Basi di Dati

Introduzione ai DBMS

Definizioni
Dato: elemento di informazione costituito da simboli che devono essere elaborati.
Informazione: notizia, o elemento che consente di avere conoscenza più o meno esatta di fatti, situazioni e modi d'essere.
Sistema Informativo (SI): una componente di un'organizzazione il cui scopo è quello di gestire le informazioni utili ai fini dell'organizzazione stessa. L'esistenza di un SI è indipendente dalla sua automatizzazione (non è necessariamente implicata la presenza di computer).
Sistema Informatico: porzione automatizzata di un Sistema Informativo, nel quale le informazioni sono codificate e rappresentate dai dati.
DBMS: sistema software in grado di gestire collezioni di dati grandi, condivise e persistenti, in maniera efficiente e sicura.

Approccio di gestione dei dati
Necessità di gestire i dati in maniera persistente (dati salvati in memoria)
- Approccio convenzionale (basato su *files* - problemi: gestione di grandi quantità di dati, *condivisione* ed *accesso concorrente*)
- Approccio strutturato (basato su *DBMS*: è possibile implementare un paradigma di separazione tra dati ed applicazioni, le applicazioni necessitano solo di conoscere la struttura logica dei dati)

Caratteristiche DBMS
Efficienza: i DBMS forniscono adeguate strutture dati per organizzare i dati all'interno dei file, e per supportare le operazioni di ricerca/aggiornamento (in genere tabelle hash). Si utilizzano *indici*, strutture che contengono informazioni sulla posizione di memorizzazione delle tuple in base al valore del campo chiave, per avere accesso diretto ai dati.
L'ottimizzazione delle operazioni di ricerca (query) avviene partendo da un programma SQL, al quale viene applicata un'analisi sintattica e lessicale, al risultato si applica un'ottimizzazione algebrica (in cui si traduce la query SQL in una sequenza di operatori algebrici per l'accesso ai dati), si esegue poi un ottimizzazione basata sul modello dei costi e infine di accede ai dati.
TIP: per realizzare gli indici sono utilizzati alberi dinamici di tipo B-Tree che garantiscono costi di ricerca, inserimento e cancellazione in ordine O(log(n)).

Concorrenza: i DBMS devono fornire la possibilità di effettuare operazioni di accesso ai dati in concorrenza e al tempo stesso garantire il fatto che accessi da parte di applicazioni diverse non interferiscano tra loro (mantenendo la consistenza dei dati).

Il *Lock Manager* è una componente del DBMS responsabile di gestire i lock alle risorse del DB, e di rispondere alle richieste delle transazioni (vi sono due algoritmi per gestire l'ordine di acquisizione dei lock: 2FL e S2FL).
Affidabilità: tramite meccanismi di *roll-back* è possibile eseguire gruppi di operazioni in modo atomico, ovvero o viene completato tutto o niente (es. una transazione bancaria). Inoltre tramite un *log* (e algoritmi appositi come quelli di ripresa a caldo/freddo) è possibile ripristinare lo stato dei dati in caso di improvviso malfunzionamento.
Le proprietà ACID di un sistema transazionale sono: **A**tomicity: la transazione deve essere eseguita con la regola del "tutto o niente" - **C**onsistency: la transazione deve lasciare il DB in uno stato consistente, vincoli di integrità sui dati non devono essere violati - **I**solation: l'esecuzione di una transazione deve essere indipendente dalle altre - **D**urability (persistenza): l'effetto di una transazione conclusa con successo non deve essere perso.

Sicurezza: tramite politiche di controllo degli accessi e un sistema di permessi (con un alto livello di granularità) si garantisce un buon livello di sicurezza nel DBMS.

Scalabilità: possibilità di gestire grandi moli di dati aumentando il numero nodi usati per lo storage (database distribuiti). Tramite una politica di *sharding* si determina la politica di distribuzione dei dati tra i nodi. Vi sono anche meccanismi di load-balancing (evitare che un nodo abbia un carico troppo elevato) e gestione delle repliche.

<u>Architettura a tre livelli</u>
Un DBMS può essere visto come un'architettura software a 3 livelli:
- livello *esterno* (descrive come i dati appaiono per un utente o un gruppo di utenti)
- livello *logico* (descrive l'organizzazione logica dei dati)
- livello *fisico* (come e dove sono memorizzati i dati in memoria secondaria)
TIP: un esempio di modello logico è quello relazionale, nel quale i dati sono organizzati in tabelle.

Il modello relazionale

Il *modello logico* rappresenta un insieme di concetti per strutturare/organizzare i dati relativi ad un certo dominio d'interesse e contiene un insieme di regole, indipendenti dal dominio d'interesse, per modellare eventuali vincoli e restrizioni sui dati (business rules).

In generale si vuole interagire con il modello logico in modo indipendente dallo schema fisico, e si vuole interagire con il livello esterno in modo indipendente dallo schema logico dei dati.

Nel *modello relazionale* i dati sono organizzati in record (variabile composta da più campi) di dimensione fissa, e divisi in tabelle (relazioni). Le righe sono istanze della relazione e le colonne sono attributi.

Vincoli
Sull'ordine dei dati:
l'ordinamento delle righe e delle colonne è irrilevante.
Sui dati:
non possono esistere attributi uguali, non possono esistere righe uguali, i dati di una colonna devono essere omogenei (tutta la colonna di un solo tipo di dato).
È possibile avere uno schema di relazioni senza istanze (alla creazione ad esempio) ed ogni attributo dispone di un dominio (tipo di dato consentito all'interno di una colonna) che definisce l'insieme di valori validi per quell'attributo.

Prima Forma Normale
Una relazione si dice in *Prima Forma Normale (PFN)* se tutti gli attributi sono definiti su domini atomici e non su domini complessi (non si può avere ad esempio il tipo 'oggetto').

Def.: Dati n insiemi D_1,..., D_n una *relazione matematica* sugli insiemi D_1,..., D_n è definita come un sottoinsieme del prodotto cartesiano D_1 x ... x D_n
Def.: Il *prodotto cartesiano* di 2 insiemi A e B è l'insieme delle coppie ordinate (a, b) per ogni a in A e b in B.

Relazioni
Nel modello relazionale la relazione matematica non vale in quanto in quest'ultima non vale la proprietà commutativa, mentre se inverto due colonne in una tabella la tabella è la stessa. Si introduce quindi il concetto di ennupla per risolvere il problema.
Una *ennupla* su un insieme di attributi X è una funzione che associa a ciascun attributo A in X un valore del dominio di A: t[A] indica il valore della ennupla t sull'attributo A (es. val1[Campo1] = 123; val1[Campo2] = "Mario"; val2 [Campo1] = 321).
TIP: quando si parla di una relazione si fa riferimento a una tabella.

In una relazione, le ennuple di dati devono essere omogenee (ossia avere tutte la stessa struttura). In caso di mancanza di valore in un campo è possibile immettere un valore di default (valore speciale), ma potrebbe creare problemi tra i tipi, quindi si utilizza il valore *NULL* per tutti i tipi di dato.
E' però necessario limitare il numero di valori NULL aumentando il trade-off tra numero di tabelle e valori NULL. Si aumenta il numero di tabelle in modo che in

caso di NULL vi sarà un NULL al puntatore alla nuova tabella, invece che tanti NULL quanti sono i campi della seconda tabella.
TIP: NULL != NULL

Un *vincolo* è una funzione booleana,che associa ad una istanza r di una base di dati definita su uno schema R = {$R_1(X_1)$, … , $R_K(X_K)$} un valore di verità (true/false).

I *vincoli di ennupla* esprimono condizioni su ciascuna ennupla (riga), considerata singolarmente (espressi tramite espressioni algebriche o booleane).
Es. voto ≥ 18 && voto ≤ 30

I *vincoli di chiave* devono valere su tutti i valori che compongono una colonna o una combinazione di colonne.

Chiavi e Super-chiavi
Una *superchiave* è un sottoinsieme K di attributi di una relazione che non contiene due ennuple distinte t1 e t2 tali che t1[K] = t2[K].
Una *chiave* è un insieme di attributi che consente di identificare in maniera univoca le ennuple di una relazione. Nello specifico una *chiave* di una relazione R è una *superchiave minimale* di R (ossia non esiste un'altra superchiave K' che sia contenuta in K).

In generale le chiavi dovrebbero essere definite a livello di schema e non di istanza (in fase di progettazione si indicano i campi che si sà già saranno univoci).
Le chiavi servono per accedere a ciascuna ennupla della base di dati in maniera univoca e per correlare dati tra relazioni differenti.

Una *chiave primaria* è una colonna o insieme di colonne che: sono una superchiave, sono una chiave e non sono presenti valori NULL. Inoltre ogni relazione deve disporre di una chiave primaria (in caso non ci sia si aggiunge un nuovo capo progressivo).

Vincoli Inter-relazionali
Collegamenti tra relazioni differenti sono espresse mediante valori comuni in attributi replicati.
Un importante *vincolo* sulle dipendenze tra relazioni è che ogni riga della tabella referenziante si collega al massimo ad una riga della tabella referenziata, sulla base dei valori comuni nell'attributo/negli attributi replicati.

Un *vincolo di integrità referenziale* ("foreign key") fra gli attributi X di una relazione R_1 e un'altra relazione R_2 impone ai valori (diversi da NULL) su X in R_1 di comparire come valori della chiave primaria di R_2. Consente in pratica di collegare le informazioni tra tabelle diverse attraverso valori comuni.

In caso di violazione di un vincolo di integrità su certe tabelle in seguito ad un'operazione di aggiornamento su altre si può: non consentire l'operazione, effettuare un'eliminazione a cascata o inserire valori NULL.

Riflessioni
I *pro* del modello relazionale sono l'indipendenza dal modello fisico e la riflessività (le meta-informazioni di una relazione sono gestite a loro volta attraverso relazioni). I *contro* sono la poca flessibilità (stessa struttura per ogni istanza di una relazione) e la ridondanza dei dati causata dai vincoli.

Il linguaggio SQL - SQL-DDL (Pt.1)

I due componenti principali del linguaggio SQL sono il *DDL* (Data Definition Language), che contiene i costrutti necessari per la creazione/modifica dello schema della base di dati. E il *DML* (Data Manipulation Language), che contiene i costrutti per le interrogazioni e di inserimento/eliminazione/modifica dei dati.

```
CREATE DATABASE DB_NAME AUTHORIZATION USER_NAME

CREATE TABLE TABLE_NAME (
        AttrName1 Domain [DefaultVal] [Constraints]
        ...
)
```

Con *dominio* si intende ad esempio character, numeric, decimal, integer, float, real, double, date, time, timestamp, ...

La keyword *auto_increment* permette di creare campi numerici che si auto-incrementano ad ogni nuovo inserimento nella tabella

I domini blob e cblob consentono di rappresentare oggetti di grandi dimensioni come sequenza di valori rispettivamente binari o di caratteri.

È possibile inoltre creare un proprio dominio: CREATE DOMAIN DOM_NAME AS SMALLINT DEFAULT 40.

Vincoli
Per ogni dominio o attributo è possibile definire vincoli che devono essere rispettati da tutte le istanze di quel dominio o attributo.
I vincoli *intra-relazionali* sono: generici di ennupla, not null, unique, primary key. Mentre quelli *inter-relazionali* sono references.

Tramite la clausola *CHECK* è possibile esprimere vincoli di ennupla arbitrari: VOTO SMALLINT CHECK((VOTO>=18) AND (VOTO<=30))

Con il vincolo *NOT NULL* si indica che il valore NULL non è ammesso:
NSTUDENTI SMALLINT NOT NULL
Il vincolo *UNIQUE* impone che l'attributo su cui è applicato non presenti valori comuni in righe differenti, ossia che tale attributo sia una superchiave della tabella:
CODICE SMALLINT UNIQUE oppure UNIQUE(CODICE, UFFICIO, …)
Nota: rendere unici un insieme di attributi vuole che sia unica la combinazione tra essi, non che ognuno sia unico.
Il vincolo *PRIMARY KEY* impone che l'attributo su cui si applica non presenti valori comuni in righe differenti e non assuma valori NULL, ossia che l'attributo sia chiave primaria. Tale vincolo può apparire al massimo una sola volta per tabella.
CODICE SMALLINT PRIMARY KEY oppure PRIMARY KEY (CODICE, UFFICIO,...)

I vincoli *REFERENCES* e *FOREIGN KEY* consentono di definire dei vincoli di integrità referenziale tra i valori di un attributo nella tabella in cui è definito (tabella interna) ed i valori di un attributo in una seconda tabella (tabella esterna). L'attributo cui si fa riferimento nella tabella esterna deve essere soggetto al vincolo UNIQUE.
CORSO VARCHAR(4) REFERENCES CORSI(CODICE) oppure
FOREIGN KEY(NOME, COGNOME, DATANASCITA) REFERENCES ANAGRAFICA(NOME, COGNOME, DATA)
È possibile associare azioni specifiche da eseguire sulla tabella interna in caso di violazioni del vincolo di integrità referenziale (ad esempio si elimina nella tabella esterna l'elemento a cui si fa riferimento):
ON (DELETE | UPDATE) (CASCADE | SET NULL | SET DEFAULT | NO ACTION)

Per modificare gli schemi di dati precedentemente creati si può usare *ALTER*, per eliminare uno schema invece *DROP*.

Il linguaggio SQL - SQL-DDL (Pt.2)

Le operazioni di interrogazione vengono implementate dal costrutto di *SELECT*.
SELECT Attributo_1,...,Attributo_M
FROM Tabella_1,...,Tabella_N
WHERE Condizione
La semantica è la seguente: effettua il prodotto cartesiano delle tabelle Tabella_1,...,Tabella_N. Da queste, estrai le righe che rispettano la Condizione. Di quest'ultime, preleva solo le colonne corrispondenti a Attributo_1,...,Attributo_M.

<u>Operatori nella clausola WHERE</u>
LIKE: effettua confronti tra stringhe, es. (Nome LIKE 'M_R%O').

BETWEEN: permette di verificare l'appartenenza ad un certo insieme di valori, es. (Stipendio BETWEEN(24000, 34000)).
IS NULL e *IS NOT NULL*: controlla se il valore è nullo o meno restituendo true, false o unknown. es. (Stipendio IS NULL).

Costrutti
DISTINCT: dopo la keyword SELECT consente di rimuovere i duplicati nel risultato.
ALL: dopo la keyword SELECT non rimuove i duplicati (default).
ORDER BY Attributo: consente di ordinare le righe del risultato di un'interrogazione in base al valore di un attributo specificato (presente dopo la clausola WHERE).

Gli *operatori aggregati* si applicano a gruppi di tuple (e non tupla per tupla), e producono come risultato un solo valore. Sono inseriti nella SELECT e valutati dopo le clausole WHERE e FROM. Questi operatori sono COUNT, SUM, AVG, MIN, MAX.
Nota: SELECT CODICE, MAX(STIPENDIO) FROM STRUTTURATI WHERE (TIPO = "ASSOCIATO") non è corretta, in quanto MAX restituisce un solo valore e CODICE potrebbe restituirne molti.

L'*operatore di raggruppamento GROUPBY* consente di dividere la tabella in gruppi, ognuno caratterizzato da un valore comune dell'attributo specificato nell'operatore. Ogni gruppo produce una sola riga nel risultato finale.
SELECT ListaAttributi$_1$ FROM ListaTabelle WHERE Condizione GROUPBY ListaAttributi$_2$
Nota: ListaAttributi$_1$ deve essere un sottoinsieme di ListaAttributi$_2$ e può contenere operatori aggregati.

Il costrutto *HAVING* permette di filtrare i gruppi in base a determinate condizioni, tale clausola viene valutata su ciascun gruppo e contiene operatori aggregati o condizioni sugli attributi del GROUPBY.
SELECT ListaAttributi$_1$ FROM ListaTabelle WHERE Condizione GROUPBY ListaAttributi$_2$ HAVING Condizione

Costrutto SELECT in forma generale
SELECT ListaAttributi
FROM ListaTabelle
WHERE Condizione
LIMIT Numero
GROUPBY AttributiRaggruppamento
HAVING CondizioniGruppi
ORDERBY ListaAttributiOrdinamento

Operazioni insiemistiche
È possibile effettuare operazioni insiemistiche tra tabelle, o in generale tra risultati di SELECT, tramite UNION, INTERSECT, EXCEPT. Gli attributi della SELECT devono avere tipi di dato compatibili e (possibilmente) gli stessi nomi.
SELECT NOME, COGNOME FROM $TABELLA_1$
UNION / INTERSECT / EXCEPT
SELECT NOME, COGNOME FROM $TABELLA_2$

Altri comandi per la modifica dell'istanza del DB
INSERT (inserisce una o più righe), DELETE (cancella una o più righe) e UPDATE (aggiorna un attributo o più).

INSERT INTO IMPIEGATI(Codice, Nome, Cognome, Ufficio) VALUES ('8', 'Vittorio', 'Rossi', 'A')
DELETE FROM IMPIEGATI WHERE (UFFICIO="A")
UPDATE IMPIEGATI SET NOME="Mario" WHERE (CODICE=5)

Join tra tabelle
È possibile implementare il join tra tabelle in due modi:
1 - inserendo le condizioni del join direttamente nella clausola del WHERE
2 - attraverso l'utilizzo dell'operatore di INNER JOIN nella clausola FROM

1: SELECT Modello FROM GUIDATORI, VEICOLI WHERE (GUIDATORI.NrPatente = VEICOLI.NrPatente) AND (Nome="Sara")
2: SELECT Modello FROM GUIDATORI JOIN VEICOLI ON GUIDATORI.NrPatente = VEICOLI.NrPatente WHERE (Nome="Sara")

Altre varianti sono:
LEFT JOIN (equivalente al risultato dell'inner join piu le righe della tabella di sinistra che non hanno un corrispettivo a destra (completate con valori NULL).
SELECT ListaAttributi FROM Tabella LEFT JOIN Tabella ON CondizioneJoin [WHERE Condizione]
RIGHT JOIN (equivalente al risultato dell'inner join piu le righe della tabella di destra che non hanno un corrispettivo a sinistra (completate con valori NULL).
SELECT ListaAttributi FROM Tabella RIGHT JOIN Tabella ON CondizioneJoin [WHERE Condizione]
FULL JOIN (equivalente al risultato dell'inner join piu le righe della tabella di sinistra/destra che non hanno un corrispettivo a sinistra/destra (completate con valori NULL).
SELECT ListaAttributi FROM Tabella FULL JOIN Tabella ON CondizioneJoin [WHERE Condizione]

Interrogazioni annidate, Viste, Asserzioni

Esempio di query annidata: estrarre il codice dello strutturato che riceve lo stipendio più alto.
SELECT CODICE FROM STRUTTURATI WHERE (STIPENDIO = SELECT MAX(STIPENDIO) FROM STRUTTURATI)

Se però tuttavia la query annidata restituisce non un solo valore, ma un insieme, gli operatori di confronto (<,=,>) non sono più utilizzabili da soli. Si usano allora in aggiunta gli *operatori speciali* di confronto *ANY* (la riga soddisfa la condizione se è vero il confronto tra il valore dell'attributo e *almeno uno* dei valori ritornati dalla query annidata) e *ALL* (la riga soddisfa la condizione se è vero il confronto tra il valore dell'attributo e *tutti* i valori ritornati dalla query annidata).

Il costrutto *IN* restituisce true se un certo valore è contenuto nel risultato di una query annidata, e false altrimenti.
Il costrutto *EXISTS* restituisce true se la query annidata restituisce un risultato non vuoto (almeno 1 elemento trovato).

Le query annidate possono essere sia *semplici* (non c'è un binding tra un contesto e l'altro, si valuta prima quella più interna fino alla più esterna) che *complesse* (c'è un binding attraverso variabili condivise tra le varie query, le query più interne vengono valutate su ogni tupla).

Nome e cognome degli strutturati appartenenti al dipartimento di informatica che hanno uno stipendio maggiore di tutti quelli appartenenti al dipartimento di fisica.
SELECT NOME, COGNOME
FROM STRUTTURATI
WHERE (DIPARTIMENTO = "INFORMATICA")
AND (STIPENDIO > ALL (SELECT STIPENDIO FROM STRUTTURATI WHERE (DIPARTIMENTO = "FISICA")))

Nome e cognome degli impiegati che hanno omonimi.
SELECT NOME, COGNOME
FROM IMPIEGATI AS I
WHERE (I.NOME, I.COGNOME) = (SELECT NOME, COGNOME
FROM IMPIEGATI AS I2
WHERE (I.NOME = I2.NOME) AND (I.COGNOME = I2.COGNOME) AND (I.CODICE <> I2.CODICE))

In questo caso la query annidata può essere riscritta utilizzando il join tra tabelle:

```
SELECT NOME, COGNOME
FROM IMPIEGATI AS I,IMPIEGATI AS I2
WHERE (I.NOME=I2.NOME) AND (I.COGNOME=I2.COGNOME) AND
(I.CODICE <>
I2.CODICE))
```

Viste

Le viste rappresentano "*tabelle virtuali*" ottenute da dati contenuti in altre tabelle del database. Ogni vista ha associato un nome ed una lista di attributi, e si ottiene dal risultato di una SELECT.
CREATE VIEW NomeView [ListaAttributi] AS SELECTSQL [with [local | cascade] check option]
Esempio: CREATE VIEW
STUDENTI(CODICE,NOME,COGNOME,DATANASCITA) AS SELECT
CODICE,NOME,COGNOME,NASCITA FROM PROFESSORI

I dati delle viste (ad eccezione delle viste materializzate) *non* sono fisicamente memorizzati a parte, in quanto dipendono da altre tabelle. Le viste esistono a livello di schema ma non hanno istanze proprie.

Con le viste si vuole:
- Implementare meccanismi di indipendenza tra il livello logico ed il livello esterno
- Scrivere interrogazioni complesse, semplificandone la sintassi
- Garantire la retro-compatibilità con precedenti versioni dello schema del DB, in caso di ristrutturazione dello stesso

Una vista può essere costruita a partire da altre viste dello schema (viste derivate), l'aggiornamento delle viste derivate è possibile tramite WITH LOCAL CHECK OPTION (il controllo di validità si limita alla vista corrente) o WITH CASCADE CHECK OPTION (il controllo di validità si estende ricorsivamente a tutte le viste da cui la corrente è derivata).

Le Common Table Expression (CTE) rappresentano viste temporanee che possono essere usate in una query come se fossero una vista a tutti gli effetti. Una CTE non
esiste a livello di schema del DB.
WITH NAME(Attributi) AS SQLQuery (includendo VIEW nella definizione)

Asserzioni (SQL2)

Le asserzioni sono un costrutto per definire vincoli generici a livello di schema. Consentono di definire vincoli non altrimenti definibili con i costrutti visti fin ora, il vincolo può essere immediato o differito (ovvero verificato al termine di una transazione).

CREATE ASSERTION NomeAsserzione CHECK Condizione
Esempio: CREATE ASSERTION VotoValido CHECK (Voto IS NOT NULL AND (VOTO >= 18) AND (Voto <= 30))

Costrutti avanzati SQL2 ed SQL3

Stored procedures

Le stored procedures sono frammenti di codice SQL con la possibilità di specificare un nome, dei parametri in ingresso e dei valori di ritorno.

```
Procedure ModificaStipendio (MatricolaN:varchar(20), StipendioNew:smallint)
        UPDATE Impiegati SET Stipendio = StipendioNew
        WHERE Matricola = MatricolaN
```

Le estensioni procedurali consentono di:
- Aggiungere strutture di controllo al linguaggio SQL (cicli, if-else,...).
- Dichiarare variabili e tipi di dato user-defined.
- Definire funzioni avanzate ed ottimizzate, che sono ritenute “sicure” dal DBMS.

Triggers

I trigger sono meccanismi di gestione della base di dati basati sul paradigma ECA (Evento / Condizione / Azione). *Evento*: primitive per la manipolazione dei dati (INSERT, DELETE, UPDATE), *condizione*: predicato booleano e *azione*: sequenza di istruzioni SQL, talvolta procedure SQL specifiche del DBMS.

I trigger servono a *garantire* il soddisfacimento di *vincoli di integrità referenziale*, e/o specificare meccanismi di reazione ad hoc in caso di violazione dei vincoli. Servono inoltre a *specificare regole aziendali* (business rules), ossia vincoli generici sullo schema della base di dati.
Sintassi:

```
Create trigger Nome
Modo Evento on Tabella
[referencing Referenza]
[for each Livello]
[when (IstruzioneSQL)]
Istruzione/ProceduraSQL
```

Modo: BEFORE/AFTER
Evento: INSERT/DELETE/UPDATE
Referenza: eventuali variabili globali
Livello: ROW (trigger agisce a livello di righe) / STATEMENT (tigger agisce a livello di tabella).
Le modalità di esecuzione sono o *immediata* o *differita*.

Permessi
Il comando *GRANT* consente di assegnare privilegi su una certa risorsa ad utenti specifici.
GRANT Privilegio ON Risorsa/e TO Utente/i [with grant option]

Il comando *REVOKE* consente di revocare privilegi su una certa risorsa ad utenti specifici.
REVOKE Privilegio ON Risorsa/e FROM Utente/i [cascade|restrict]

È possibile inoltre definire ruoli (contenitori di privilegi) per l'accesso alle risorse di un database tramite i comandi *CREATE ROLE / SET ROLE*.

Gestore delle transazioni

Le transazioni rappresentano unità di lavoro elementare (insiemi di istruzioni SQL) che modificano il contenuto di una base di dati.
START TRANSACTION
UPDATE SalariImpiegati
SET conto = conto * 1.2
WHERE (CodiceImpiegato = 123)
if conto > 0
commit work;
else rollback work

Le proprietà ACID delle transazioni:
- *Atomicità* (la transazione deve essere eseguita con la regola "tutto o niente")
- *Consistenza* (la transazione deve lasciare il DB in uno stato consistente)
- *Isolamento* (l'esecuzione di una transazione deve essere indipendente dalle altre)
- *Persistenza* (l'effetto di una transazione che ha fatto commit work non deve essere perso)

Il *gestore dell'affidabilità* garantisce *atomicità* e *persistenza* tramite l'utilizzo di log e checkpoint.
Il *gestore della concorrenza* garantisce l'*isolamento* in caso di esecuzione concorrente di più transazioni.

Gestore della concorrenza
Dato un *insieme di transazioni* $T_1, T_2, \dots, T_n$ di cui ciascuna formata da un certo insieme di operazioni di scrittura (w_i) e lettura (r_i), si definisce *schedule* la sequenza di operazioni di lettura/scrittura di tutte le transazioni così come eseguite sulla base di dati: $r_1(x)$, $r_2(y)$, $r_1(y)$, $w_4(y)$, $w_2(z)$, …

Uno schedule S si dice *seriale* se le azioni di ciascuna transazione appaiono in sequenza, senza essere inframezzate da azioni di altre transazioni.
$S = \{T_1, T_2, \dots, T_n\}$
Lo schedule seriale è ottenibile se le transazioni sono eseguite una alla volta (scenario irrealistico), o sono completamente indipendenti l'una dall'altra (scenario improbabile). Nella realtà le transazioni vengono eseguite in concorrenza (per maggiore efficienza e scalabilità).

Uno schedule S si dice *serializzabile* se produce lo stesso risultato di un qualunque schedule seriale S' delle stesse transazioni (gli schedule seriali sono un sottoinsieme degli schedule serializzabili).

I lock (controllo della concorrenza)
Grazie al meccanismo dei lock, prima di effettuare una qualsiasi operazione di lettura/scrittura su una risorsa, è necessario aver precedentemente acquisito il controllo (lock) sulla risorsa stessa. Vi sono lock in lettura (accesso condiviso) e lock in scrittura (mutua esclusione).
Su ogni lock possono essere definite due operazioni:
- *richiesta* del lock in lettura/scrittura
- rilascio del lock (*unlock*) acquisito in precedenza

Il lock manager è un componente del DBMS responsabile di gestire i lock alle risorse del DB, e di rispondere alle richieste delle transazioni. Per ciascun oggetto del DBMS conserva lo stato (libero, lock_r, lock_w), la lista delle transazioni attive e la lista delle transazioni bloccate.

2PL
Two-Phase Lock: *una transazione dopo aver rilasciato un lock, non può acquisirne un altro.* Questo costringe una transazione ad acquisire prima tutti i lock delle risorse di cui necessita, utilizzare gli oggetti e poi man mano rilasciare tutti i lock. Ogni schedule che rispetta il 2PL è anche serializzabile, in quanto produce lo stesso risultato di un qualunque altro schedule con le stesse transazioni. Tuttavia possono esservi casi di lettura sporca ad esempio se T_1 prende il lock di scrittura, T_2 prende il lock in lettura, T_1 scrive sull'oggetto e T_2 legge l'oggetto.

S2PL
Strict Two-Phase Lock: *i lock di una transazione vengono rilasciati solo dopo aver effettuato le operazioni di commit / abort.* Questo protocollo garantisce l'assenza di problemi di lettura sporca.

Deadlocks
Entrambi i protocolli sopra possono generare schedule con situazioni di deadlock. Per gestire queste situazioni si possono usare:

- *Timeout* (ogni operazione di una transazione ha un timer entro il quale deve essere completata, pena l'annullamento (abort) della transazione stessa)
- *Deadlock avoidance* (si cerca di prevenire le configurazioni che potrebbero creare deadlocks tramite il lock/unlock di tutte le risorse allo stesso tempo o l'utilizzo di time-stamp o classi di priorità tra transazioni)
- *Deadlock detection* (utilizzare algoritmi per identificare eventuali situazioni di deadlock e prevedere meccanismi di recovery)

TS

Time-Stamp: *ad ogni transazione si associa un timestamp che rappresenta il momento di inizio della transazione*, ogni transazione non può leggere o scrivere un dato scritto da una transazione con timestamp maggiore. Inoltre ogni transazione non può scrivere su un dato già letto da una transazione con timestamp maggiore.
Ad ogni oggetto x sono associati gli indicatori WTM(x), il timestamp della transazione che ha fatto l'ultima scrittura su x e RTM(x), il timestamp dell'ultima transazione (con t più alto) che ha letto x. Lo scheduler di sistema verifica se un'eventuale azione ($r_t(x)$ o $w_t(x)$) eseguita da una transazione T con timestamp t può essere eseguita o meno.

$r_t(x)$ - se t < WTM(x) allora la transazione viene uccisa. Se t >= WTM(x), la richiesta viene eseguita, ed RTM(x) viene aggiornato al massimo tra il valore precedente di RTM(x) e t stesso.
$w_t(x)$ - se t < WTM(x) oppure t < RTM(x) allora la transazione viene uccisa. Altrimenti, la richiesta viene accettata, e WTM(x) viene posto uguale a t.
Nota: le transazioni sono utilizzabili solo su tabelle di tipo InnoDB

Gestore dell'affidabilità

Il gestore dell'affidabilità verifica che siano garantite le proprietà di atomicità e persistenza delle transazioni. È responsabile di implementare i comandi START TRANSACTION, COMMIT E ROLLBACK. È responsabile di ripristinare il sistema dopo malfunzionamenti software (ripresa a caldo) e hardware (ripresa a freddo).
Questo viene effettuato grazie all'utilizzo di un *log* tramite il quale è possibile fare un do/undo delle transazioni. Questo elemento si presenta come un file sequenziale suddiviso in record:
- Record di transizione (tengono traccia delle operazioni svolte da ciascuna transazione sul DBMS)
- Record di sistema (tengono traccia delle operazioni di sistema: dump (traccia gli eventi di backup) e checkpoint (traccia le transazioni attive presenti nel sistema))

Checkpoint

Per prima cosa si sospendono tutte le operazioni in corso sul DBMS, si rendono effettive anche su memoria secondaria le operazioni effettuate da transazioni che hanno fatto commit, i cui dati si trovino ancora nel buffer (flush). Poi si scrivono nel record di checkpoint del log tutte le transazioni attive del sistema e infine si riprende l'esecuzione delle operazioni.

Regole di scrittura del log

Regola *Write Ahead Log*: la parte BS (before state) di ogni record di log deve essere scritta prima che la corrispondente operazione venga effettuata nella base di dati. In pratica i record di log devono essere scritti prima delle corrispondenti operazioni sulla base di dati, in modo da poter fare undo delle operazioni.

Regola di *Commit Precedence*: la parte AS (after state) di ogni record di log deve essere scritta nel log prima di effettuare il commit della transazione. I record di log devono essere scritti prima di effettuare l'operazione di commit, in modo da poter garantire il redo di una transazione non ancora scritta su memoria secondaria.

Ripresa a caldo

1. Si accede al log e lo si scorre fino all'ultimo record di checkpoint.
2. Si decidono le transazioni da annullare (undo) o rifare (redo): si costruiscono due insiemi: UNDO e REDO, il primo è inizializzato con la lista delle transazioni attive, il secondo è vuoto. Si aggiungono ad UNDO tutte le transazioni T di cui esiste un record B(T) [begin] e si spostano da UNDO a REDO tutte le transazioni di cui esiste un record C(T) [commit].
3. Per il set di UNDO si ripercorre il file il file di log disfacendo tutte le azioni effettuate da ogni transazione T contenuta nel set, dalla più recente alla meno recente.
4. Per il set di REDO si applicano le azioni effettuate da ogni transazione T contenuta nel set, dalla meno recente alla più recente.

Ripresa a freddo

1. Si copia il dump nel DB attuale (cercando di sovrascrivere solo la parte deteriorata).
2. Relativamente alla parte deteriorata si scorre il log dal dump in poi fino all'ultimo checkpoint, e si effettuano tutte le azioni delle transazioni concluse con un commit.
3. Si effettua la ripresa a caldo (come visto prima).

I sistemi NoSQL

Il NoSQL è un movimento che promuove l'adozione di DBMS non basati sul modello relazionale.

- *Big data*: moli di dati, eterogenee, destrutturate, difficili da gestire attraverso tecnologie tradizionali (come RDBMS).
- *Limitazione del modello relazionale*: il modello relazionale presuppone una rappresentazione tabellare, in SQL alcune operazioni non possono essere implementate e la scalabilità dei RDBMS tende a diminuire all'aumentare dei nodi.
- Il teorema di Brewer (*CAP Theorem*) afferma che un sistema distribuito può soddisfare al massimo due delle tre proprietà: *Consistency* (tutti i nodi della rete vedono gli stessi dati), *Availability* (il servizio è sempre disponibile) e *Partition Tolerance* (il servizio continua a funzionare correttamente anche in presenza di perdita di messaggi o di partizionamenti della rete).

Proprietà BASE:
- ***B****asically* ***A****vailable*: i nodi del sistema distribuito possono essere soggetti a guasti, ma il servizio è sempre disponibile.
- ***S****oft State*: la consistenza dei dati non è garantita in ogni istante.
- ***E****ventually Consistent*: il sistema diventa consistente dopo un certo intervallo di tempo, se le attività di modifica dei dati cessano.

Scalabilità: capacità di un sistema di migliorare le proprie prestazioni per un certo carico di lavoro, quando vengono aggiunte nuove risorse al sistema.
La *scalabilità verticale* è ad esempio aggiungere più potenza di calcolo (RAM, CPU) ai nodi che gestiscono il DB. La *scalabilità orizzontale* è ad esempio aggiungere più nodi al cluster.

I modelli logici dei DBMS NoSQL sono: chiave/valore, document-oriented (MongoDB), column-oriented o graph-oriented.

Progettazione di basi di dati

Il tutto si sviluppa in 6 fasi principali: lo studio della fattibilità, la raccolta/analisi dei requisiti, la progettazione, l'implementazione, la validazione e il funzionamento. Più in dettaglio la fase si *progettazione* si articola in tre ulteriori fasi: la progettazione *concettuale*, la progettazione *logica* e la progettazione *fisica*.

La raccolta/analisi dei requisiti consiste nella completa individuazione dei problemi che il sistema informativo da realizzare deve risolvere e le caratteristiche il sistema deve avere.

Progettazione concettuale: il suo scopo è quello di rappresentare le specifiche informali della realtà di interesse in termini dì una descrizione formale e completa, ma indipendente dai criteri di rappresentazione utilizzati nei sistemi di gestione di basi di dati. Il prodotto di questa fase viene chiamato schema concettuale e fa riferimento a un modello concettuale dei dati. I modelli concettuali ci consentono di descrivere l'organizzazione dei dati a un alto livello di astrazione, senza tenere conto degli aspetti implementativi. in questa fase infatti, il progettista deve cercare di rappresentare il contenuto informativo della base di dati, senza preoccuparsi né delle modalità con le quali queste informazioni verranno codificate in un sistema reale, né dell'efficienza dei programmi che faranno uso di queste informazioni.

Progettazione logica: consiste nella traduzione dello schema concettuale definito nella fase precedente, in termini del modello di rappresentazione dei dati adottato
dal sistema di gestione di base di dati a disposizione. Il prodotto di questa fase viene denominato schema logico della base di dati e fa riferimento a un modello logico dei dati. Come noto, un modello logico ci consente di descrivere i dati secondo una rappresentazione ancora indipendente da dettagli fisici, ma concreta perché disponibile nei sistemi di gestione di base di dati. In questa fase, le scelte
progettuali si basano, tra l'altro, su criteri di ottimizzazione delle operazioni da effettuare sui dati. Si fa comunemente uso anche di tecniche formali di verifica della qualità dello schema logico ottenuto. Nel caso del modello relazionale dei dati, la tecnica comunemente utilizzata è quella della normalizzazione.

Progettazione fisica: in questa fase lo schema logico viene completato con la specifica dei parametri fisici di memorizzazione dei dati (organizzazione dei file e degli indici). Il prodotto di questa fase viene denominato schema fisico e fa riferimento a un modello fisico dei dati. Tale modello dipende dallo specifico sistema di gestione di basi di dati scelto e si basa sui criteri di organizzazione fisica dei dati in quel sistema.

<u>Progettazione concettuale: Il diagramma E-R</u>
Entità: rappresentano classi di oggetti (per esempio, fatti, cose, persone) che hanno proprietà comuni ed esistenza "autonoma" ai fini dell'applicazione di interesse.

Relazioni (o associazioni): rappresentano legami logici, significativi per l'applicazione di interesse, tra due o più entità. L'insieme delle occorrenze di una relazione del modello E-R è, a tutti gli effetti, una relazione matematica tra le occorrenze delle entità coinvolte, ossia, è un sottoinsieme del loro prodotto cartesiano. Questo significa che *tra le occorrenze di una relazione* del modello E-R *non ci possono essere ennuple ripetute*.

Attributi: descrivono le proprietà elementari di entità o relazioni che sono di interesse ai fini dell'applicazione. Un attributo associa a ciascuna occorrenza di entità (o di relazione) un valore appartenente a un insieme, detto dominio, che contiene i valori ammissibili per l'attributo. Per esempio, l'attributo Cognome dell'entità Impiegato può avere come dominio l'insieme delle stringhe di 20 caratteri, mentre l'attributo Età può avere come dominio gli interi compresi tra 18 e 65. I domini non sono rappresentati nel modello ER ma nella documentazione.

Cardinalità delle relazioni: vengono specificate per ciascuna partecipazione di entità a una relazione e descrivono il numero minimo e massimo di occorrenze di relazione a cui una occorrenza dell'entità può partecipare. Dicono quindi quante volte, in una relazione tra entità, un'occorrenza di una di queste entità può essere legata a occorrenze delle altre entità coinvolte.

Cardinalità degli attributi: possono essere specificate per gli attributi di entità o relazioni e descrivono il numero minimo e massimo di valori dell'attributo associati
a ogni occorrenza di entità o relazione. Nella maggior parte dei casi, la cardinalità
di un attributo è pari a (1,1) e viene omessa, in questi casi l'attributo rappresenta sostanzialmente una funzione che associa a ogni occorrenza di entità un solo valore
dell'attributo.

Identificatori delle entità: vengono specificati per ciascuna entità di uno schema e descrivono i concetti (attributi e/o entità) dello schema che permettono di identificare in maniera univoca le occorrenze delle entità. In molti casi, uno o più attributi di un'entità sono sufficienti a individuare un identificatore: si parla in questo caso di
identificatore interno (detto anche chiave). Un'entità E può essere identificata da altre entità solo se tali entità sono coinvolte in una relazione a cui E partecipa con cardinalità (1,1). Nei casi in cui l'identificazione di un'entità è ottenuta utilizzando altre entità si parla di identificatore esterno.

Generalizzazioni: rappresentano legami logici tra un'entità E, detta entità genitore, e una o più entità $E_1...E_n$, dette entità figlie, di cui E è più generale, nel senso che le comprende come caso particolare. Si dice in questo caso che E è generalizzazione di $E_1...E_n$ e che le entità $E_1...E_n$ sono specializzazioni dell'entità E. Una generalizzazione è totale se ogni occorrenza dell'entità genitore è un'occorrenza di almeno una delle entità figlie, altrimenti è parziale. Una generalizzazione è esclusiva se ogni occorrenza dell'entità genitore è al più un'occorrenza di una delle entità figlie, altrimenti è sovrapposta.

Business Rules: uno strumento per la descrizione di proprietà di un'applicazione che non si riesce a rappresentare direttamente con modelli concettuali.
Vincoli di integrità: <concetto> deve/non deve <espressione su concetti>
Vincoli di derivazione: <concetto> si ottiene <operazione su concetti>

Strategie di progettazione concettuale

Strategia top-down: lo schema concettuale viene prodotto mediante una serie di raffinamenti successivi a partire da uno schema iniziale che descrive tutte le specifiche con pochi concetti molto astratti. Lo schema viene poi via via raffinato mediante opportune trasformazioni che aumentano il dettaglio dei vari concetti presenti.

Strategia bottom-up: le specifiche iniziali sono suddivise in componenti via via sempre più piccole, fino a quando queste componenti descrivono un frammento elementare della realtà di interesse. A questo punto, le varie componenti vengono rappresentate da semplici schemi concettuali che possono consistere anche in singoli concetti. I vari schemi così ottenuti vengono poi fusi fino a giungere,attraverso una completa integrazione di tutte le componenti, allo schema concettuale finale.
Strategia inside-out: un caso particolare della strategia bottom-up. Si individuano inizialmente solo alcuni concetti importanti e poi si procede, a partire da questi, a "macchia d'olio". Si rappresentano cioè prima i concetti in relazione con i concetti iniziali, per poi muoversi verso quelli più lontani attraverso una "navigazione" tra le specifiche.

Strategia mista: La strategia mista cerca di combinare i vantaggi della strategia top-down, con quelli della strategia bottom-up, Il progettista suddivide i requisiti in componenti separate, come nella strategia bottom-up. ma allo stesso tempo definisce uno schema scheletro contenente, a livello astratto, i concetti principali dell'applicazione. Questo schema scheletro fornisce una visione unitaria, sia pure astratta, dell'intero progetto e favorisce le fasi di integrazione degli schemi sviluppati separatamente.

Analisi delle prestazioni

Il costo di un'operazione viene valutato in termini di numero di occorrenze di entità e associazioni che mediamente vanno visitate per rispondere a un'operazione sulla base di dati; questa schematizzazione è molto forte e, pur nelle semplici valutazioni che svilupperemo, sarà talvolta necessario riferirci a un criterio più fine.
L'occupazione di memoria viene valutata in termini dello spazio di memoria (misurato per esempio in numero di byte) necessario per memorizzare i dati descritti dallo schema.

Un importante strumento è la *tavola dei volumi*, nel quale per ogni concetto (entità o relazione) si indica il volume (la stima del numero medio di occorrenze di un concetto). La *tavola delle operazioni* invece definisce l'insieme delle operazioni che devono essere implementate sui dati, la tipologia di queste operazioni (interattive/batch) e la frequenza (es. 100 al giorno).

Data un'operazione O di tipo T, definiamo il suo costo $c(O_T)$ come:
$c(O_T) = f(O_T) \cdot w_T \cdot (\alpha \cdot NC_{write} + NC_{read})$

$f(O_T)$ - frequenza dell'operazione
NC_{read} - numero di accessi in lettura a componenti (entità/relazioni) dello schema
NC_{write} - numero di accessi in scrittura a componenti (entità/relazioni) dello schema
W_T - peso dell'operazione (interattiva/batch)
α - coefficiente moltiplicativo delle operazioni in scrittura
Dato uno schema S, ed un'insieme di operazioni sui dati $O_1, O_2, \ldots, O_n$, con costi $c(O_1), c(O_2), \ldots, c(O_n)$, il costo dello schema è definito come: $c(S) = \sum c(O^i)$.

L'occupazione di memoria dello schema la si può determinare conoscendo la tavola dei volumi, il tipo di ciascun attributo e la sua dimensione su disco.
$M(S) = \sum V(e) * size(e) + \sum V(r) * size(r)$

V(e), size(e) = tabella dei volumi e dimensione in termini di occupazione di memoria dell'entità e.
V(r), size(r) = tabella dei volumi e dimensione in termini di occupazione di memoria della relazione r.
(L'obiettivo è quello di determinare il miglior trade-off tra occupazione di memoria e costo delle operazioni dello schema)

Progettazione Logica

<u>Eliminazione delle generalizzazioni</u>:
> *Accorpamento delle figlie della generalizzazione nel genitore*: le entità E_1 ed E_2 vengono eliminate e le loro proprietà (attributi e partecipazioni ad associazioni e generalizzazioni) vengono aggiunte all'entità genitore E_0. A tale entità viene aggiunto un ulteriore attributo che serve a distinguere il "tipo" di un'occorrenza di E_0 cioè se tale occorrenza apparteneva a E_1, a E_2 o, nel caso di generalizzazione non totale, a nessuna di esse. NOTA: possibile presenza di valori nulli, consigliata quando le operazioni non fanno molta distinzione tra occorrenze e attributi di E_0, E_1 e E_2.

> *Accorpamento del genitore della generalizzazione nelle figlie*: l'entità genitore E_0 viene eliminata e, per la proprietà dell'ereditarietà, i suoi attributi, il suo identificatore e le relazioni a cui tale entità partecipava, vengono aggiunti alle

entità figlie E_1 ed E_2. NOTA: possibile presenza di valori nulli e realizzabile solo se la generalizzazione è totale, conveniente quando ci sono operazioni che coinvolgono solo E_1 ed E_2 ma non l'entità genitore E_0.

> *Sostituzione della generalizzazione con associazioni*: la generalizzazione si trasforma in due associazioni uno a uno che legano rispettivamente l'entità genitore con le entità figlie E_1 ed E_2. Non ci sono trasferimenti di attributi o associazioni e le entità E_1 ed E_2 sono identificate esternamente dall'entità E_0. Nello schema ottenuto vanno aggiunti però dei vincoli: ogni occorrenza di E_0 non può partecipare contemporaneamente a R_{01} e R_{02}; inoltre, se la generalizzazione è totale, ogni occorrenza di E_0 deve partecipare o a un'occorrenza di R_{01} oppure a un'occorrenza di R_{02}. NOTA: conveniente quando la generalizzazione non è totale, non crea valori nulli.

<u>Eliminazione degli attributi multi-valore</u>
Gli attributi multivalore non sono presenti nel modello logico, ma possono essere sostituiti introducendo una relazione uno-a-molti. NOTA: non introduce valori nulli, ma aumenta il numero di entità presenti nel sistema.

<u>Partizionamento/accorpamento di concetti</u>:
Come principio generale gli accessi si riducono separando attributi di uno stesso concetto che vengono acceduti da operazioni diverse (partizionamento) e raggruppando attributi di concetti diversi che vengono acceduti dalle medesime operazioni (accorpamento).
NOTA: Gli accorpamenti si effettuano in genere su associazioni di tipo uno-a-uno, raramente su associazioni uno-a-molti e praticamente mai su relazioni molti-a-molti. Questo perché gli accorpamenti di entità legate da un'associazione uno-a-molti o molti-a-molti generano ridondanze.

<u>Analisi delle ridondanze</u>:
Una ridondanza in uno schema concettuale corrisponde alla presenza di un dato che può essere derivato (cioè ottenuto attraverso una serie di operazioni) da altri dati. Casi frequenti:
- Attributi derivabili, occorrenza per occorrenza, da altri attributi della stessa entità (o associazione)
- Attributi derivabili da attributi di altre entità (o associazioni), di solito attraverso funzioni aggregative
- Attributi derivabili da operazioni di conteggio di occorrenze
- Associazioni derivabili dalla composizione di altre associazioni in presenza di cicli.

Traduzione in modello E-R

> *Entità e associazioni molti-a-molti*: si crea per ogni entità, una relazione con lo stesso nome avente per attributi i medesimi attributi dell'entità e per chiave il suo identificatore; per l'associazione, una relazione con lo stesso nome avente per attributi gli attributi dell'associazione e gli identificatori delle entità coinvolte; tali identificatori formano la chiave della relazione.

> *Associazioni uno-a-molti*: si creano relazioni per le entità coinvolte e nella relazione con cardinalità (1,1) si mette come chiave esterna la chiave della relazione con cardinalità (1,N), su quel valore esisterà ovviamente un vincolo di integrità.

> *Entità con identificatore esterno*: le entità con identificatori esterni danno luogo a relazioni con chiavi che includono gli identificatori delle entità "identificanti".

> *Associazioni uno-a-uno*: per le associazioni uno a uno con partecipazioni obbligatorie [(1,1) - (1,1)] per entrambe le entità è possibile mettere in una delle due relazioni (in modo intercambiabile) una chiave esterna che punta alla chiave primaria dell'altra relazione. Per le associazioni uno a uno con partecipazione opzionale per una sola entità [(0,1) - (1,1)] nella relazione con cardinalità (1,1) si mette come chiave esterna la chiave primaria dell'altra relazione con (0,1).

Normalizzazione

Le ridondanze sui dati possono essere di tipo *concettuale* (non ci sono duplicazioni dello stesso dato, ma sono memorizzate informazioni che possono essere ricavate da altre già contenute nel DB) o *logico* (esistono duplicazioni sui dati, che possono generare anomalie nelle operazioni sui dati).

Dipendenza Funzionale (DF): data una tabella su uno schema R(X) e due attributi Y e Z di X, esiste la dipendenza funzionale $Y \rightarrow Z$ se per ogni coppia di tuple t_1 e t_2 di r (istanza di R) con $t_1[Y] = t_2[Y]$, si ha anche che $t_1[Z] = t_2[Z]$. Nel caso di K come superchiave si avrebbe $t_1[K] = t_2[K] \rightarrow t_1 = t_2$.

Es. se in una tabella si ha Impiegato e Stipendio, in cui ogni impiegato ha un unico stipendio, ogni volta che appare il nome dell'impiegato, si ha il suo stipendio corrispondente, in tal caso vi è una dipendenza funzionale Impiegato →Stipendio (ovviamente ogni campo ha anche una DF con se stesso).

Le Dipendenze Funzionali sono una generalizzazione del vincolo di chiave (e di superchiave). Data una tabella con schema R(X), con superchiave K, esiste un vincolo di dipendenza funzionale tra K e qualsiasi attributo dello schema r:
$K \rightarrow X_1, \qquad X_1 \subseteq X$

Forma Normale di Boyce-Codd (FNBC)
Uno schema R(X) si definisce in forma normale di Boyce e Codd se per ogni dipendenza funzionale (non ovvia) Y→Z definita su di esso, Y è una superchiave di R(X).

Per normalizzare una tabella è possibile creare tabelle separate per ogni dipendenza funzionale, ognuna in FNBC.

Decomposizione senza perdita
Uno schema R(X) si decompone senza perdita negli schemi R1(X1) ed R2(X2) se, per ogni possibile istanza r di R(X), il join naturale delle proiezioni di r su X1 ed X2 produzione la tabella di partenza: $\pi_{X1}(r) \bowtie \pi_{X2}(r) = r$.
In caso di decomposizione con perdite, possono generarsi delle *tuple spurie* dopo il join. NOTA: per conservare le dipendenze in una decomposizione, ogni dipendenza funzionale dello schema dovrebbe coinvolgere attributi che compaiono *tutti insieme* in uno degli schemi decomposti.

Le decomposizioni dovrebbero sempre soddisfare tre proprietà:
- *Soddisfacimento della FNBC*: ogni tabella deve essere in FNBC (non sempre ottenibile).
- *Decomposizione senza perdita*: il join delle tabelle decomposte deve produrre la relazione originaria.
- *Conservazione delle dipendenze*: il join delle tabelle decomposte deve rispettare tutte le DF originarie.

Terza Forma Normale (TFN)
Una definizione di forma normale meno restrittiva di Boyce e Codd è la TFN.
Una tabella r è in terza forma normale se per ogni dipendenza funzionale X→A (non banale) dello schema, almeno una delle seguenti condizioni è verificata:
- X contiene una chiave K di r (X è una superchiave di r)
- A appartiene ad almeno una chiave K di r

Note: i contro sono che la TFN è meno restrittiva della FNBC: tollera alcune ridondanze ed anomalie sui dati e certifica meno la qualità dello schema ottenuto. I pro sono che la TFN è sempre ottenibile, qualsiasi sia lo schema di partenza, attraverso l'algoritmo di normalizzazione in TFN.

Una dipendenza funzionale X →Y è *banale* se Y è contenuto in X.
Es. Impiegato Progetto →Impiegato

Data una relazione r con schema R(X) non i TFN, normalizzare in TFN vuol dire: decomporre r nelle relazioni $r_1, r_2, \dots, r_n$ garantendo che:
- Ogni $r_i (1 \le i \le n)$ è in TFN
- La decomposizione è senza perdite: $r_1 \bowtie r_2 \bowtie r_n = r$

- La decomposizione conserva tutte le DF definite sullo schema R(X) di partenza.

In generale con l'algoritmo di normalizzazione in terza forma normale si vuole *semplificare l'insieme di dipendenze F*, rimuovendo quelle non necessarie, e trasformando ogni dipendenza in modo che nella parte destra compaia un singolo attributo. *Raggruppare gli attributi coinvolti nelle stesse dipendenze*, e costruire le relazioni corrispondenti. Assicurarsi che *almeno una delle relazioni prodotte contenga la chiave* della relazione originaria.

Implicazione Funzionale

Dato un insieme di dipendenze funzionali F, ed una dipendenza funzionale f, diremo che *F implica f* se ogni tabella che soddisfa F soddisfa anche f.
Es. F: {Impiegato → Livello; Livello → Stipendio } - f: Impiegato → Stipendio
Qui F implica f.

Chiusura di una Dipendenza Funzionale

Dato uno schema R(U), con un insieme di dipendenze F. Sia X un insieme di attributi contenuti in U, si definisce la chiusura di X rispetto ad F (X^+_F) come l'insieme degli attributi che dipendono funzionalmente da X:
X^+_F = {A | A ∈ U e F implica X →A }
Esempio: R = (ABCDE) e F = {A → B, A → C, E → B, C → D} la chiusura di A è
A^+_F = {A, B, C, D}

Come è possibile allora verificare se F implica f: X→Y? Si calcola la chiusura X^+_F, se Y appartiene ad X^+_F, allora F implica f.
Es. F = {A → B, BC → D, B → E, E → C} - f: A → E
A^+_F = {A, B, E, C, D} quindi F implica A → E.

Determinare una superchiave tramite chiusura

Per verificare se X è una superchiave di r, è sufficiente determinare la chiusura X^+_F.
Dato uno schema R(U), con un insieme F di dipendenze funzionali, allora un insieme di attributi K è una (super)chiave di R(U) se F implica K → U.
Es. R = (ABCDE) ed F = {A → B, BC → D, B → E, E → C}
Se A è una chiave allora F implica A → ABCDE, A^+_F = {A, B, E, C, D} quindi A è una chiave.

Insiemi di dipendenze equivalenti

Dati due insiemi di dipendenze funzionali F_1 e F_2, essi si dicono equivalenti se F_1 implica ciascuna dipendenza di F_2 e viceversa.
Es. F = {A → B, AB → C} e F' = {A → B, A → C} sono equivalenti

Insiemi di dipendenze non ridondanti

Dato un insieme di dipendenze funzionali F definito su uno schema R(U), esso si dice non ridondante se non esiste una dipendenza f di F tale che F-{f} implichi f.
Es. F = {A → B, AB → C, A → C} è ridondante perché: F-{A → C} implica A → C

Insiemi di dipendenze ridotte

Dato un insieme di dipendenze funzionali F definito su uno schema R(U), esso si dice ridotto se (1) *non è ridondante*, e (2) *non è possibile ottenere un insieme F' equivalente eliminando attributi dai primi membri di una o più dipendenze di F*.
Es. F = {A → B, AB → C} non è ridotto perché B può essere eliminato da AB → C e si ottiene ancora un insieme F' equivalente ad F.

Trovare la copertura ridotta

Dato uno schema R(U) con insieme di dipendenze F, per trovare una copertura ridotta di F si procede in tre passi:
1. Sostituire F con F_1, che ha tutti i secondi membri composti da un singolo attributo
Es. F = {M → RSDG, MS → CD, G → R, D → S, S → D, MPD → AM} diventa
F_1 = {M → R, M → S, M → D, M → G, MS → C, MS → D, G → R, D → S, S → D, MPD → M}

2. Eliminare gli attributi estranei
Es. F = {AB → C, A → B}, calcolando A^+_F = {A, B, C}; C dipende solo da A (in quanto C compare nella chiusura di A), quindi l'attributo B in AB → C può essere eliminato preservando l'uguaglianza: F_1 = {A → C, A → B}

3. Eliminare le ridondanze non necessarie
Per stabilire se la dipendenza del tipo X → A è ridondante, A si elimina da F, si calcola $X^+_{F-\{X\to A\}}$, e si verifica se tale insieme include ancora A. Nel caso lo inclusa, si elimina la dipendenza funzionale X→A.
Es. F = {B → C, B → A, C → A}; B → A è ridondante in quanto bastano le dipendenze B → C e C → A per capire che A dipende da B. Formalmente si dovrebbe dimostrare che F-{B → A} implica {B → A} quindi, verificare che $B^+_{F-\{B\to A\}}$ contiene A.

Algoritmo di Normalizzazione in Terza Forma Normale (TFN)

Dati R(U), ed un insieme di dipendenze F, l'algoritmo di normalizzazione in terza forma normale procede come segue:
1. Costruire una copertura ridotta F_1 di F.
2. Decomporre F_1 nei sottoinsiemi $F_1^{(1)}$, $F_1^{(2)}$, … , $F_1^{(n)}$, tali che ad ogni sottoinsieme appartengano dipendenze con gli stessi lati sinistri.
3. Se due o più lati sinistri delle dipendenze si implicano a vicenda, si fondono i relativi insiemi.

4. Trasformare ciascun $F_1^{(i)}$ in una relazione $R^{(i)}$ con gli attributi contenuti in ciascuna dipendenza. Il lato sinistro diventa la chiave della relazione.
5. Se nessuna relazione $R^{(i)}$ così ottenuta contiene una chiave K di R(U), inserire una nuova relazione $R^{(n+1)}$ contenente gli attributi della chiave.

Esempio
F = {M → RSDG, MS → CD, G → R, D → S, S → D, MPD → AM}
1. F_1 = {M → D, M → G, M → C, G → R, D → S, S → D, MP → A}
2. $F_1^{(1)}$ = {M → D, M → G, M → C}; $F_1^{(2)}$ = {G → R}; $F_1^{(3)}$ = {D → S}; $F_1^{(4)}$ = {S → D}; $F_1^{(5)}$ = {MP → A};
3. $F_1^{(3)}$ = {D → S}; $F_1^{(4)}$ = {S → D} diventano $F_1^{(3)}$ = {D → S, S → D}
4. $F_1^{(1)}$: $R^{(1)}$(<u>M</u>DGC); $F_1^{(2)}$: $R^{(2)}$(<u>G</u>R); $F_1^{(3)}$: $R^{(3)}$(<u>SD</u>); $F_1^{(4)}$: $R^{(4)}$(<u>MP</u>A)
5. La chiave è contenuta da $R^{(4)}$(<u>MP</u>A), non vengono aggiunte relazioni.
In conclusione la decomposizione in 3FN di F è $R^{(1)}$(<u>M</u>DGC), $R^{(2)}$(<u>G</u>R), $R^{(3)}$(<u>SD</u>), $R^{(4)}$(<u>MP</u>A).

<u>Seconda Forma Normale (SFN)</u>
Una tabella con schema R(U) è in Seconda Forma Normale quando non presenta dipendenze parziali, nella forma: Y → A, dove:
- Y è un sottoinsieme proprio della chiave
- A è un qualsiasi sottoinsieme di R(U)

Es. IMPIEGATO(<u>Impiegato</u>, Categoria, Stipendio) con Impiegato → Categoria e Categoria → Stipendio è in SFN ma non in TFN.

Implementazione di basi di dati

HTML è un linguaggio di markup per la creazione di ipertesti multimediali distribuiti, la tecnica di rappresentazione markup avviene attraverso l'utilizzo di tag che definiscono le proprietà grafiche o strutturali del testo.
HTTP è un protocollo applicativo per l'interazione client/server che fa affidamento sul protocollo TCP (porta 80), utilizza una comunicazione asimmetrica (pull-based) e connessioni persistenti, ma è un protocollo stateless.

Un Middle-Tier (MT) è un sistema di collegamento tra il Web-server e il DBMS. Riceve i parametri in ingresso dal Web-server, interroga il DBMS ed estrae le informazioni di interesse (tramite SQL) e produce la pagina HTML con le informazioni richieste.
Un MT consiste in un programma esterno Common Gateway Interface (si usa l'URL della richiesta HTTP per invocare un programma presente sul server, gateway, che viene eseguito e calcola la pagina da restituire al client), un interprete di linguaggi di scripting server-side (che possiede librerie/moduli per la connessione al DBMS) integrato nel web-server e di un application-server multilivello.

Un Web Service (W3C) è un sistema software per il supporto delle interazioni tra macchine in rete, le due componenti principali sono il linguaggio per la definizione dei servizi offerti (WSDL) e il protocollo per lo scambio dei messaggi tra i servizi (SOAP - Simple Object Access Protocol).

Cenni di PHP

Per creare un oggetto: $myObj = new myClass();
Per accedere a un attributo di un oggetto: $myObj->$attr;
Per accedere a una funzione di un oggetto: $myObj->fun();
Per accedere a metodi/attributi di un oggetto all'interno della definizione della classe si utilizza: $this->fun() o $this->var.
Il costruttore di una classe si definisce tramite: __construct()
Una classe può avere metodi e attributi statici a cui si accede tramite: myClass:: $var o myClass::func();
Si può definire la visibilità degli attributi/metodi tramite: public, private e protected
L'ereditarietà: class B extends A { … }
Possono essere definite anche classi astratte (abstract), interfacce (interface) e type hinting.

Connessione al DB

```
try {
        $pdo = new PDO('mysql:host=localhost;
dbname=mydb',"marco","mypasswd');
} catch (PDOException ex) {
        echo("Connessione non riuscita");
        exit();
}
```

Impostazione di attributi della connessione

```
$pdo -> setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$pdo -> exec('SET NAMES "utf8"');
```

Esecuzione di query

```
try {
        $sql='INSERT into STUDENTI VALUES("Marco", "Rossi")';
        $result=$pdo -> exec($sql);
} catch(PDOException $e) {
        echo('Codice errore'.e -> getMesssage());
        exit();
}
$sql='SELECT * FROM STUDENTI';
$result=$pdo -> query($sql);
```

Recupero dei dati restituiti

```
while($row=$result -> fetch()) {
        echo('Nome:'.$row['name']);
}

foreach($result as $row) {
        echo('Nome:'.$row['name']);
}
```

Costruire il template della query (SQLInjections) e predisporne l'esecuzione

```
$sql='SELECT COUNT(*) AS counter FROM Login WHERE (Utente=:lab1) AND
(Password=:lab2)';

$res=$pdo->prepare($sql);

$res -> bindValue(":lab1", $username);
$res -> bindValue(":lab2", $password);

$res -> execute();
```

Transazioni

PDO::beginTransaction(void) (Inizia una transazione, settando autocommit a false)
PDO::commit() (Effettua il commit della transazione)
PDO::rollback() (Richiede l'UNDO della transazione corrente)

Cookies

setcookie(name, value, expiriyTime, path, domain, secure, httpOnly)

In breve

1FN: Le relazioni non devono contenere attributi multivalore o relazioni annidate.
Normalizzazione: creare nuove relazioni per ciascun attributo multivalore o per ciascuna relazione annidata.
2FN: Deve essere in 1FN e nel caso di relazioni in cui la chiave primaria contiene più attributi, nessun attributo non chiave deve essere funzionalmente dipendente da una sola parte della chiave (possono esserci però DF tra attributi non chiave).
Normalizzazione: decomporre creando una nuova relazione per ciascuna chiave parziale con i relativi attributi dipendenti. Deve rimanere anche una relazione che contiene la chiave primaria originaria e tutti gli attributi che sono dipendenti in modo completo dalla chiave.
TIP: è in SFN (neanche in TFN) se si ha tipo A -> B, B -> C con A chiave; NON è in SFN (neanche in TFn) A -> B, C -> D con AC chiave.
3FN: Deve essere in 2FN e la relazione non deve contenere attributi non chiave che siano funzionalmente dipendenti da altri attributi non chiave.
Normalizzazione: decomporre creando una relazione che contenga gli attributi non chiave che determinano funzionalmente altri attributi non chiave.
TIP: dato A -> B se ne A ne B sono chiave, non è in TFN, se però B ha tutti attributi che appartengono ad almeno una chiave è in TFN, oppure se A è una superchiave è in TFN.
FNBC: Deve essere in 1FN e tutti gli attributi dipendono da superchiavi.
Normalizzazione: decomporre creando tabelle separate per ogni dipendenza funzionale, ognuna in FNBC.
TIP: dato A -> B è in FNBC se A è superchiave.

```
CREATE DATABASE DatabaseName;
CREATE TABLE MyTable (
  Id INT NOT NULL AUTO_INCREMENT,
  Code VARCHAR(10),
  Creation DATE,
  City VARCHAR(50) CHECK (City LIKE 'CITY_%'),
  PRIMARY KEY (Id, Code),
  FOREIGN KEY (Creation) REFERENCES Dates(MainDate) ON UPDATE
CASCADE
  FOREIGN KEY (City) REFERENCES Cities(Name) ON DELETE CASCADE
);
```

```
CREATE DOMAIN MyDomain AS SMALLINT DEFAULT 40 CHECK(MyDomain
>= 40);

SELECT Attr1, Attr2, Attr3
FROM TableA, TableB
WHERE Attr4 LIKE “Word”
AND Attr6 BETWEEN #07/04/1996# AND #07/09/1996#
LIMIT 2
GROUPBY Attr1, Attr2
HAVING COUNT(Attr2) > 3
ORDERBY Attr3;

INSERT INTO Table(Attr1, Attr2, Attr3) VALUES ('A', 10, '2012-06-18');

DELETE FROM Table WHERE Attr1 = “Word”;

UPDATE Customers SET Attr1 = 'X', Attr2 = 2 WHERE Attr3 = 'A';

SELECT1 [...] UNION/INTERSECT/EXCEPT SELECT2 [...];

SELECT Attr2 FROM TableA AS A INNER JOIN TableB AS B ON A.Attr1 = B.Attr1;
SELECT Attr2 FROM TableA AS A FULL OUTER JOIN TableB AS B ON A.Attr1 =
B.Attr1;
SELECT Attr2 FROM TableA AS A LEFT JOIN TableB AS B ON A.Attr1 = B.Attr1;
SELECT Attr2 FROM TableA AS A RIGHT JOIN TableB AS B ON A.Attr1 = B.Attr1;
CREATE ASSERTION AssertionName CHECK ((SELECT COUNT FROM Table
WHERE Attr1 = 1) = 0);

CREATE VIEW ViewName AS SELECT [...];

DELIMITER %
CREATE PROCEDURE ProcedureName (IN InParam VARCHAR(50), OUT
OutParam BOOLEAN)
BEGIN
  SET @donations = (SELECT COUNT(*) FROM Table);
  SELECT @donations;
  IF @donations >= 1 THEN
        SET OutParam = 1;
  ELSE
        SET OutParam = 0;
  END IF;
END; %
DELIMITER ;
```

```
CREATE TRIGGER TriggerName AFTER/BEFORE INSERT/UPDATE/DELETE
ON TableName
FOR EACH ROW Query(Update/Insert/delete);

CREATE USER 'user'@'localhost' IDENTIFIED BY 'password';

GRANT/REVOKE INSERT, UPDATE, SELECT ON TableName TO
'user'@'localhost';

CREATE ROLE DatabaseName AUTHORIZATION 'user'@'localhost';

START TRANSACTION
UPDATE SalariImpiegati SET conto = conto * 1.2
IF conto > 0
   COMMIT;
ELSE ROLLBACK;
```

MongoDB

```
db.DBNAME.find(
   {$or:[ {"field1": "Mario"}, {"field2": {$lt: 1000}} ]},
   {“_id”:0, “field3”:1}
);
```