sigma A = 10(R_2)$$

con relativo albero:



Procedura euristica di ottimizzazione
La procedura di ottimizzazione consiste quindi nel:

- Decomporre le selezioni congiuntive in successive selezioni atomiche;
- Anticipare il più possibile le selezioni;
- In una sequenza di selezioni, anticipare le più selettive;
- Combinare prodotti cartesiani e selezioni per formare join;
- Anticipare il più possibile le proiezioni.

33 Relazioni Derivate

Una **relazione di base** è una relazione il cui contenuto è autonomo. Una **relazione derivata** è una relazione il cui contenuto è funzione di altre relazioni. Si possono cosi avere rappresentazioni diverse per gli stessi dati, definite mediante interrogazioni, che possono rivolgersi a relazioni di base come ad altre relazioni derivate. Esistono due tipi di relazioni derivate:

- **Viste materializzate**, dove i risultati sono precalcolati;
- **Viste virtuali**, o più semplicemente viste. Sono le più comuni.

Viste materializzate
Sono relazioni derivate memorizzate nella base di dati. Il loro vantaggio è che sono precalcolate, e quindi immediatamente disponibili. I loro svantaggi sono che:

- Sono ridondanti;
- Appesantiscono gli aggiornamenti;
- Sono raramente supportate dai DBMS.

Viste virtuali
Relazioni derivate non memorizzate nella base di dati, e quindi ricalcolate ad ogni accesso. Sono supportate da tutti i DBMS.

Interrogazioni su viste
Le interrogazioni su viste sono eseguite sostituendo alla vista la sua definizione. Le viste possono semplificare la scrittura di interrogazioni, espressioni complesse, e sotto-espressioni ripetute. L'uso delle viste virtuali inoltre non influisce sull'efficienza delle interrogazioni. Ad esempio, supponiamo di avere le seguenti relazioni:

$$R_1(ABC), R_2(DEF), R_3(GH)$$

e di definire una vista R:

$$R = \sigma_{A>D}(R_1 \bowtie R_2)$$

Un'interrogazione potrà a questo punto essere definita:

- Senza vista:
$$\sigma_{B=G}(\sigma_{A>D}(R_1 \bowtie R_2) \bowtie R_3)$$

item Con vista:
$$\sigma_{B=G}(R \bowtie R_3)$$

Viste e aggiornamenti
Aggiornare una vista significa modificare le relazioni di base in modo che la vista ricalcolata rispecchi l'aggiornamento. L'aggiornamento sulle relazioni di base dovrebbe essere associato univocamente a quello specificato sulla vista, ma questo non è sempre vero. Per questo motivo ben pochi aggiornamenti sono ammissibili sulle viste.

34 Calcolo relazionale

Il calcolo relazionale è una famiglia di linguaggi dichiaratvi basati sul calcolo dei predicati del primo ordine. Ne esistono diverse versioni:

- Calcolo relazionale sui domini, in breve calcolo dei domini;
- Calcolo su n-uple con dichiarazione di range, in breve calcolo delle tuple, usato in SQL.

Assunzioni
I simboli di predicato corrispondono alle relazioni presenti nelle basi di dati, più alcuni predicati standard come l'uguaglianza e la disuguaglianza. Non compaiono simboli di funzione. Nel calcolo relazionale vengono utilizzate prevalentemente formule aperte, cioè formule con variabili libere, il cui valore di verità dipende dai valori assegnati alle variabili libere. Il risultato di un'interrogazione (formula aperta) è costituito da tutte le tuple di valori che sostituiti alle variabili libere, la rendono vera. In coerenza con quanto fatto in algebra relazionale, useremo una notazione non posizionale (attributi con nome, etichettati).

Calcolo dei domini
Vediamo innanzitutto la sintassi: nel calcolo dei domini un'espressone ha forma:

$$\{A_1 : x_1, ..., A_k : x_k \mid f\}$$

dove:

- $A_1, ..., A_k$ sono attributi distinti (che possono anche non comparire nello schema della base di dati).
- $x_1, ..., x_K$ sono variabili (che assumiamo essere distinte, non è necessario).
- $A_1 : x_1, ..., A_k : x_k$ è la *target list* (lista degli obiettivi) e descrive il risultato.

- f è una formula costruita a partire da formule atomiche utilizzando eventualmente i connettivi booleani e i quantificatori logici $\forall x$, $\exists x$, con x variabile (logica del primo ordine).

Il risultato di un'espressione nel calcolo dei domini è una relazione su $A_1, ..., A_k$, contenente n-uple in $x_1, ..., x_2$ che rendono vera la formula f sull'istanza di basi di dati considerata. F rispetto all'istanza

Formule atomiche del calcolo dei domini
Il predicato f è costituito da formule atomiche. Una formula atomica è definita come una delle tre forme:

- **Schemi di relazione**

  $$R(A_1 : x_1..., A_p : x_p)$$

  dove $R(A_1, ..., A_P)$ è uno schema di relazione.

- **Operatori fra variabili**

  $$x_i \; OP \; x_j$$

  dove x e y sono variabili e OP è un operatore di confronto (uguale, non uguale, maggiore, minore e varianti strette).

- **Operatori fra variabili e costanti**

  $$x_i \; OP \; c, \quad c \; OP \; x_i$$

  dove c è una costante nel dominio A_i di x_i.

Definiamo allora alcune proprietà delle formule:

- Le formule atomiche sono formule.
- Se f è una formula, allora anche $\neg f$ è una formula.
- Se f_1 e f_2 sono formule, allora anche $f_1 \wedge f_2$ e $f_1 \vee f_2$ sono formule.
- Se f è una formula e x una variabile, allora anche $\exists x(f)$ e $\forall x(f)$ sono formule, dove $\exists$ and $\forall$ sono qualificatori (rispettivamente esistenziale e universale).
- Per convenienza, si raggruppano le variabili usate da quantificatori simili: ad esempio, $\exists x(\exists y(f))$ si può scrivere come $\exists x, y(f)$.

Valore di verità
L'inclusione di una data n-upla o meno da un'espressione è determinato dal valore di verità della sua formula. Il valore di verità di una formula è così definito (sulle formule atomiche):

- Una formula atomica $R(A_1 : x_1, ..., A_p : x_p)$ è vera sui valori $x_1, ..., x_p$ che costituiscono una n-upla valida di R.
- Una formula atomica $x\ \theta\ y$ o $x\ \theta\ c$ (con c costante) è vera sui valori x, y che soddisfano la condizione θ.
- Il valore di verità degli operatori logici (congiunzione $\wedge$, disgiunzione $\vee$, negazione $\not$)è definito nel modo usuale.
- Una formula della forma $\exists x(f)$ è vera se esiste almeno un elemnto del dominio di x che rende vera f. Analogamente, una formula della forma $\forall x(f)$ è vera se tutti gli elementi del dominio di x rendono f.

Interpretazione logica del calcolo dei domini
Un'espressione del calcolo sui domini $\{A_1 : x_1, ..., A_k : x_k | f\}$ può essere sostanzialmente interpretata come una formula logica del tipo:

$$\{x_1, ..., x_k | f(x_1, ..., x_l)\}$$

dove $x_1, ..., x_k$ sono variabili o costanti, e $f(x_1, ..., x_k)$ è un predicato (vero o falso) che determina l'appartenenza o meno della n-upla di variabili al risultato dell'espressione.

Vantaggi e svantaggi del calcolo dei domini
Il vantaggio principale del calcolo dei domini è la dichiaritività: si *dichiarano* le proprietà del risultato desiderato, senza specificare i passaggi necessari a calcolarli. I difetti stanno invece nella grande verbosità (non è permesso fare proiezioni su stadi intermedi, ergo siamo costretti a portarci dietro tutti gli attributi nell'enunciato della formula). Inoltre, il calcolo dei domini è dipendente dal dominio (ovvero non tutte le espressioni sintatticamente valide nel calcolo dei domini sono indipendenti dal dominio):

Indipendenza dal dominio
Si dice che un'espressione di un linguaggio d'interrogazione è indipendente dal dominio quando il suo risultato non varia al variare del dominio rispetto alla quale l'espressione su cui viene valutata, salvo ovviamente i valori presenti nell'istanza e nell'espressione. A questo punto un linguaggio si dice indipendente dal dominio se tutte le sue espressioni sono tali. L'algebra relazionale è indipendente dal dominio: i risultati vengono costruiti sulla base di

operazioni svolte sulle relazioni presenti nella base di dati senza mai fare riferimento ai domini dei valori che compongono le n-uple delle istanze. In altre parole, tutti i valori compaiono in istanze effettive di relazioni della base di dati o nella formulazione dell'espressione. Il calcolo dei domini, al contrario, è dipendente dal dominio. Si può ad esempio scrivere, senza violare nessuna regola sintattica:

$$\{A : x, B : y | R(A : x) \wedge y = y\}$$

nel risultato, il valore x dell'attributo A è legato alla relazione R, mentre il valore y può essere qualsiasi valore del dominio dell'attributo B, in quanto $\forall B : y | y = y$. Al variare del dominio di B, il risultato dell'espressione cambia (l'espressione è dipendente dal dominio!). Fosse stato B infinito, l'espressione avrebbe restituito un insieme infinito. Un'altro esempio può essere:

$$\{A : x | \neg R(A : x)\}$$

Tale espressione ottiene esattamente il complemento dei valori x sull'attributo A contenuti nella relazione R, ovvero tutti i valori del dominio di A meno quelli in R.
Per ovviare ai problemi di dipendenza dal dominio presentati dal calcolo dei domini è stato introdotto il calcolo delle tuple.

Calcolo delle tuple
Il calcolo delle tuple risolve i problemi di verbosità del calcolo dei domini, utilizzando variabili per denotare tuple anzichè singoli valori legati ad attributi. Ogni relazione coinvolta presenta quindi una variabile, che rappresenta ogni possibile tupla di quella relazione. Per accedere ai singoli attributi di una tupla occorre quindi associare una struttura ad ogni variabile. Le espressioni nel calcolo delle tuple hanno forma:

$$\{T | L | f\}$$

dove:

- T è una **target list** (obiettivi dell'interrogazione);
- L è una **range list**;
- f è una formula.

Target list
Stabiliamo che:

- x è una variabile;

- X è un insieme di attributi di una relazione;
- Y e Z sono sottoinsiemi di attributi di X di pari lunghezza.

A questo punto, possiamo dire che T è una lista di elementi del tipo:

- $Y : x.Z$, Denotiamo con Y solo gli attributi Z della variabile (tupla) x, che assumerà valori definiti in L (*range list*).
- $x.Z \equiv Z : x.Z$ vogliamo solo gli attributi Z della variabile x, che assumerà valori definiti in L, ma non li ridenominiamo.
- $x.* \equiv X : x.X$ Prendiamo tutti gli attributi della variabile x, che assumerà valori definiti in L.

Range list
La *range list* L è una lista che contiene, senza ripetizioni, tutte le variabili della *target list*, con associata la relazione da cui viene prelevata ogni variabile:

$$L \equiv x_1(R_1), ..., x_k(R_k)$$

dove x_i è una variabile e R_i è una relazione. L è una dichiarazione di range, ovvero specifica l'insieme dei valori che possono essere assegnati alle variabili. Non occorrono, come occorrevano nel calcolo dei domini, condizioni atomiche che vincolano una tupla ad appartenere ad una relazione $(A_1 : x_1, ..., A_n : x_n)$.

Formule atomiche del calcolo delle tuple
Possiamo definire, come avevamo fatto per il calcolo dei domini, 2 formule atomiche (manca la condizione di vincolo d'appartenenza alla relazione):

- **Operatori fra variabili**

  $$x_i.A_1 \; OP \; x_j.A_j$$

  dove x e y sono variabili, A_1 e A_j attributi di tali variabili, e OP è un operatore di confronto (uguale, non uguale, maggiore, minore e varianti strette).

- **Operatori fra variabili e costanti**

  $$x_i.A_i \; OP \; c, \quad c \; OP \; x_i.A_i$$

  dove c è una costante nel dominio dell'attributo A_i di x_i.

Possiamo quindi definire alcune proprietà delle formule, quasi del tutto analoghe a quelle definite per il calcolo dei domini:

- Le formule atomiche sono formule.
- Se f è una formula, allora anche $\neg f$ è una formula.
- Se f_1 e f_2 sono formule, allora anche $f_1 \wedge f_2$ e $f_1 \vee f_2$ sono formule.
- Se f è una formula e x una variabile che indica una n-upla sulla relazione R, allora anche $\exists x R(f)$ e $\forall x R(f)$ sono formule, dove $\exists$ and $\forall$ sono qualificatori (rispettivamente esistenziale e universale). Notiamo che anche i quantificatori contengono adesso delle dichiarazioni di range(*range list*). La notazione $\exists(R)(f)$ significa "esiste nella relazione R una n-upla x che soddisfa la formula f".

Vantaggi e svantaggi del calcolo delle tuple Il vantaggio principale del calcolo delle tuple è la minore verbosità rispetto al calcolo dei domini. Svantaggiosa è invece l'effettiva impossibilità di esprimere alcune interrogazioni, in particolare l'unione:

$$R_1(AB) \cup R_2(AB)$$

Questo perchè ogni variabile nel risultato ha un solo range, mentre l'unione comporta la derivazione di n-uple da più relazioni distinte. Per questo motivo linguaggi quali l'SQL, definiscono un operatore esplicito di unione (l'UNION). Intersezione e differenza sono invece esprimibili.

Calcolo e algebra
Calcolo e algebra sono sostanzialmente equivalenti, ovvero ogni espressione dell'algebra (indipendente dal domino!) ha un equivalente nel calcolo e viceversa. Esistono però alcune interrogazioni non esprimibili:

- **Calcolo dei valori derivati:** abbiamo la possbilita di estrarre valori, mai di generarne di nuovi.
- **Ricorsività:** le interrogazioni non hanno la possibilità di esprimere ricorsività, come avevamo visto nell'esempio della chiusura transitiva (riguardante riferimenti potenzialmente infiniti fra relazioni).

35 Modellazione e progettazione concettuale

La definizione di schemi adeguati per le basi di dati richiede metodologie precise per la modellazione accurata della realtà che ci interessa. Bisogna tenere a mente che:

- Non conviene concetrarsi subito sui dettagli;

- Conviene invece stabilire subito interdipendenze fra relazioni;
- Il modello relazionale sarà rigido una volta ultimato.

In generale si può dire che esiste sempre (e spesso è unico) uno schema che modella accuratamente e nel modo più semplice possibile una certa realtà di interesse. La progettazione di basi di dati è solo una fase delloo sviluppo di un sistema informativo (vedere a riguardo le prime lezioni di teoria). Bisogna quindi tenere conto di:

- **Ciclo di vita** *(lifecycle)* del sistema informativo, ovvero l'insieme delle attivita svolte da analisti, progettisti e utenti nello sviluppo e nell'uso del sistema informativo. Questa attività è iterativa, e quindi ciclica.

I passi ciclo di vita dovranno essere ben definiti attraverso linguaggi e modelli prestabiliti. Per le basi di dati, in particolare, conviene adottare modelli di facile utilizzo, che consentano la decomposizione delle attività in fasi (e/o livelli) distinti, e di utilizzare strategie e criteri di scelta nei vari passaggi.

Modello a cascata

Il modello a cascata (*waterfall model*) le fasi sono sequenzialmente ordinate, etichettate, e non ripetibili. In ordine, esse sono le seguenti:

1. **Studio di fattibilità**: definizione di costi e priorità della produzione;
2. **Raccolta e analisi dei requisiti**: studio delle proprietà del sistema che andranno implementate;
3. **Progettazione**: progettazione di strutture dati e funzioni;
4. **Realizzazione**: implementazione effettiva del codice;
5. **Validazione e collaudo**: sperimentazione del prodotto;
6. **Funzionamento**: il sistema diventa opertaivo in produzione (*shipping*).

Vediamo alcune fasi nel dettaglio.

36 Raccolta e analisi dei requisti

Questa fase può essere, a sua volta, divisa in due sotto-fasi:

- **Acquisizione dei requisiti**: il reperimento dei requisiti è un'attività non standardizzata. Esistono più modalità:

  - Direttamente dagli utenti:
    * Interviste, focus group, recensioni, ecc...
    * Documentazioni apposite;
  - Attraverso documentazioni preesistenti:
    * Normative (legislazioni, regolamenti di settore);
    * Regolamenti interni, procedure aziendali;
    * Realizzazioni preesistenti.

  Esistono linguaggi per definire requisiti (*UML*).

- **Analisi dei requisiti**: si analizzano i requisiti raccolti, spesso nella prospettiva di successive acquisizioni.

Interazione con gli utenti
Le problematiche dell'interazione con gli utenti possono essere:

- Utenti diversi danno risposte diverse;
- Utenti a livello più alto hanno spesso una visione più ampia ma meno dettagliata;
- Spesso l'acquisizione di requisiti avviene per raffinazione.

Conviene quindi:

- Effettuare spesso verifiche di comprensione e coerenza;
- Verificare anche attraverso esempi (sopratutto nei casi limite);
- Richiedere definizioni e classificazioni chiare e specifiche;
- Separare gli aspetti essenziali da quelli marginali (*ranking*).

Interazione con gli utenti tramite documentazione
E' opportuno seguire alcune linee guide generali, assicurando:

- Standardizzazione della struttura delle frasi;
- Separazione delle frasi riguardanti dati da quelle riguardanti funzioni;
- Organizzazione di termini e concetti:
  - Unificazione di termini (eliminazione di sinonimi);
  - Esplicitazione del riferimento fra termini;
- Organizzazione delle frasi per concetti.

37 Progettazione

La progettazione è una fase del ciclo di vita. Per un sistema software la progettazione si divide effettivamente in:

- Progettazione dei dati;
- Progettazione delle applicazioni.

Progettazione per astrazione
Come in tutte le applicazioni informatiche, abbiamo visto che è necessario progettare per livelli successivi di astrazione:

- **Livello concettuale**: esprime i requisiti di un sistema in una descrizione adatta all'analisi da punti di vista esterni
- **Livello logico**: evidenzia l'organizzazione dei dati dal punto di vista del loro contenuto informativo, descrivendo la struttura dei record e le loro interdipendenze.
- **Livello fisico**: si concentra sulla base di dati vista come un insieme di blocchi fisici sul disco, e riguarda quindi l'allocazione dei dati e le modalità di memorizzazione.

Con riferimento a quanto detto sugli schemi logici avremo quindi che la progettazione si divide in:

- **Progettazione concettuale**, parte dai requisiti individuati della base e produce uno schema concettuale;
- **Progettazione logica**, parte dallo schema concettuale e produce uno schema logico;
- **Progettazione fisica**, parta dallo schema logico e produce lo schema fisico finale.

Come lo era stato il modello a cascata, la progettazione per astrazione è composta da fasi sequenzialmente ordinate che vanno eseguite strettamente in ordine.

Modello dei dati
Il modello dei dati è l'insieme dei costrutti utilizzati per organizzare i dati di interesse e definirne la dinamica. Componente fondamentale del modello sono i meccanismi di strutturazione (costruttori di tipi). Come per i linguaggi di programmazione comuni, esistono meccanismi che permettono di definire

nuovi tipi. Ogni modello dei dati prevede alcuni costruttori: ad esempio il modello relazionale prevede un costruttore relazionale che permette di definire insiemi di record omogenei. Per riassumere in breve, in ogni base di dati si ha:

- Lo **schema**, invariante nel tempo, che ne descrive la struttura. Si notino inoltre le **intestazioni** delle tabelle (previste nel modello relazionale).
- L'**istanza**, i valori effettivi assunti dalla base di dati in un dato momento. Nel modello relazionale rappresenta il **corpo** di ciascuna tabella.

38 Modello concettuale Entity-Relationship

Il modello concettuale che utilizzeremo sarà quello **Entity-Relationship (ER)**. Si noti che la parola *relationship*, sebbene abbia lo stesso significato in lingua inglese, non si riferisce al concetto matematico di relazione che sta alla base del modello relazionale. Per questo motivo useremo sempre il termine inglese per descrivere le relazioni del modello entity-relationship, in modo da distinguerle dalle relazioni del modello relazionale. Il modello entity-relationship viene sviluppato da P.P. Chen nel 1976, ed è oggi una delle metodologie più affermate nel campo della progettazione dei sistemi informatici, anche se in un'accezione leggermente diversa da quella in cui era stato concepito inizialmmente.
I suoi costrutti base sono:

- Entità
- Relationship
- Attributi

Entità
Un'entità è una classe di oggetti dell'applicazione d'interesse con proprietà comuni e esistenza autonoma. Un'occorrenza (o istanza) di entità è un elemento della classe (un'elemento, non i dati ad esso legati!). Ogni entità deve avere un nome che la identifica univocamente nello schema. Graficamente è rappresentato da una scatola.

Relationship
Una relationship è un legame logico fra due o più entita, rilevante nell'applicazione d'interesse. Può essere chiamata anche relazione (vedi sopra), correlazione o associazione. Ogni relationship, come per le entità, ha un nome che la identifica univocamente nello schema. Graficamente è rappresentata

da una losanga.
Vediamo allora di definire il concetto di occorrenza di relationship, più complesso di quello di occorrenza di entità:

- Un'occorrenza di **relationship binaria** è una coppia di occorrenza di entità, una per ciascuna entità coinvolta.
- Una occorrenza di una **relationship n-aria** è una n-upla di occorenze di entità, una per ciascuna delle n entità coinvolte.

Nell'ambito di una relationship non ci possono essere occorrenze (né coppie né n-uple) ripetute.

Attributo
L'attributo è una proprietà elementare di un'entità o di una relationship, che ci interessa ai fini dell'applicazione d'interesse. Associa a ogni occorrenza di entità o relationship un valore appartenente ad un dominio, il cosiddetto dominio dell'attributo.

Attributo composito
Gli attributi commpositi raggruppano attributi di una medesima entità o relatioship che presentano affinità nel loro significato o uso (e.g. giorno, mese, anno si compone in data, via, numero civico, CAP in indirizzo, ecc...).

39 Concetti inesprimibili nel modello ER

Alcuni concetti sono inesprimibili attraverso il modello ER tradizionale, e bisognano quindi di costrutti particolari:

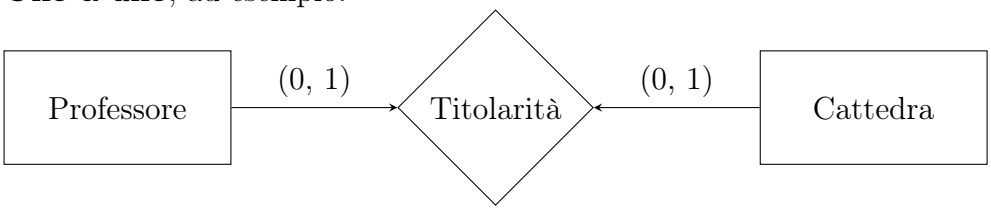
- **Cardinalità**
  - di relationship;
  - di attributo;
- **Identificatore**
  - interno;
  - esterno;
- **Generalizzazione**

Cardinalità di relationship
Una cardinalità di relationship è rappresentata da una coppia di valori associati ad ogni entità che partecipa alla relationship. Questi specificano il numero minimo e massimo di occorrenze della relationship a cui ciascuna occorrenza di entità può partecipare. I simboli usati saranno:

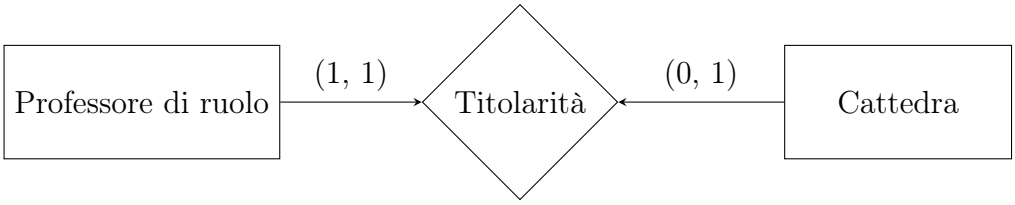
- Per la minima:
  - 0 → partecipazione opzionale
  - 1 → partecipazione obbligatoria
- Per la massima:
  - 1 → partecipazione opzionale
  - N → non pone alcun limite

Con riferimento alle cardinalità massime, abbiamo relationship:

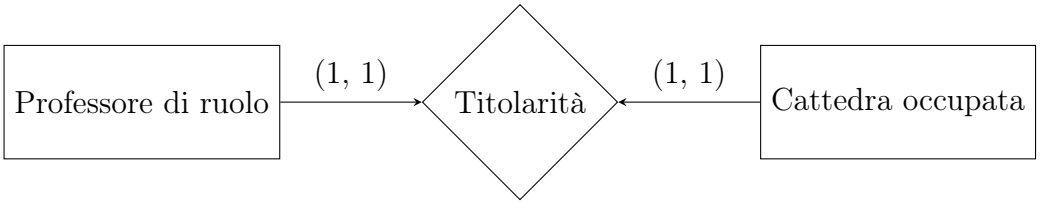
- **Uno a uno**, ad esempio:



Un professore può essere tale senza essere titolare di alcuna cattedra, e una cattedra può restare vuota.

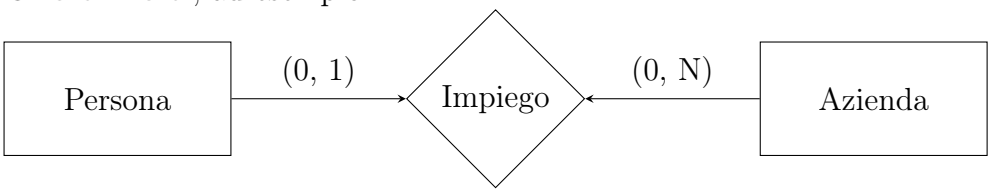


Un professore di ruolo dovrà ovviamente essere titolare di almeno una cattedra, ma questo non significa comunque che tutte le cattedre siano occupate.

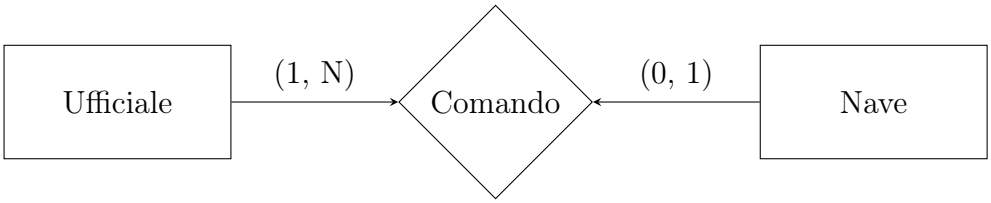


E' una forzatura? Sì. Serve a spiegare il concetto? Sempre sì.

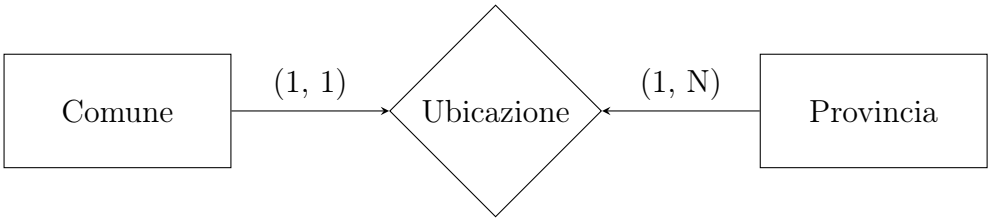
- **Uno a molti**, ad esempio:



Una persona può essere o non essere assunta, ma al massimo da una azienda. Al contrario, un'azienda avrà probabilmente più di un'impiegato.

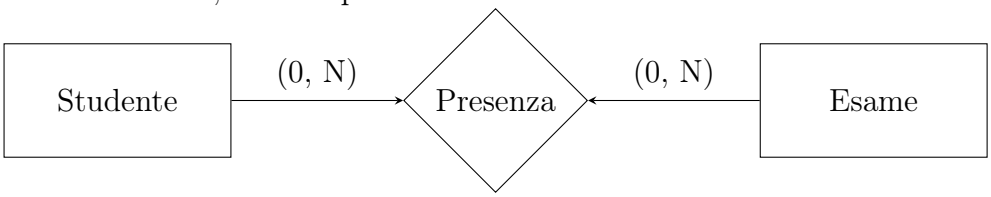


Un ufficiale per considerarsi tale deve comandare almeno una nave, ma non è detto che tutte le navi abbiano un comandante. Alcune sono navi fantasma come nei Pirati dei caraibi.

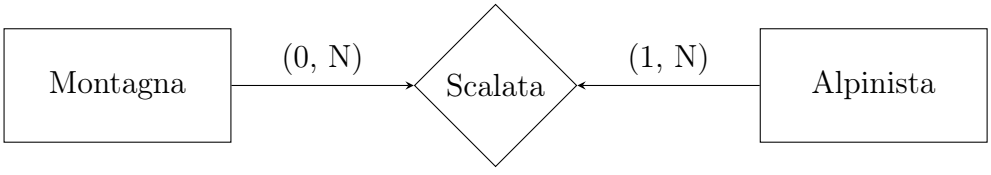


Ogni comune è ubicato in una e una sola provincia, mentre ogni provincia è l'ubicazione di più comnuni.

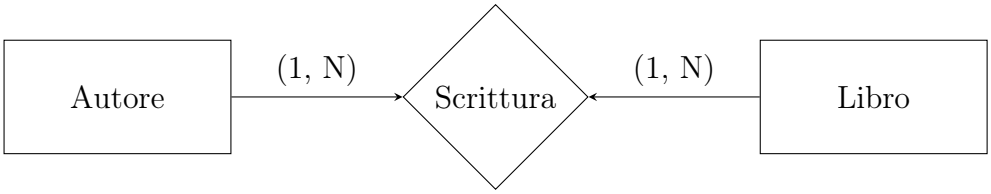
- **Molti a molti**, ad esempio:



Non è detto che ogni studente abbia sostenuto un'esame, come non è detto che ogni esame sia stato sostenuto da almeno uno studente.



Per potersi considerare alpinisti occorre aver scalato almeno una montagna, ma non è detto che tutte le montagne siano state scalate.

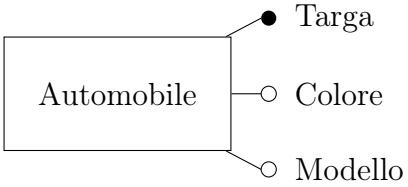


Ogni autore ha scritto almeno un libro per considerarsi tale, e ogni libro deve essere stato scritto da almeno un autore. Questo però non pregiudica che un autore non possa scrivere più libri o un libro non possa essere scritto scritto da più autori.

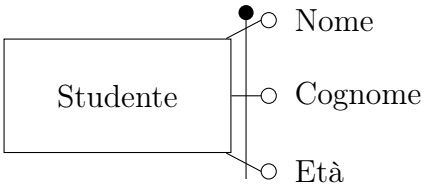
Identificatore di entità

L'identificatore di entità è uno strumento per l'identificazione univoca delle occorrenze di un entità.

- Gli attributi dell'entità possono formare l'**identificatore interno** (o chiave). Per capirsi, un'identificatore interno è una chiave primaria o comunque una chiave candidata.



Come vediamo dalla figura, targa è la chiave primaria, ovvero l'identificatore. Nel caso un'insieme di attributi sia una chiave, adottiamo la rappresentazione seguente:



- Gli attributi dell'entità e l'identificatore interno di entità esterne raggiunte attraverso relationship formano un'**identificatore esterno**. un'identificazione esterna può essere possibile solo nel caso esista una relationship in cui l'entità da identificare abbia cardinalità $(1, 1)$. Per

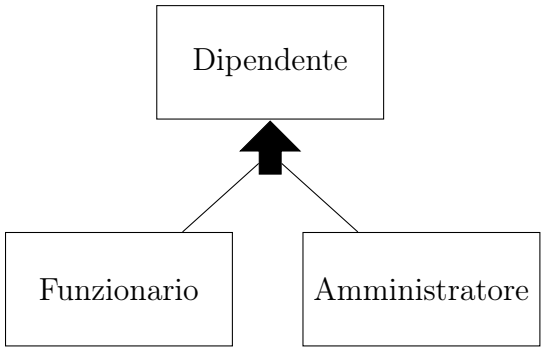
capirsi, un identificatore è una chiave contenente una chiave esterna.

Dalla figura si capisce che il nome dello studente e la sua iscrizione a una certa facolà formano il suo identificatore esterno.

Ogni entità deve possedere almeno un identificatore, ma può averne in generale più di uno.

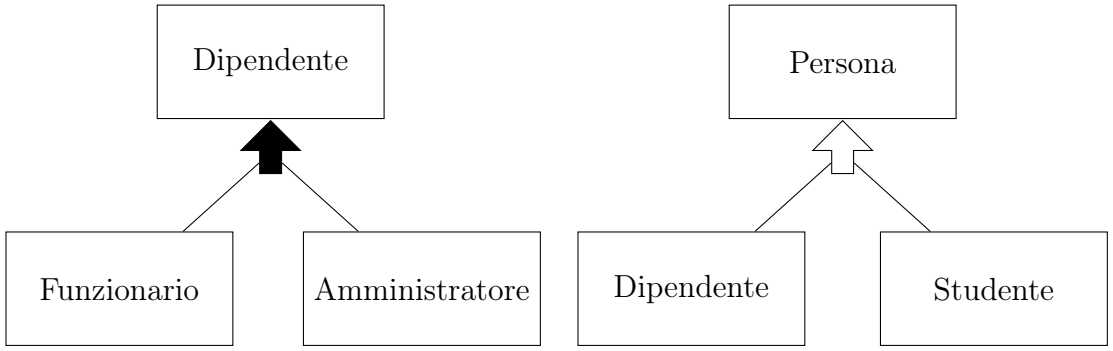
Generalizzazione

La generalizzazione mette in relazione una o più entità $E_1, E_2, ..., E_n$ con una singola entità E che le comprende come casi particolari. Si dice che E è una generalizzazione (oppure entità genitrice, madre) di $E_1, E_2, ..., E_n$ (dette specializzazioni, sottotipi o entità figlie).



Le caratteristiche delle generalizzazioni sono:

- **Ereditarietà**: tutte le proprietà dell'entità genitore vengono ereditate dalle figlie e non rappresentate esplicitamente.
- **Generalizzazione totale**: se ogni occorenza dell'entità genitore è rimpiazzata da almeno un occorrenza delle entità figlie, altrimenti si parla di **generalizzazione parziale**. Di norma le generalizzazioni parziali si indicano con una freccia vuota, mentre quelle totali con una freccia piena.



- **Generalizzazione esclusiva**: se ogni occorrenza dell'entità genitrice è occorrenza di al massimo una delle entità figlie, altrimenti si parla di **generalizzazione sovrapposta**.
- Possono esistere **gerarchie a più livelli** e multiple generalizzazioni allo stesso livello.
- Un'entità può essere inclusa in più gerarchie, sia come genitore che come figlia, se non entrambe.
- Se una generalizzazione ha solo un'entita figlia si dice **sottoinsieme**.
- Il genitore di una generalizzazione totale può non avere identificatore (**anonimato**) purché siano identificate le figlie.

40 Progettazione concettuale

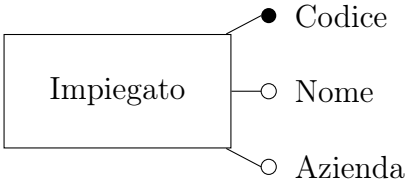
Vediamo adesso la progettazione concettuale nel dettaglio. Il modello ER sarà lo strumento fondamentale nel corso di questa fase: dovremo innanzitutto decidere, per una qualsiasi specifica fornitaci, quale sia il costrutto ER più adeguato da utilizzare. Per fare questo ci basiamo sulle definizioni dei costrutti del modello ER:

- **Entità**, se un oggetto ha proprietà significative e descrive oggetti con esistenza autonoma;
- **Attributo**, se un'oggetto è semplice e non ha proprietà specifiche;
- **Relationship**, se un concetto correla più oggetti;
- **Generalizzazione**, se un concetto è caso particolare di un altro.

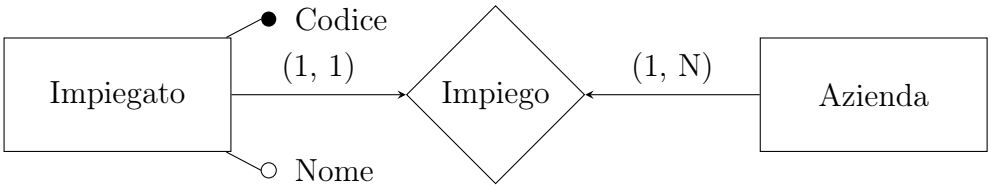
Design pattern

I design pattern sono soluzioni progettuali a problemi comuni. Sono largamente usati nell'ingegneria del software, e ne esistono alcuni comuni nella progettazione delle basi di dati.

- **Reificazione di attributo di entità**
  La reificazione di un attributo di un'entità è la promozione di tale attributo ad un'entità a sé stante, con ovvia creazione di una relationship che esprima il rapporto fra l'entità di partenza e l'attributo che abbiamo trasformato in entità. Supponiamo di avere il modello ER:



  L'attributo azienda potrebbe essere trasformato in un entità a sé, per poter includere maggiori informazioni rispetto all'azienda stessa:



- **Reificazione di relationship in entità**
  La reificazione di una relationship in entità è la promozione di una relationship in un'entità a sé stante, che sarà opportunamente collegata alle entità che metteva prima in relazione fra di loro attraverso altre relationship ausiliarie. Nello specifico, vediamo come usare la reificazione di relationship in entità per reificare relationship binarie e ternarie:

  – **Reificazione di relationship binarie**
    Diciamo di avere un modello abbastanza semplice, contenente due entità con identificatore interno legate da una relationship: Magari la relationship potrebbe avere bisogno di più informazioni specifiche: potremo allora usare più approcci alternativi. Il primo sarà di creare un entità corrispondente alla relationship, e di rendere poi gli attributi di quell'entità e le relazioni con le entità di partenza identificatore esterno. Un'altro approccio sarà quello di assegnare semplicemente alla nuova entità un identificatore univoco.

- **Reificazione di relationship ternarie**
  Un problema più sostanziale può essere rappresentato da una relationship ternaria. In questo caso si può adottare un approccio analogo al precedente, trasformando la relationship in un entità contenente tutte le informazioni necessario e legata alle entità di partenza con 3 distinte relationship, che potranno anche formare l'identificatore esterno della nuova entità.
- **Reificazione di attributo di relationship**
  Nel modello ER, nessuno ci impedisce di assegnare attributi alle relationship. Questo però andrà molto probabilmente tradotto in un entità a sé stante, che sarà legata attraverso le metodologie che abbiamo appena visto alle entità che legava in partenza.

- **"Parte di"**
  Il pattern parte di permette di definire concetti che sono parte di altri concetti, come ad esempio un cinema con le sue sale. Viene rappresentato da una relazione "composizione", uno a molti, dove l'oggetto che è parte dell'altro ha identificatore esterno nel primo.

- **"Istanza di"**
  Il pattern istanza di permette di definire concetti che sono istanze di altri concetti generali, come ad esempio un torneo e le sue edizioni. In questo caso si usa sempre una relazione, "occorrenza", uno a molti, dove l'oggetto occorrenza ha identificatore esterno nell'oggetto generale.

- **Caso particolare di entità**
  Il pattern caso particolare di entità permette definire casi particolari di determinate entità, attraverso la generalizzazione totale di un entità specializzata su un'entita generale. L'entità specializzata sarà il caso particolare.

Grafici di modelli ER per i pattern riportati si possono trovare al link: `https://github.com/Guray00/IngegneriaInformatica/blob/master/PRIMO%20ANNO/II%20SEMESTRE/Basi%20di%20dati/Diapositive/Slides%20Tonellotto/A.A.%2022-23/%5B06%5D%20Progettazione%20Concettuale%20e%20Logica.pdf`

Documentazione associata agli schemi concettuali

Il modello ER non basta mai da solo a descrivere l'interezza della realtà che vogliamo modellare. Esistono infatti, nonostante tutte le aggiunte che abbiamo apportato al modello ER, vincoli inesprimibili, che dovranno essere espressi dalla documentazione di supporto. Serviranno quindi:

- Un **dizionario dei dati**, ovvero una tabella che contenga informazioni riguardo alle entità e alle relationship che dovranno essere modellizzate dal nostro database.
- **Regole aziendali**: ovvero informazioni riguardo ai vincoli di integrità necessarie, nonché possibili derivazioni specifiche ai tipi di dati che staremo trattando (formule particolari per il calcolo di indici, medie...).

41 Strategie di progetto

Possiamo definire alcuni tipi di strategie di progetto:

- **Top-down**
  Si parte da uno schema iniziale e generale (top) che viene poi definito nei dettagli fino allo schema definitivo (down), muovendosi così, figurativamente, dall'alto verso il basso (zip!). La raffinazione dettagli avviene attraverso l'applicazione di primitive:

  Primitive di raffinamento

  - Da entità a associazione tra entità;
  - Da entità a generalizzazione;
  - Da associazione a insiemi di associazioni;
  - Da associazione a entità con associazioni;
  - Introduzione di attributi su entità e associazioni.

  Dove associazione è sinonimo di relationship.

- **Bottom-up**
  Si parte dalle specifiche delle componenti minime dello schema: queste specifiche vengono poi integrate fra di loro a formare lo schema completo. Come prima, figurativamente, dal basso verso l'alto. Questa integrazione avviene attraverso le primitive di trasformazione (o generalizzazione).

  Primitive di trasformazione

  - Generazione di entità;
  - Generazione di associazione;
  - Generazione di generalizzazione.

  Come sopra.

- **Inside-out**
  La progettazione inside-out è sostanzialmente una generalizzazione della bottom-down, dove si parte da strutture atomiche di complessità minore fino a schemi più complessi e sempre più vicini al risultato finale.

Nella pratica, una strategia non è mai puramente top-down o bottom-up, ma è più una **strategia mista**, che può essere sintetizzata come:

- Individuazione dei concetti principali e realizzazione di uno schema scheletro (un semplice schema concettuale che racchiude i dettagli più importanti della realtà da modellare);
- Decomposizione dello schema scheletro;
- Raffinazione, espansione e integrazione delle componenti dello schema.

Possiamo definire questa metodologia nei dettagli.

- **Analisi dei requisiti**
  - Analisi dei requisiti ed eliminazione delle ambiguità;
  - Costruzione di un glossario di termini;
  - Raggruppamento dei requisiti in insiemi omogenei.
- **Passo base**
  - Definizione uno schema scheletro con i concetti più rilevanti.
- **Passo iterativo**
  - Raffinazione dei concetti presenti sulla base delle loro specifiche;
  - Aggiunta di concetti necessari a descrivere specifiche non implementate.
- **Analisi di qualità**
  - Verifica della qualità dello schema e seguente modifica se necessaria. La qualità di uno schema concettuale può essere riassunta nella sua:
    * **Correttezza**
    * **Completezza**
    * **Leggibilità**
    * **Minimalità**

42 Progettazione logica

Giungiamo adesso alla progettazione logica del nostro schema. L'obiettivo della progettazione logica è quello di tradurre lo schema concettuale in uno schema logico che rappresenti i dati in modo corretto ed efficiente. Prendiamo quindi in ingresso lo schema concettuale, il modello logico usato e una serie di informazioni sul **carico applicativo**, ovvero lo sforzo a cui sarà sottoposto il sistema; in uscita avremo uno schema logico ben definito con relativa documentazione.

Valutazione delle prestazioni
La valutazione delle prestazioni può essere effettuata attraverso determinati indicatori, ovvero parametri che caratterizzano le prestazioni. I più importanti sono

- **Spazio**: numero di occorrenze previste;
- **Tempo**: numero di occorrenze (o relationship) visitate per portare a termine un'operazione.

Il calcolo degli indicatori di tempo può essere effettuato attraverso una tavola degli accessi: la tavola degli accessi relativa a una data operzione si ricava dallo schema di navigazione, costruito sul diagramma ER.

Ristrutturazione dello schema ER
A volte potrebbe essere necessario ristrutturare lo schema ER: ciò permette di semplificarne la traduzione e ottimizzare le prestazioni del sistema. Notiamo che uno schema ER ristrutturato non è più uno schema concettuale nel senso stretto del termine. La ristrutturazione si effettua attraverso i seguenti passaggi:

- **Analisi delle ridondanze**
  Una ridondanza in uno schema ER è un'informazione significativa ma derivabile da altre. Nella fase di progettazione logica si decide se eliminare le ridondanze eventualmente presenti nello schema oppure mantenerle (se non addirittura introdurne di nuove). Il vantaggio delle ridondanze è la semplificazione delle interrogazioni; gli svantaggi sono il maggiore spazio richiesto e l'appesantimento delle operazioni di aggiornamento. Le principali forme di ridondanza in uno schema ER sono:
  - **Attributi derivabili**
    Attributi che si possono derivare da attributi o della stessa entità o di entità diverse (o relationship).

- **Relationship derivabili**
  Relationship che si possono derivare dalla composizione di altre relationship (e.g. cicli).

- **Eliminazione delle generalizzazioni**
  Il modello relazionale non può rappresentare direttamente le generalizzazioni: dispone solamente di entità (relazioni) e relationship (dipendenza). Per questo motivo è necessario eliminare le gerarchie, sostituendole a loro volta con entità e relationship. Le metodologie possibili sono:

  - **Accorpamento delle figlie** della generalizzazione nel genitore, ovvero l'eliminazione della generalizzazione figlia, che verrà trasformata in una relationship ad entità la cui esistenza o non è segnalata da un certo attributo ("tipo"). Si noti che l'entità di partenza mantiene i suoi attributi;
  - **Accorpamento del genitore** della generalizzazione sulle figlie, ovvero un processo simile al precedente in cui si trasforma la gerachia in una serie di relationship. In questo caso gli attributi vanno a ricadere sulle figlie (si "accorpa" sulle figlie), e il genitore può essere eliminato direttamente;
  - **Sostituzione** della generalizzazione con relationship, ovvero la trasformazione diretta in relationship, dove gli attributi dell'entità genitrici e generalizzate restano alle loro entità d'appartenenza, e la loro gerarchia viene trasformata in relationship con cardinalità (0,1) $\rightarrow$ (1,1).

  La scelta fra le alternative possibili di eliminazione di generalizzazione può essere fatta sulla base della tabella degli accessi, seguendo alcune linee guida generali:

  - L'accorpamento delle figlie conviene se gli accessi al padre e alle figlie sono contestuali;
  - L'accorpamento del genitore conviene se gli accessi al padre e alle figlie sono differenti;
  - La sostituzione conviene se gli accessi alle entità figlie sono separati da quelli al padre.

  Notiamo poi che non sono escluse soluzioni ibride, sopratutto nel caso di gerarchie a più livelli.

- **Partizionamento/accorpamento di entità e relationship**
  Passiamo adesso alle ristrutturazioni fatte per rendere più efficienti le operazioni in base al principio di riduzione degli accessi. Questo si può ottenere:

  - **Separando gli attributi** di un concetto a cui si accede separatamente;
  - **Raggruppando attributi** di concetti diversi a cui si accede insieme.

  Considereremo, per semplicità, che ad ogni accesso si legge sempre l'intera informazione. Le casistiche principali saranno:

  - **Partizionamento verticale** di entità: il partizionamento verticale consiste nel dividere una data entità in due (sotto)entità, con un'opportuna relationship fra di esse. Ognuna delle due entità è in corrispondenza biunivoca con l'altra (cardinalità (1,1)) con chiave esterna corrispondente alla relationship per l'entità che non si prende la chiave interna.
  - **Partizionamento orizzontale** di relationship: il partizionamento orizzontale consiste nel dividere una data relationship dotata di attributi in più relationship corrispondenti a diversi valori di un attributo. Ad esempio, la divisione di una relationship d'appartenenza a due relationship di "appartenza corrente" e "appartenenza passata" in termini temporali.
  - **Eliminazione di attributi multivalore**: l'eliminazione di attributi multivalore viene incontro all'impossibilità di rappresentare attributi multivalore nel modello relazionale. Si trasformano quindi dati attributi in un'entità a sé stante che sarà collegata all'entità di partenza con una relazione di cardinalità (1,N) $\rightarrow$ (1,1) (o (1,N)?).
  - **Accorpamento di entità e relationship**: l'accorpamento di entità e relationship è l'esatto opposto del partizionamento verticale: due entità collegate da una relationship di cardinalità (1,1) $\rightarrow$ (0,1) potranno tranquillamente essere accorpate in un'unica entità con attributi a cardinalità (0,1) corrispondenti agli ex attributi della seconda entità.

- **Scelta di identificatori primari** La scelta degli identificatori primari è indispensabile alla traduzione nel modello relazionale, dove esiste il concetto di chiave. I criteri da adottare sono:

- L'assenza di opzionalità;
- La semplicità;
- Utilizzo nelle operazioni più frequenti o importanti

Nel caso nessuno degli identificatori soddisfi i requisiti necessari può essere necessario, sebbene sia poco elegante, introdurre nuovi attributi (codici) generati appositamente, magari in maniera incrementale.

Traduzione verso il modello relazionale
Approcciando la traduzione nel modello relazionale dello schema logico, dovremmo pensare di trasformare entità in relazioni sugli stessi attributi, e relationship in relazioni sugli identificatori delle entità coinvolte.

43 Dipendenze funzionali

Definiamo un metodo per valutare formalmente la qualità della progettazione degli schemi relazionali, ovvero misurare se un raggruppamento di attributi è migliore o peggiore di un altro, senza doversi affidare al buon senso del progettista dello schema o del modello ER. Seguiamo l'approccio top-down: iniziamo dall'individuare raggruppamenti più generali di attributi, ed effettuiamo poi decomposizioni successive per raffinare tali raggruppamenti generali. Gli obiettivi che abbiamo durante lo sviluppo del processo logico sono:

- **Conservazione dell'informazione**, ovvero il mantenimento delle informazioni che ci interessano della realtà in analisi.
- **Minimizzazione della ridondanza**, ovvero la riduzinoe al minimo della memorizzazione ripetuta della stessa informazione.

Possiamo ricavare da questo alcune linee guida:

- **Linea guida 1: semplice è bello**
  Uno schema di relazione deve essere progettato in modo da essere semplice da capire. Non si devono raggruppare attributi da più tipi di entità in un'unica relazione. Intuitivamente, anzi, dovremmo far corrispondere ad ogni schema di relazione una sola entità o una sola relationship, in modo da evitare **amibiguità semantiche**.
- **Linea guida 2: no alle anomalie**
  Gli schemi di basi di dati vanno progettati in modo che non si possano presentare anomalie in fase di definizione dei dati. La mancanza di anomalie va certificata attraverso una descrizione fondamentale

della semantica dei fatti descritti della realtà di interesse. Se si possono presentare anomalie, esse vanno rilevate e si devi assicurare che i programmi che aggiornano la base operino correttamente e in sicurezza.

- **Linea guida 3: evitare valori NULL**
  Conviene evitare il più possibile i valori NULL, in quanto riempiono lo schema di informazioni inutili. I valori NULL possono rivelarsi necessari solamente nei casi eccezionali rispetto alla cardinalità di una relazione.

Dipendenza Funzionale

Una dipendenza funzionale (*functional dependency, FD*) esprime un legame semantico tra due gruppi di attributi di uno schema di relazione R. Una FD è una proprietà di R, non un suo particolare stato valido r di R. Una FD non può quindi essere dedotta da uno stato valido r, ma deve essere definita esplicitamente sulla base della semantica degli attributi di R.

Forma normale

Una forma normale è una proprietà di una base di dati relazionale che ne garantisce la qualità, ovvero l'assenza di determinati difetti (quelli di cui si parla nelle linee guida). Una relazione non normalizzata presenta:

- **Ridondanze**;
- **Comportameti anomali** in fase di aggiornamento.

La normalizzazione è solitamente definita sui modelli relazionali, ma ha senso in altri contesti, ad esempio il modello ER.

Normalizzazione

La procedura di normalizzazione (che è un vero è proprio algoritmo) serve a trasformare schemi non normalizzati in schemi che godono della forma normale. La normalizzazione va utilizzata come tecnica di verifica di una base di dati, e non costituisce una metodologia di progettazione. Cercare di progettare uno schema di relazione attraverso la normalizzazione sarebbe infatti troppo complesso per rappresentare un'opzione viabile.

Definizione (pseudo)formale di dipendenza funzionale

Data una relazione r su $R(X)$, e due sottoinsiemi non vuoti Y e Z di x. Esiste una dipendenza funzionale in r da Y a Z se, per ogni coppia di n-uple t_1 e t_2 di r con gli stessi valori di Y, risulta che t_1 e t_2 hanno gli stessi valori anche su Z. Come avevamo già visto, la notazione è:

$$Y \to Z$$

Notiamo poi che $Y \to Z$ non implica $Z \to Y$. Una dipendenza funzionale è detta **completa** quando $Y \to Z$ e per ogni $W \subset Y$, non vale $W \to Z$.

Se Y è una superchiave di $R(X)$, allora Y determina ogni altro attributo della relazione, ergo $Y \to X$. A questo punto, se Y è una chiave (ovvero superchiave minimale), si ha che $Y \to X$ dalla chiave a $R(X)$ è completa. Una dipendenza funzionale si dice **banale** se è sempre soddisfatta:

- $Y \to Y$ è banale;
- $Y \to A$ se $A \notin Y$ non è banale;
- $Y \to Z$ non è banale se nessun attributo di Z appartiene a Y (è una condizione più generale della precedente).

Legami fra dipendenze funzionali e anomalie
Le dipendenze funzionali possono essere usate per verificare l'eventuale presenza di anomalie in un progetto. Tornano utili anche nella normalizzazione di schemi. Data la loro importanza, indicheremo con $R(X, F)$ uno schema di relazione $R(X)$ che rispetta un'insieme di dipendenze funzionali F.

Implicazione
Sia F un insieme di dipendenze funzionali definite su $R(Z)$, e sia $X \to Y$. Si dice che F implica (logicamente) $X \to Y$ se, per ogni possibile istanza r di R che verifica tutte le dipendenze funzionali di F, risulta verificata anche la dipendenza funzionale $X \to Y$. In simboli, $F \models X \to Y$. La definizione di implicazione non è direttamente utilizzabile nella pratica. Essa prevede una quantificazione universale sulle istanze della base di dati, e non disponiamo di un'algoritmo per calcolare tutte le dipendeze funzionali implicate da un'insieme F.

Regole di inferenza di Armstrong
Armstrong (1974) fornisce delle regole di inferenza che permettono di derivare costruttivamente tutte le dipendenze funzionali che sono implicate da un insieme di dipendenze iniziale. Esse sono:

- **Riflessività:**
$$Y \subseteq X \models X \to Y$$
- **Additività** (o arricchimento):
$$X \to Y \models XZ \to YZ \; \forall Z$$
- **Transitività:**
$$X \to Y \wedge Y \to Z \models X \to Z$$

Derivazione

Le regole di inferenza di Armstrong permettono di definire derivazioni. Dato un insieme di regole di inferenza RI, un insieme di dipendenze funzionali F, e una dipendenza funzionale f, una derivazione di f da F è una sequenza finita $f_1, ..., f_n$ tale che:

- $f_n = f$
- ogni f_i è un elemento di F o è ottenuta dalle precedenti dipendenze $f_1, ..., f_{n-1}$ attraverso una regola di inferenza RI.

Indichiamo con $F \vdash X \to Y$ il fatto che la dipendenza funzionale $X \to Y$ sia derivabile da F usando RI. Le regole di derivazione comunemente usate sono, a partire dalle regole di inferenza di Armstrong:

- **Unione:**
$$\{X \to Y, X \to Z\} \vdash X \to YZ$$
Dimostrazione:
  1. $X \to Z$ da ipotesi;
  2. $X(X) \to ZX$ da additività di (1);
  3. $X \to Y$ da ipotesi;
  4. $ZX \to YZ$ da additività di (3);
  5. $X \to ZX,\ ZX \to YZ \Rightarrow X \to YZ$ dalla transitività di (2) e (4).

- **Decomposizione:**
$$\{X \to YZ\} \vdash X \to Y$$
Dimostrazione:
  1. $X \to YZ$ da ipotesi;
  2. $YZ \to Y$ da riflessività;
  3. $X \to YZ,\ YZ \to Y \Rightarrow X \to Y$ dalla transitività di (1) e (2).

- **Indebolimento:**
$$\{X \to Y\} \vdash XZ \to Y$$
  1. $XZ \to X$ per riflessività;
  2. $X \to Y$ da ipotesi;
  3. $XZ \to X\ X \to Y \Rightarrow XZ \to Y$ dalla transitività di (1) e (2).

- **Identità:**

$$\{\} \vdash X \to X$$

Chiusura di un insieme di attributi
Dato uno schema $R(T, F)$ con $X \subseteq T$, la chiusura di X rispetto a F, indicata col simbolo R_F^+, è definita come:

$$X_F^* = \{A \subset T | F \vdash X \to A\}$$

Se non vi sono amibiguità si può semplicemente scrivere X^+.

Teorema di chiusura degli attributi
Se è possibile, usando le regole di inferenza, scrivere, partendo da F, che $X \to Y$, allora Y è contenuto nella chiusura di X e viceversa. In simboli:

$$F \vdash X \to Y \Leftrightarrow Y \subseteq X^+$$

Correttezza e completezza
Dato un qualche insieme di regole di inferenza RI e un insieme di dipendenze funzionali F, RI è corretto se:

$$F \vdash X \to Y \Rightarrow F \models X \to Y$$

Ovvero applicando RI ad un insieme F di dipendenze funzionali, si ottengono solo dipendenze logicamente implilcate da F. Inoltre RI è completo se:

$$F \models X \to Y \Rightarrow F \vdash X \to Y$$

Ovvero applicando RI a un insieme F di dipendenze funzionali si ottengono tutte le dipendenze logicamente implicate da F. A questo punto si può enunciare:

Teorema fondamentale della correttezza e completezza
Le regole di inferenza di Armstrong sono corrette e complete. Questo teorema ci permette di scambiare i simboli $\models$ e $\vdash$. In particolare questo si applica alla definizione di chiusura di attributi, cioè:

$$X_F^+ = \{A \subset T | F \models X \to A\}$$

Si può dimostrare che le regole di inferenza di Armstrong sono minimali, ovvero che non se ne può ignorare anche soltanto una senza privarle della completezza. Esistono però altri insieme di regole di inferenza completi che non coincidono con le regole di inferenza di Armstrong.

Chiusura di un'insieme di dipendenze funzionali

Sia F un insieme di dipendenze funzionali definite su $R(Z)$. Allora la chiusura di F è l'insieme F^+ di tutte e sole le dipendenze funzionali implicate da F:

$$F^+ = \{X \to Y | F \models X \to Y\}$$

Dato un'insieme di dipendeze funzionali F definite su $R(Z)$, un'istanza r di R che soddisfa F soddisfa anche le dipendenze funzionali di F^+.

Algoritmo per il calcolo di $\mathbf{F^+}$

Riportiamo adesso un'algoritmo per il calcolo della chiusura di un insieme di dipendenze funzionali F^+ a partire da F, usando le regole di inferenza di Armstrong:

- Input: $R(T, F)$;
- Output: F^+ *chiusura*.

```
metti F in F+
while (F+ non cambia) do
  foreach f in F+ do
    #applica riflessivita' e addittivita' a f e aggiungi le
    dipendenze ottenute a F+
  foreach f_1, f_2 in F+ do
    #se possibile, applica la transitivita' a f_1 e f_2 e
    aggiungi a F+ la dipendenza ottenuta
return F+
```

Calcolare F^+ è molto costoso: l'algoritmo è a complessità esponenziale. Spesso però, quello che ci interessa beramente e verificare se F^+ contiene una certa dipendenza e non generare l'intera lista di chiusura. Per fare ciò basta calcolare X^+ per il teorema di chisura degli attributi:

$$F \vdash X \to Y \Leftrightarrow Y \subseteq X^+$$

Algoritmo per il calcolo di $\mathbf{X^+}$

Vediamo quindi come calcolare la chisurua X^+:

- Input: $R(T, F), \quad X \subseteq T$;
- Output: X^+ *chiusura*.

```
metti X in X+
while (X+ non cambia) do
  foreach W -> V in F do
    if W in X+ and V not in X+ then
      metti V in X+
return X+
```

Chiavi

Dato uno schema $R(T, F)$, un'insieme di attributi $K \subseteq T$ si dice **superchiave** di R se la dipendenza funzionale $K \to T$ è implicata da F, ovvero se $K \to T \in F^+$. Un'insieme di attributi $K \subseteq T$ si dice a questo punto **chiave** di R se K è una superchiave di R e non esiste alcun sottoinsieme proprio di K che sia superchiave di R. Dato che in uno schema possono esserci più chiavi, di solito si identifica una chiave primaria che possa fare da identificatore per tutte le n-uple dello schema. Tutte le altre chiavi si dicono chiavi candidate.

Algoritmo per tutte le chavi

Il problema di trovare tutte le chiavi di una relazione $R(Z)$ ha complessità esponenziale nel caso peggiore:

- Gli attributi che stanno solo a sinistra sono in tutte le chiavi. Si chiami N questo insieme;
- Gli attributi che stanno solo a destra non sono in nessuna chiave. Si chiami M questo insieme;
- Si aggiunge a N un'attributo alla volta tra quelli che non sono né in N nè in M, poi una coppia di attributi, e così via. Chiamiamo X_i questo sottoinsieme di attributi: ad ogni aggiunta controlleremo se la dipendenza $N \cup X_i \to Z$ esiste.

Verifica di una chiave

Spesso è molto più conveniente verificare se una chiave è tale, piuttosto che trovare ogni possibile chiave. Per fare ciò, possiamo usare l'algoritmo per il calcolo della chiusura di un'insieme di attributi:

- $X \subseteq T$ è superchiave di $R(T, F)$ se e solo se $X \to T \in F^+$, ovvero $T \subseteq X^+$
- $X \subseteq T$ è chiave di $R(T, F)$ se e solo se $T \subseteq X^+$, e non esiste $Y \subset X$ tale che $T \subseteq Y^+$.

Equivalenza

Due insiemi di dipendenze funzionali F e G sugli attributi T di una relazione $R(T)$ sono equivalenti, in simboli $T \equiv G$, se e solo se $F^+ = G^+$. La relazione di equivalenza permette di stabilire se due insiemi di dipendenza rappresentano gli stessi fatti. Per verificare l'equivalenza è sufficiente che:

- Tutte le dipendenze di F appartengano a G^+;
- Tutte le dipendenze di G appartengano a F^+.

Ridondanza

Sia F un insieme di dipendenze funzionali. Data $X \to Y \in F$, X contiene un **attributo estraneo** se e solo se:

$$(F - \{X \to Y\}) \cup (X - \{A \to\} Y) \equiv F$$

ovvero $X - \{A\} \to Y \in F^+$

$X \to Y$ è una **dipendenza ridondante** se e solo se $(F - \{X \to Y\}) \equiv F$, ovvero $X \to Y \in (F - \{X \to Y\})^+$. Le dipendenze che non contengono attributi estranei e la cui parte destra è un unico attributo sono dette **dipendenze elementari**.

Copertura minimale

Sia F un'insieme di dipendenze funzionali. F è una **copertura minimale** (detta anche *insieme minimale* e *copertura canonica*) se e solo se:

- Ogni parte destra di una dipendenza ha un unico attributo;
- Le dipendenze non contengono attributi estranei;
- Non esistono dipendenze ridondanti;

In soldoni, la copertura minimale rappresenta l'insieme di dipendenze funzionali F' più piccolo che implica tutte le dipendenze di F.

Algoritmo per il calcolo della copertura minimale

- Input: F;
- Output G *copertura minimale*:

```
metti F in G
for each X -> G do
  metti X in Z
  for each A in X Do
    if Y in (Z - {A})+_F then
      metti Z - {A} in Z
    metti (G - {X -> Y}) U (Z -> Y) in G
  foreach F in G do
    if f in (G-{f})+ then
      metti G - {f} in G
return G
```

Teorema della copertura minimale

Il precedente algoritmo dimostra il fatto che per ogni insieme di dipendenze funzionalil F esiste una copertura minimale. Si noti che questo non afferma che la copertura minimale è unica: possono tranquillamente esistere coperture minimali equivalenti.

44 Normalizzazione

Eliminare le anomalie
La teoria che abbiamo sviluppato finora viene usata per identificare le anomalie in uno schema mal definito. Definiamo quindi il concetto di **forma normale**: una forma normale è una proprietà che deve essere soddisfatta dalla dipendenza fra attibuti di schemi "ben fatti". Noi vedremo due esempi: la **forma normale di Boyce-Codd**, e un suo miglioramento, la **terza forma normale**.

Forma normale di Boyce-Codd
Uno schema $R(T, F)$ è in forma normale di Boyce-Codd (BCNF) se e solo se per ogni dipendenza funzionale non banale $X \to Y \in F^+$, X è superchiave di R. Per definizione, il fatto che uno schema sia o meno in BCNF dipende dalla chiusura F^+, non dalla specifica copertura F. Calcolare F^+, come abbiamo visto, ha complessità esponenziale: fortunatamente esiste un'algoritmo di complessità polinomiale per valutare se uno schema è in forma BCNF. Si usa il corollario: uno schema $R(T, F)$ con F copertura minimale è in BCNF se e solo se per ogni dipendenza funzionale elementare $X \to A \in F$, X è superchiave.

- Input: $R(T, F)$;
- Output true se R è in BCNF, false altrimenti.

```
for each X -> Y in F do
  if Y not in X and T not in X+ then
    return false
return true
```

Decomposizione di schemi
Dato unos schema $R(T)$, l'insieme di schemi $\rho = \{R_1(T_1), ..., R_k(T_k)\}$ è una **decomposizione** di R solo se $\cup_i T_i = T$ (l'unione degli schemi dà lo schema di partenza). Si noti che questo non richiede che gli schemi R_i siano disgiunti. Una decomposizione equivale allo schema di partenza in quanto

- Preserva i dati;
- Preserva le dipendenze funzionali.

Teorema della perdita di dati
Si ha che se $\rho = \{R_1(T_1)...., R_k(T_k)\}$ è una decomposizione di $R(T, F)$, allora per ogni relazione r che soddisfa $R(T, F)$:

$$r \subseteq \pi_{T_1}(r) \bowtie ... \bowtie \pi_{T_k}(r)$$

Ciò significa che una decomposizione ha perdita di dati quando, ricostruendo una relazione, otteniamo più n-uple della relazione originaria.

Decomposizioni che preservano i dati
Dato uno schema $R(T, F)$ e una decomposizione $\rho\{R_1(T_1), ..., R_k(T_k)\}$, ρ preserva i dati se e solo se, per ogni relazione r che soddisfa $R(T, F)$ si ha:

$$r = \pi_{T_1}(r) \bowtie ... \bowtie \pi_{T_k}(r)$$

Ciò significa che, per una decomposizione che preserva i dati, ogni istanza valida r della relazione di partenza deve essere identica al join naturale delle sue proiezioni sui T_1 (cioè ricostruendo la relazione si ottiene la relazione di partenza e nulla di più).

Teorema di preservazione dei dati
Sia $\rho = \{R_1(T_1), R_2(T_2)\}$ una decomposizione di $R(T, F)$. Essa preserva i dati se e solo se $T_1 \cap T_2 \to T_1 \in F^+$ oppure $T_1 \cap T_2 \to T_2 \in F^+$. In altre parole, gli attributi comuni alle due relazioni devono essere chiave in una delle due tabelle (relazioni).

Proiezione di un insieme di dipendenze
Dato $R(T, F)$ e $T_1 \subseteq T$, la proiezione dell'insieme di dipendenze F sull'insieme di attributi T_i è:

$$\pi_{T_i} = \{X \to Y \in F^+ | X, Y \subseteq T_i\}$$

Nota bene che la proiezione si costruisce sulle dipendenze di F^+, non di F soltanto.

Algoritmo per il calcolo di $\pi_{T_i}(F)$
Si presenta un'algoritmo per il calcolo della proiezione dell'insieme di dipendenze appena definita.

- Input: $R(T, F)$ e $T_i \subseteq T$
- Output: $\pi_{T_i}(F)$

```
Z vuoto
for each Y in T do
  metti Y - Y in W
  metti Z unito (Y -> (W intersecato T_i)) in Z
return Z
```

Questo algoritmo ha, nel caso pessimo, complessità esponenziale.

Decomposizioni che preservano le dipendenze
Dato uno schema $R(T, F)$, e una decomposizione $\rho = \{R_1(T_1), ..., R_k(T_k)\}$, ρ è una decomopsizione di $R(T, F)$ che preserva le dipendenze se e solo se

$\cup_i \pi_{T_i}(F) \equiv F$ In altre parole, la decomposizione di $R(T, F)$ in due relazioni con attributi X e Y preserva le dipendenze se $\pi_X(F) \cup \pi_Y(F) \equiv F$, cioè se $(\pi_X(F) \cup \pi_Y(F)) = F^+$. Dovremo verificare quest'ultima uguaglianza per dimostrare che una decomposizione di $R(T, F)$ preserva le dipendenze. Fare ciò significa:

- Calcolare la proiezione di un insieme di dipendenze funzionali su un insieme di attributi, cosa che richiede un algoritmo a complessità esponenziale;
- Determinare l'equivalenza di due insiemi di dipendenze funzionali X e G, cosa che richiede un'algoritmo a complessità poliniomiale:
  - Per ogni $X \to Y \in F$, calcoliamo X_G^+ e verifichiamo se $Y \in X_G^+$;
  - Per ogni $X \to Y \in G$, calcoliamo X_F^+ e verifichiamo se $Y \in X_F^+$;

Algoritmo per la decomposizione in BCNF

Possiamo adesso vedere un'algoritmo che decompone uno schema $R(T, F)$ nella sua forma normale di Boyce-Codd:

- Input: $R(T, F)$ con F in forma $X \to A$, non normalizzata
- Output: ρ che preserva i dati

```
metti R(T,F) in P
while esiste R_i(T_i, F_i) in P che non e in BCNF do
  for each X -> A in F_i do
    if A not in X and T_i not in X+ then
      metti R_i(T_i - A, proiezione su T_1 - A di F_i) in R_1
    #questi sono tutti gli attributi di R - A
      metti R_i(X + A, proiezione su X + A di F_i) in R_2 #
   questi sono tutti gli attributi della dipendenza
   funzionale in esame
      metti P - (R_i unito a {R_1,R_2}) in P
      break
return P
```

Questo algoritmo termina e produce una decomposizione della relazione tale che:

- La decomposizione prodotta è in BCNF;
- La decomposizione prodotta preserva i dati.

Non è detto che la decomposizione preservi le dipendenze!

Qualità delle decomposizioni

Una decomposizione dovrebbe sermpe garantire:

- Di essere in BCNF;
- Di non presentare perdite di dati, in modo da permettere la ricostruzione dell'informazione originale attraverso join naturali;
- Di conservare le dipendenze funzionali, in modo da mantenere i vincoli di integrità originari.

Quando non si riesce a raggiungere una forma normale di Boyce-Codd, probabilmente c'è stato un cattivo processo di progettazione prima della normalizzazione. In ogni caso, esiste una forma normale alternativa (e meno restrittiva) alla Boyce-Codd:

Terza forma normale
Una relazione $R(T, F)$ è in terza forma normale (3NF) se e solo se, per ogni dipendenza funzionale non banale $X \to A \in F^+$, è verificata almeno una delle due condizioni:

- X è una superchiave di R (come nella Boyce-Codd);
- A è contenuto in almeno una chiave di R (si dice che A è un attributo primo).

Vediamo che la BCNF implica la 3NF: uno schema in BCNF è automaticamente in 3NF, ma non tutti gli schemi in 3NF sono in BCNF. Il problema della 3NF è la sua **verifica**: il problema di decisione è NP-completo. Il miglior algoritmo deterministico di risoluzione ha complessità esponenziale nel caso peggiore. Questo è dato dalla ricerca degli attributi primi, cioè le chiavi, che ha complessità esponenziale. Tuttavia, si può sempre ottenere una decomposizione in 3NF che preserva dati e dipendenze funzionali.

Algoritmo per la decomposizione in 3NF
Ci basiamo su un intuizione: dato un insieme di attributi T e una copertura minimale G, si divide G in gruppi G_i in modo che tutte le dipendenze funzionali di ogni gruppo G_i abbiano la stessa parte sinistra. Da ogni gruppo G_i si definisce uno schema di relazione composto da tutti gli attributi che appaioni in G_i, la cui chiave (detta **chiave sintetizzata**) è la parte sinistra comune. L'algoritmo risulta quindi:

- Input: $R(T, F)$
- Output: ρ che preserva dati, dipendenze ed è in 3NF

```
metti la copertura minimale di F in G
poni P vuoto
#sostituisci ogni insieme di dipendenze {X -> A_1, ..., X ->
   A_h} con X -> A_1, ..., A_h
for each X -> Y in G do
  #crea uno schema con attributi XY in P
#elimina da P ogni schema che sia contenuto in un'altro
   schema di P
#se P non contiene nessuno schema i cui attributi
   costituiscono una superchiave di R, aggiungere a P uno
   schema con attributi W, dove W e' una chiave di R
```

L'esecuzione dell'algoritmo termina e produce una decomposizione tale che:

- La decomposizione prodotta è in 3NF;
- La decomposizione prodotta preserva i dati e le dipendenze funzionali.

Questo algoritmo ha complessità polinomiale: paradossalmente, trovare una soluzione è polinomiale, ma verificarla è esponenziale

45 Tecnologia della base di dati

Finora abbiamo dato una descrizione della base di dati (o meglio, del DBMS) in quanto astrazione, cioè nei limiti delle funzionalità che offriva all'utente. Vediamo ora alcuni dettagli del funzionamento interno e dell'implementazione effettiva dei DBMS, per poterne fare un'uso più adeguato ed avere accesso ad alcuni servizi particolari. Avevamo specificato che un DBMS è un software in grado di gestire collezioni di dati che siano

- **Grandi**: molto maggiori della memoria centrale dei sistemi di calcolo utilizzati;
- **Persistenti**: con un periodo di vità maggiore delle esecuzioni dei programmi che le utilizzano;
- **Condivise**: usate in contemporanea da più applicazioni diverse.

Tutto questo mantenendosi:

- **Affidabili**: in grado di resistere a malfunzionamenti;
- **Sicuri**: in grado di assicurare la privacy degli utenti e la sicurezza dei dati attraverso una politica sugli accessi;
- **Efficienti**: in grado di non sprecare risorse di sistema e tempo;

- **Efficaci**: utili nel rendere produttive le attività degli utenti.

Vediamo più nel dettaglio alcune di queste caratteristiche.

Grandezza e persistenza

Il fatto che le moli di dati gestite dai DBMS sono grandi e persistenti richiede l'utilizzo di una memoria secondaria, in quanto sarebbe impossibile mantenere l'intero database in memoria centrale. Questo implica lo sviluppo di una certa rappresentazione fisica più efficiente per i dati immagazzinati in memoria secondaria, che differisce dal modello logico che è disponibile agli utenti. Inoltre, la memoria secondaria sarà più lenta della memoria centrale, ergo bisogna sviluppare interazioni fra memoria principale e secondaria che limiti il più possibile gli accessi alla secondaria, in modo da ottimizzare i tempi e l'uso delle risorse.

Condivisione

L'accesso condiviso ai dati delle basi di dati (che sono cosiddette **risorse integrate**) consiste in:

- **Attività diverse** su dati condivisi in parte (ergo meccanismi di autorizzazione);
- **Attività multi-utente** su dati condivisi (ergo controllo della concorrenza).

Questo tipo di situazioni viene risolto dal meccanismo delle **transazioni**, che verrà approfondito più avanti. Intuitivamente, le transazioni sono corrette quando sono seriali, ovvero eseguite in sequenza. Di contro, imporre questa sequenzialità renderebbe la base di dati inefficiente. Per questo si usano meccanismi di **controllo della concorrenza**, che raggiungono un buon compromesso.

Affidabilità

L'affidabilità di una base di dati consiste nella sua capacità di resistere e conservare informazioni anche in presenza di malfunzionamenti. Questo diventa difficile nel caso le transazioni (o semplicemente gli aggiornamenti della base di dati) siano frequenti, e richiedano sistemi di gestione anche piuttosto complessi della memoria. Nelle basi di dati, l'affidabilità viene assicurata attraverso la cosiddetta **acidità** (*acid compliance*), dall'acronimo Atomiche Consistenti Isolate Definitive. Le transazioni acide devono essere:

- **Atomiche**: una transazione viene eseguita o meno, senza vie di mezzo;
- **Consistenti**: una transazione non deve violare i vincoli di integrità già stabiliti sulla base di dati;

- **Isolate**: una transazione deve essere poter eseguita assieme ad altre come da sola, e dare lo stesso risultato;
- **Definitive**: una volta eseguita, una transazione non deve essere dimenticata.

46 Gestione delle transazioni

Vediamo quindi la parte del DBMS che gestisce le transazioni, ergo la concorrenza e l'affidabilità delle operazioni di modifica operate sul database. Un criterio per classificare le basi di dati è il numero di utenti che ne possono usufruire simultaneamente. Si ha che una DBMS può essere:

- **Monoutente**: se può essere usata da un solo utente per volta, senza rischi di concorrenza;
- **Multiutente**: se può essere usata da più utenti per volta, quindi con il rischio di accessi concorrenti. La stragrande maggioranza dei DBMS è di questo tipo.

Multitasking

L'accesso di più utenti alla base di dati è permesso dal concetto del multitasking. Il multitasking consente al calcolatore (anzi uno dei calcolatori che ospita la base di dati) di eseguire più programmi (o meglio processi) contemporaneamente. Se esiste una sola CPU, in verità, si può eseguire effettivamente un solo processo per volta, ma i sistemi operativi sono forniti di meccanismi che sospendono e riprendono i processi in corso per assicurare l'esecuzione contemporanea. Si parla in questo caso di esecuzione **alternata** (*interleaved*). Su processori multithreaded (ovvero a più CPU) si può invece vere l'**esecuzione parallela** dei processi.

Transazioni

Una transazione identifica un'unità elementare di lavoro svolta da un'applicazione, a cui si vogliono associare caratteristiche acide. Sostanzialmente, una transazione è una successione di operazioni da eseguire successivamente che forma un'unità logica da eseguire sui dati. Una transazione comprende una o più operazioni di accesso alla base di dati, che possono essere:

- Inserimenti;
- Cancellazioni;
- Modifiche;

- Interrogazioni.

Che si traducano effettivamente in una sequenza di letture e scritture. Un sistema che mette a disposizione un meccanismo per la definizione ed esecuzione, sopratutto concorrente, di transazioni è detto **sistema transazionale**, o OLTP (*online transaction processing*).
Una transazione, più formalmente è una parte di programma caratterizzata da un inizio e una fine al cui interno deve essere eseguito uno e uno solo fra i comandi:

- **Commit**: applica le modifiche e prosegue;
- **Rollback/Abort**: annulla la transazione e riporta la base di dati allo stato precedente.

Stato di una transazione
Lo stato di una transazione è importante alla sua corretta esecuzione. I possibili stati in cui può trovarsi una transazione sono:

- **Attivo**: la transazione è stata appena avviata, e resterà in questo stato fino alla fine della sua esecuzione;
- **Commit parziale**: la transazione è terminata e le modifiche alla base di dati sono pronte ad essere applicate;
- **Commit**: la transazione è stata eseguita con successo e tutti i suoi aggiornamenti alla base di dati sono permanenti;
- **Fallito**: la transazione non può fare commit, per un errore interno autorilevato (suicidio), o è stata interrotta per fallimento mentre si trovava nello stato attivo (omicidio).
- **Abortito**: la transazione è stata interrotta e tutti gli aggiornamenti alla base di dati sono stati annullati.

Di norma, una transazione parte nello stato attivo, e si va a trovare poi, a seconda dell'esecuzione, in uno stato di commit parziale o in uno stato di fallimento. Non è detto che un commit parziale non possa all'ultimo essere rilevato come fallimento. In ogni caso, se il commit parziale va a buon fine, ci si ritrova nello stato di commit, mentre se si verifica un fallimento, ci si ritrova nello stato abortito.

Proprietà delle transazioni
Vediamo nel dettaglio le proprietà acide delle transazioni:

- **Atomicità**
  Una transazione è un'unità atomica di elaborazione, che non può lasciare la base in stati intermedi, ma solamente in uno stato pre-transazione o post-transazione. Un guasto prima del commit non deve avere conseguenze della base di dati, e deve causare un annullamento (*undo*) delle operazioni. Allo stesso tempo, nemmeno un guasto dopo il commit deve avere conseguenze, ed eventualmmente si deve poter ripetere la transazione (*redo*). Nella maggior parte dei casi, l'esito di una transazione deve essere il commit con successo. Nella minoranza dei casi di errore, si deve poter effettuare un rollback (o abort), che può essere richiesto sia dall'applicazione, per il rilevamento di un'errore interno (suicidio), o dal sistema, per la violazione delle regole della base di dati (omicidio).

- **Consistenza**
  Una transazione deve rispettare i vincoli di integrità. Applicata ad una base di dati corretta, una transazione deve lasciare la base di dati in uno stato corretto. In altre parole, all'inizio e alla fine di una transazione il sistema è lasciato in uno stato consistente. Può accadere che durante la transazione la base di dati si ritrovi *temporaneamente* in uno stato incosistente.

- **Isolamento**
  La transazione non deve risentire degli effetti di altre transazioni concorrenti, e l'esecuzione concorrente delle transazioni non deve dare risultati diversi dalla loro esecuzione sequenziale. Questo significa che una transazione non espone i suoi stati intermedi, e si evita il cosiddetto "effetto domino", dove gli effeti di una transazione si ripercuotono su altre transazioni concorrenti.

- **Durabilità** (o persistenza)
  Gli effetti di una transazione dopo il commit non vanno perduti (perdurano per sempre), e la transazione stessa deve poter essere ripetuta. Questo significa che eventuali guasti dopo il commit devono poter essere ripristinati e non devono avere effetti permanenti sulla base di dati.

Controllo della concorrenza

La concorrenza è la carattersitica fondamentale delle transazioni. Visto che la mole di transazioni da eseguire in media su una base di dati è molto grande, non si può optare per un sistema seriale. Si rende quindi necessario un sistema di gestione della concorrenza. Il *modello di riferimento* sarà formato da operazioni di input-output su oggetti astratti $x, y, z...$, dove il problema è

quello di eliminare le anomalie causate dall'esecuzione concorrente. Possiamo classificare queste anomalie:

- **Perdita di aggiornamento**
  Poniamo di avere due transazioni identiche:

  - $T_1 : r(x), x \to x + 1, w(x)$
  - $T_2 : r(x), x \to x + 1, w(x)$

  Ergo, leggi x, incrementalo di 1, e scrivi. Partendo da $x = 2$. un'esecuzione seriale di queste due transazioni dovrebbe giustamente portare $x = 4$. Vediamo cosa potrebbe accadere nel caso di esecuzione concorrente:

```
T_1     |  T_2
--------|--------
bot     |
r(x)    |
x++     |
        | bot
        | r(x)
        | x++
w(x)    |
commit  |
        |  w(x)
        | commit
```

  In questo caso, la lettura di T_2 avverrebbe dopo l'incremento di T_1, ma prima del suo commit: tutto ciò che è stato fatto da T_1 verrebbe quindi perso, e il risultato (scorretto) sarebbe $x = 3$.

- **Lettura sporca**
  Abbiamo le transazioni:

  - $T_1 : r(x), x \to x + 1, w(x)$
  - $T_2 : r(x)$

  Consideriamo l'esecuzione:

```
T_1     |  T_2
--------|--------
bot     |
r(x)    |
x++     |
w(x)    |
        | bot
        | r(x)
```

```
abort   |
        |  commit
```

In questo caso, T_2 ha letto lo stato di x mentre era in uso da T_1, ergo violando l'isolamento e costringendo T_1 ad abortire.

- **Letture inconsistenti**
  Abbiamo le transazioni:
  - $T_1 : r(x), r(x)$
  - $T_2 : r(x), x \to x + 1, w(x)$

  Consideriamo l'esecuzione:

```
T_1     |  T_2
--------|--------
bot     |
r(x)    |
        | bot
        | r(x)
        | x++
        | w(x)
        | commit
r(x)    |
commit  |
```

In questo caso, durante la stessa esecuzione, T_1 avrebbe letto due valori diversi di x, cosa impossibile!

- **Aggiornamenti fantasma**
  Abbiamo le transazioni:
  - $T_1 : r(y), r(z), s \to y + z$
  - $T_2 : y \to y - 100, r(z), z \to z + 100, w(y), w(z)$

  Assumiamo valga il vincolo $y + z = 1000$:

```
T_1          |  T_2
-------------|-------------
bot          |
r(y)         |
             | bot
             | y <- y - 100
             | r(z)
             | z <- z + 100
             | w(y)
             | w(z)
             | commit
```

```
r(z)        |
s <- y + z  |
commit      |
```

In questo caso alla fine avremo $s = 1100$, il vincolo non è soddisfatto per T_1, in quanto vede un aggiornamento non coerente (lo sarebbe diventato un'istante dopo).

- **Inserimento fantasma**
  Poniamo di avere due transazioni su database, che comporta un'operazione di media:

```
T_1                       |  T_2
--------------------------|---------------------
bot                       |
r(stipendi_impiegati)     |
avg(stipendi_impiegati)   |
                          | bot
                          | inserisci(impiegato)
                          | commit
                          |
                          |
r(stipendi_impiegati)     |
avg(stipendi_impiegati)   |
commit                    |
```

In questo caso, nel corso della stessa esecuzione, T_1 ha calcolato due medie diverse a causa di un inserimento imprevisto da parte di T_2.

Riassumendo, abbiamo quindi le anomalie:

- **Perdita di aggiornamento** (W - W);
- **Lettura sporca** (R - W o W - W con abort);
- **Letture inconsistenti** (R - W);
- **Aggiornamento fantasma** (R - W);
- **Inserimento fantasma** (R - W nuovo dato).

Gestore della concorrenza

La risoluzione di queste concorrenze è operata da un gestore della concorrenza. Il gestore della concorrenza entra in gioco dopo altri due componenti, il **gestore dei metodi d'accesso**, che si occupa dell'autorizzazione a letture e scritture (read e write), e il **gestore delle transazioni**, che gestisce begin, commit e abort. Il gestore della concorrenza si occupa sempre di operazioni

read / write, ma nel contesto della loro esecuzione concorrente Per fare ciò ha l'ausilio della **tabella dei lock**.

Schedule

Una schedule (*"tabella di marcia"*) è una sequenza di operazioni lettura / scrittura di transazioni concorrenti, che compaiono nel loro ordine cronologico di esecuzione. Ad esempio, possiamo avere:

$$S : r_1(x) \quad r_2(z) \quad w_1(x) \quad w_2(z)$$

dove $r_1(x)$ e $w_1(x)$ sono letture e scrittura della transazione T_1 sull'oggetto x, e $r_2(z)$ e $w(z)$ sono letture e scritture della transazione T_2 sull'oggetto z.

Controllo di concorrenza

Il controllo di concorrenza, ovvero l'eliminazione di anomalie, è effettuato dallo **scheduler**, che tiene traccia di tutte le operazioni eseguite sulla base di dati e decide se accettare o rifiutare le transazioni che vengono via via richieste. Si assume, temporaneamente, che l'esito delle transazioni sia noto a priori (la cosiddettta ipotesi **commit-proiezione**), ergo lo stato finale di commit o abort sia noto prima dell'esecuzione della transazione. In questo modo possiamo automaticamente eliminare dallo schedule tutte le transazioni abortite. Si noti che questa assunzione non permette di trattare alcuni tipi di anomalie, tra cui quelle di lettura sporca.

Schedule seriali

Uno schedule di un'insieme di transazioni $T = T_1, ..., T_n$ è detto seriale se, per ogni coppia di transazioni $T_i, T_j \in T$, tutte le operazioni di T_i sono eseguite prima di qualsiasi operazione di T_j o viceversa. Una schedule si dice **serializzabile** se la sua esecuzione produce lo stesso risultato di uno schedule seriale sulle stesse transazioni. Si richiede quindi una nozione di equivalenza fra schedule. Abbiamo, intuitivamente, che nell'insieme di tutte le schedule S, l'insieme delle schedule serializzabili $S_s \in S$ è sottoinsieme di S, e l'insieme delle schedule seriali $S'_s \in S_s$ è a sua volta sottoinsieme di S_s.

View-serializzabilità

Diciamo che esiste una relazione **legge-da** tra le operazioni $r_i(x)$ e $w_j(x)$ in uno schedule S se $w_j(x)$ precede $r_i(x)$ in S e non c'è nessun $w_k(x)$ con $k \neq j$ tra $r_i(x)$ e $w_j(x)$ in S. A questo punto la scrittura $w_j(x)$ in S è detta **scrittura finale** su x se è l'ultima scrittura su x presente in S. Due schedule S_i e S_j sono dette **view-equivalenti** (in simboli $S_i i \approx_v S_j$) se hanno la stessa relazione legge-da e le stesse scritture su ogni oggetto. Uno schedule S è poi **view-serializzabile** se è view-equivalente a un qualche schedule seriale. L'insieme degli schedule view-serializzabili si indica con VSR.
Possiamo riscrivere alcune delle anomalie incontrate attraverso il linguaggio delle schedule:

- **Perdita di aggiornamento** : $S = r_1(x)\, r_2(x)\, w_1(x)\, w_2(x)$
- **Lettura inconsistente** : $S = r_1(x)\, r_2(x)\, w_2(x)\, r_1(x)$
- **Aggiornamento fantasma** : $S = r_1(x)\, r_1(y)\, r_2(z)\, r_2(y)\, w_2(y)\, w_2(z)\, r_1(z)$

Ovviamente, le ultime tre schedule riportate non sono view-equivalenti a nessuno schedule seriale. Sono anomalie! Facciamo le considerazioni del caso sulla complessità view-serializzabilità. La verifica della view-equivalenza di due dati schedule ha complessità polinomiale. Ergo decidere la view-serializzabilità di uno schedule è un problema NP-completo: occorre confrontare lo schedule con tutti i possibili schedule seriali. Questo non è concepibile nella pratica. La soluzione è quella di definire una condizione di equivalenza più ristretta, che non copra tutti i casi ma che sia utilizzabile nella pratica, cioò abbia complessità inferiore.

Conflict-serializzabilità

Un'operazione a_i è in **conflitto** con un'altra operazione a_j, con $i \neq j$, se operano sullo stesso oggetto e almeno una di esse è una scrittura. N.B.: in un conflitto conta l'ordine delle operazioni. Possiamo dividere i conflitti in due casi:

- **conflitto read-write** (R - W o W - R);
- **conflitto write-write** (W - W).

Due schedule sono dette **conflict-equivalenti**, $S_i \approx_c S_j$ se hanno le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi. Uno schedule S è **conflict-serializzabile** se è conflict-equivalente a un qualche schedule seriale. L'insieme degli schedule conflict-serializzabili è indicato con CSR.

VSR e CSR

Si può dimostrare che ogni schedule conflict-serializzabile è view-serializzabile, ma non necessariamente viceversa. Ad esempio, $r_1(x)w_2(x)w_1(x)w_3(x)$ è view-serializzabile, in quanto view-equivalente a $r_1(x)w_1(x)w_2(x)w_3(x)$, ma non è conflict-serializzabile. Dimostrare questo teorema significa dimostrare che la conflict-equivalenza ($\approx_v$) implica la view-equivalenza ($\approx_c$). Supponiamo quindi $S_1 \approx_c S_2$ I due schedule hanno:

- **Stesse scritture finali**: se così non fosse, ci sarebbero almeno due scritture in ordine diverso, e poichè scritture in ordine diverso sono in conflitto le schedule non sarebbero $\approx_c$.

- **Stesse relazioni "legge-da"**: se così non fosse, ci sarebbero scritture in ordine diverso p coppie lettura-scritture in ordine diverso e quindi, come sopra, sarebbe violata la $\approx_c$.

Abbiamo quindi introdotto due nuovi insiemi: nell'insieme di tutte le schedule S, $VSR \in S$ è l'insieme delle schedule view-serializzabili ed è sottoinsieme di S, $CSR \in VSR$ è l'insieme delle schedule conflict-serializzabili ed è sottoinsieme di VSR. Le schedule seriali $S'_s \in VSR$ sono sottoinsieme di VSR e CSR. Sia VSR che CSR sono a loro volta sottoinsieme delle schedule serializzabili S_s.

Verifica della conflict-serializzabilità

Vediamo quindi il perché dell'introduzione della conflict-serializzabilità: possiamo verificarla attraverso la costruzione di un **grafo dei conflitti**, contenente un nodo per ogni transazione T_i e un'arco orientato da T_i a T_j se c'é almeno un conflitto fra un'azione a_i e un'azione a_j tale che a_i precede a_j Uno schedule è CSR se e solo se il grafo è aciclico. In verità, anche la conflict-serializzabilità, sebbene sia verificabile in modo molto più rapido (con opportune strutture dati in tempo lineare), è inutilizzabile in pratica. Questo perché la tecnica richiede di conoscere il grafo dall'inizio, ma uno scheduler deve lavorare incrementalmente, ovvero rispondere in modo dinamico ad ogni richiesta in arrivo. Inoltre, la tecnica si basa sull'assunzione della commit-proiezione. Nella pratica, si usano tecniche che garantiscono la conflict-serializzabilità senza dover costruire grafi a priori, e senza assumere l'ipotesi della commit-proiezione.

Lock

Una tecnica è quella di precedere ogni lettura e scrittura da un lock, e di effettuare l'unlock solo dopo che vengono ultimate. Il lock manager effettua quetste operazioni e decide se accogliere o rifiutare ulteriori richieste sulla base dei lock già applicati. Per aumentare la concorrenza si possono adoperare lock di tipo diverso:

- **Lock condiviso**: un lock condiviso permette a più transazioni di accedere alla risorsa, senza che nessun'altra transazione possa scrivere sulla risora. Tutte le letture sono precedute da un r_lock (lock condiviso) e seguite dall'unlock;

- **Lock esclusivo**: un lock esclusivo assicura l'accesso alla risorsa a una sola transazione, che potrà quindi scrivere senza impattare altre transazioni. Tutte le scritture sono precedute da un w_lock (lock esclusivo) e seguite dall'unlock.

Una transazione che legge e poi scrive un'oggetto può richiedere subito un lock esclusivo, oppure può chiedere un lock condiviso e poi un lock esclusivo

(*lock escalation* o *lock promotion*). Il lock manager riceve queste richieste dalle transazioni e le accoglie o le rifiuta, sulla base della tavola dei conflitti. Quando si concede un lock su una risorsa ad una transazione, si dice che la risorsa è **acquisita** dalla transazione. Quando si effettua l'unlock, si dice che la risorsa viene **rilasciata**. Un lock condiviso impedisce la creazione di lock esclusivi (almeno che il lock condiviso sia unico). Di contro, un lock esclusivo impedisce la creazione di altri lock esclusivi o inclusivi. Un lock condiviso viene rilasciato quando il contatore dei lock scende a 0. Nel caso una transazione richieda una risorsa che non sia concessa, essa è messa in attesa (eventualmente in coda) finché la risorsa non diventa disponibile.

Lock a due fasi ($2PL$)
Un'algoritmo di scheduling usato dalla maggior parte dei sistemi commerciali è basato su due regole:

- Se una transazione vuole leggere (scrivere) un dato, prima deve acquisire un lock condiviso (esclusivo) su quel dato. Se la transazione entra in conflitto su un lock, la si pone in attesa.
- Una transazione, dopo aver rilasciato un lock, non può acquisirne un'altro.

Il meccanismo della 2PL significa che tutti i lock vanno ottenuti sequenzialmente, e rilasciati sequenzialmente. Non è assolutamente permesso rilasciare un lock per ottenerne un'altro.

2PL e CSR
Ogni schedule $2PL$ è anche conflict-serializzabile, ma non necessariamente viceversa. Ad esempio, la schedule $r_1(x)w_1(x)r_2(x)w_2(x)r_3(y)w_1(y)$ viola il $2PL$, ma è conflict-serializzabile. Questo teorema si può dimostrare. Come? cazzi tui Si ha, in definitiva, che le schedule $2PL$ formano un'ulteriore sottoinsieme, per la presciisione delle CSR, e contengono le schedule seriali S'_s. Riassumendo, si ha:

$$S'_s \subset 2PL \subset CSR \subset VSR \subset S$$

2PL e anomalie
La 2PL risolve le anomalie di perdita di aggiornamento, aggiornamento fantasma e letture inconsistente. Presenta pero altre anomalie:

- **Cascading rollback**
  Il fallimento di una transazione che ha scritto una risorsa deve causare il fallimento di tutte le transazioni che hanno letto il valore scritto.

- **Deadlock** (attese incrociate o stallo)
  Due transazioni possono detenere una risorsa e aspettare la risorsa detenuta dall'altra. In generale, la possibilità di deadlock è bassa, ma non nulla.

Locking a due fasi stretto
Una variante del lock a due fasi prevede una condizione aggiuntive, quella di fornire il rilascio dei lock solo dopo il commit. Questo elimina il rischio di letture sporche e quindi di rollback in cascata, e supera l'ipotesi di commit-proiezione.

47 Gestione dell'affidabilità

Il gestore dell'affidabilità si occupa dell'esecuzione dei comandi transazionali, assicurando le loro caratteristiche acide. I **comandi transazionali** sono:

- Start transaction (B, *begin*);
- Commit work (C, *commit*);
- Rollback work (A, *abort*).

Il gestore si occupa anche delle operazioni di **ripristino** (*recovery*) dopo eventuali guasti:

- **Ripresa a caldo** (*warm restart*);
- **Ripresa a freddo** (*cold restart*).

Il gestore dell'affidabilità mantiene un **log** delle transazioni: un'archivio permanente che registra le operazioni svolte. Il gestore dell'affidabilità lavora insieme al gestore dei metodi d'accesso (che gestisce i metodi di sistema *fix* e *unfix*), e al gestore delle transazioni.

Persistenza delle memorie
Ricordiamo brevemente il concetto di persistenza di una memoria. Riguardo alle memorie di interesse nel nostro caso, si ha che:

- **Memoria centrale:** è persistente, l'informazione viene però distrutta da qualsiasi guasto di sistema;
- **Memoria di massa:** è persistente e sopravvive ai guasti di sistema, ma può comunque danneggiarsi;

- **Memoria stabile:** astrazione che rappresenta una memoria persistente non danneggiabile, che viene perseguita nella realtà attraverso la ridondanza:
  - Dischi replicati;
  - Nastri magnetici;
  - ecc...

Log

Il log è un file **sequenziale** (quindi non ad accesso casuale) gestito dal gestore dell'affidabilità, scritto in memoria stabile. Riporta tutta le operazioni svolte sulla base di dati, nell'ordine in cui vengono svolte. E' formato da **record**:

- **Operazioni delle transazioni**
  - begin, B(T);
  - insert, I(T, O, AS) (nel record O, T memorizza per la prima volta l**after state** AS);
  - delete, D(T, O, BS) (nel record O, T elimina il **before state** BS);
  - update, U(T, O, BS, AS) (nel record O, T elimina il before state BS e memorizza l'after state AS);
  - commit, C(T);
  - abort, A(T).
- **Record di sistema**:
  - dump;
  - checkpoint.

Queste ultime due transazioni servono a fare regisitrazioni dell'intero stato di un database (*dump*) o delle sue transazioni (*checkpoint*).

Checkpoint e dump

Il log ha la funzione di permettere la ricostruzione delle operazioni. Checkpoint e dump forniscono stadi intermedi di ricostruzione, da utilizzare in caso di guasti.

- **Checkpoint**
  Il checkpoint registra quali transazioni sono attive in un dato istante temporale, cioè "a metà strada". Ha lo scopo di confermare che altre transazioni o non sono ancora iniziate, sono già finite. Per tutte le transazioni che hanno effettuate il commit i dati possono essere trasferiti

in memoria di massa. Esistono più modalità di checkpoint, di cui ne vedremo una:

  - Si sospende l'accettazione di operazioni di commit e abort finché non è completata la registrazione;
  - Si forza la scrittura in memoria di massa delle pagine in memoria modificate da transazioni che hanno già fatto commit;
  - Si forza la scrittura nel log di un record contenente gli identificatori di tutte le transazioni attive;
  - Si ricomincia ad accettare tutte le operazioni da parte delle transazioni.

  Con questo meccanismo si assicura la persistenza delle transazioni che hanno eseguito il commit.

- **Dump**
  Il dump è una copia completa della base di dati (un *backup*). Solitamente viene prodotta mentre il sistema è inattivo, salvato in memoria stabile, e registrato in un record di dump nel log che indica il momento in cui il dump è stato effettuato, vari dettagli pratici, ecc...

Esito di una transazione
Vediamo cosa succede ad una transazione quando si verifica un guasto. L'esito di una transazione è determinato irrevocabilmente quando viene scritto il record di commit nel log in modo **sincrono** e forzato. Un guasto prima di tale istante porta ad un UNDO di tutte le azioni, e ad una ricostruzione dello stato originario (prima della transazione) della base di dati. Un guasto dopo tale istante non deve avere conseguenze sugli effetti di tale transazioni: lo stato finale deve essere ricostruito, se necessario con un REDO. I record di abort possono essere invece scritti in modo **asincrono**.

Regole di modifica del log
Esistono due regole per la scrittura nel log:

- **Write-Ahead**
  Si scrive la parte BS (before state) del log prima di effettuare l'operazione sulla base di dati. Questo consente di disfare le azioni già memorizzate (UNDO) di transazioni senza commit avendo in memoria stabile un valore corretto.

- **Commit-Precedenza**
  Si scrive la parte AS (after state) dei record di log prima di commit.

Consente di rifare le azioni (REDO) di una transazione che effettuato il commit ma le cui pagine modificate non sono ancora state trascritte in memoria di massa.

Operazioni UNDO e REDO
Vediamo nel dettaglio le operazioni UNDO e REDO:

- **UNDO**
  L'UNDO (disfacimento) di un azione è, per le operazioni viste su un oggetto O:
  - update, delte: copiare il valore del BS (before state) in O;
  - insert: eliminare O.
- **REDO**
  Il REDO (rifacimento) di un'azione è, per le operazioni viste su un oggetto O:
  - insert, update: copiare il valore dell'AS (after state) in O;
  - delete: eliminare O.

Valgono le **idempotenze**:

$$\text{undo(undo(A))} = \text{undo(A)}$$

$$\text{redo(redo(A))} = \text{redo(A)}$$

Cioè, l'annullamento o il rifacimento di un'operazione A n volte è uguale all'annullamento o il rifacimento una singola volta.

Modalità di inserzione in log
Esistono più modalità di trascrizione delle operazioni sul log:

- **Modalità immediata**
  La modalità immediata comporta la trascrizione immediata di tutte le operazioni di scrittura provenienti da transazioni uncommitted (che non hanno ancora fatto commit). Richiede un UNDO al momento del guasto, ma non richiede REDO nel caso di esito con successo.
- **Modalità differita**
  Nella modalità differita si eseguono prima tutte le update, e poi le operazioni di scrittura. Non esistono valor AS nel log che vengono da transazioni uncommitted. Non c'è bisogno di UNDO, in quanto non ci sono scritture prima del commit. Richiede però un REDO nel caso non si sappia l'esito di una transazione (dopo le update potrebbe comunque non essere stata eseguita).

- **Modalità mista**
  Nella modalità mista la scrittura può avvenire sia in immediata che in differita. Potrebbero essere necessarie sia UNDO che REDO.

Alcune considerazioni pratiche: la modalità differita non viene molto utilizzata in pratica. Questo perché tale modalità è molto più efficiente nel **recupero** (*recovery*), ma è complessivamente meno efficiente di una modalità in cui il gestore può scegliere liberamente quando scrivere in memoria secondaria. Visto che il caso naturale di una transizione è quello di un'esito di successo, si preferisce adottare una modalità che ottimizzi questo tipo di risultato.

Guasti

Esistono più tipi di guasti:

- **Guasti "soft"**
  I guasti soft (*morbidi*) sono errori di programma, crash di sistema, cadute di tensione, ecc... Si perdono i contenuti della memoria centrale, ma non si perde né la memoria secondaria, cioè il database, né la memoria stabile, cioè il log. Si può fare una ripresa a caldo, (*warm start*).

- **Guasti "hard"**
  I guasti hard (*duri*) riguardano la memoria secondaria, ma non la memoria stabile. Se si perde la base di dati, bisogna fare una ripresa a freddo (*cold restart*), ricostruendo la base dai log. Nel caso di perdita della memoria stabile, cioè del log, ogni tentativo di recupero è impossibile. Si ricomincia da capo.

Modello di funzionamento fail-stop

L'individuazione di un guasto forza l'arresto completo di tutte le transazioni in corso. Quindi si riavvia il sistema, e si effettua una procedura di **restart**, al termine della quale, se si ha successo, si riavvia nuovamente, in caso contrario il **buffer** è vuoto, ma le transazioni possono ricominciare.

Procedura di restart

Il processo di restart ha l'obbiettivo di classificare le transazioni in:

- **Completate**: ergo tutte in memoria stabile;

- **In commit ma non necessariamente completate**: può servire un REDO;

- **Senza commit**: vanno annullate, ergo serve un UNDO.

Il gestore dell'affidabilità, al restart di sistema:

- Legge su un file di RESTART (nel log) l'indirizzo dell'ultimo checkpoint;
- Prepara due file: un UNDO list con gli identificatori delle transazioni attive, e un REDO list vuoto;
- Si assicura che nessun'utente si attivo.

Ripresa a caldo
La ripresa a caldo avviene in quattro fasi:

- Si trova l'ultimo checkpoint (ripercorrendo il log a ritroso);
- Si costruisce gli insiemi UNDO (transazioni attive ma non committed prima del guasto, da disfare) e REDO (transazioni committed tra il CK e il guasto, da rifare).
- Si ripercorre il log all'indietro (**rollback**) fino alla più vecchia azione delle transazioni in UNDO e REDO, disfacendo tutte le azioni delle transazioni in UNDO;
- Si ripercorre il log in avanti (**rollforward**) rifacendo tutte le azioni delle transazioni in redo.

Ripresa a freddo
La ripresa a freddo avviene in due fasi, più la ripresa a caldo:

- Ci si riporta al record di dump più recente nel log e si ripristina la parte di dati deteriorata;
- Si eseguono le operazioni registrate sul log sulla parte deteriorata fino all'istante del caldo;
- Si esegue una ripresa a caldo.