Indice

1 Introduzione ai DBMS 1
1.1 Architettura a livelli del DBMS 1
2 Modello relazionale 1
2.1 Struttura 2
2.2 Vincoli sui dati della relazione 2
2.3 Prima forma normale (PFN) 2
2.4 Schema matematico del modello relazionale 2
2.5 Informazioni incomplete 2
2.6 Vincoli di integrità 2
2.7 Problemi 3
3 Linguaggio SQL 4
3.1 SQL-DDL 4
3.2 SQL-DML 6
3.3 Interrogazioni Annidate 9
3.4 Viste 10
3.5 Common Table Expression (CTE) 10
3.6 Assertion 10
3.7 Costrutti Avanzati 10
4 MySQL 12
4.1 Autenticazione 12
4.2 Creazione database 12
4.3 Tipi di dato 13
4.4 Comandi CRUD 13
4.5 Creazione di Trigger 14
4.6 Creazione di Viste 14
4.7 Creazione di Stored Procedures 14
4.8 Gestione delle transazioni per tabelle INNODB. 15
4.9 Utilities 15
5 Gestione delle Transazioni 15
5.1 Lock Manager 17
5.2 Gestione delle transazioni 17
5.3 Deadlock 17
5.4 Timestamp (TS) 18
6 NoSQL 18
6.1 Motivi della diffusione dei database NoSQL 18
6.2 Proprietà Base dei Sistemi NoSQL 19
7 MongoDB 19
7.1 Correlazioni tra collezioni 22
7.2 Aggregazione di dati 22
8 Progettazione di basi di dati 23
8.1 Analisi dei requisiti 23
8.2 Raccolta dei requisiti 24
9 Diagramma Entità-Relazionale 24

1 Introduzione ai DBMS

Modello relazionale: organizza i dati in record di dimensione fissa mediante tabelle.
Sistema informativo (SI): componente di un'organizzazione il cui scopo è gestire le informazioni utili ai fini dell'organizzazione stessa.
DBMS: sistema software in grado di gestire collezioni di dati grandi, condivise e persistenti in maniera efficiente e sicura.
Base di dati: collezione di dati gestita da un DBMS.

1.1 Architettura a livelli del DBMS

Ogni livello è indipendente dall'altro

- **Schema esterno**: come si presenta il DB, e come varia in base ai permessi di accesso.
- **Schema logico**: come sono strutturati i dati e che relazioni hanno. In genere il modello logico utilizzato è quello relazionale, dove i dati sono organizzati in tabelle. Si usano delle regole per modellare eventuali vincoli e restrizioni sui dati.
- **Schema fisico**: come/dove sono memorizzati i dati.

SQL è un linguaggio orientato ai dati, usato per il modello relazionale.
Un DBMS si usa quando:

- i dati sono condivisi da più utenti;
- i dati sono persistenti;
- i dati sono voluminosi e complessi;
- servono meccanismi di sicurezza e controllo degli accessi;

2 Modello relazionale

Siamo nello schema logico: definiamo le tabelle, le relazioni tra di esse e i vincoli sui dati.
Il modello relazionale è il più utilizzato, garantisce indipendenza tra i livelli e si basa su nozioni di algebra relazionale.

2.1 Struttura

I dati sono organizzati in record di dimensione fissa e divisi in tabelle (relazioni).

- Colonne: rappresentano gli attributi, hanno un nome e un tipo di dato.
- Intestazione della tabella (nome tabella + nome attributi): **schema** della relazione.
- Righe della tabella: istanze della relazione (ennuple)[1].

L'ordine delle righe e delle colonne non ha importanza, ma l'ordine degli attributi sì.

2.2 Vincoli sui dati della relazione

- non possono esistere attributi con lo stesso nome;
- non possono esistere righe uguali;
- i dati di una colonna devono essere omogenei (omogeneità di tipo);

Si possono avere schemi senza istanze, ma non istanze senza schema.

2.3 Prima forma normale (PFN)

Una relazione è in PMF se tutti gli attributi sono atomici, cioè non possono essere ulteriormente divisi.
Non è possibile omettere un valore da una ennupla: devono avere tutti i campi obbligatoriamente.

Una base di dati può essere costituita da molte tabelle. Spesso, le informazioni contenute in relazioni diverse sono correlate logicamente tra loro. Nel modello relazionale, i riferimenti tra dati in relazioni differenti sono espressi mediante valori.

Corsi		
Nome corso	**Codice Corso**	**Nome Docente**
Basi di dati	BD001	Mario Rossi
Sistemi informativi	SI002	Giovanni Verdi

Esami		
Corso	**Studente**	**Voto**
BD001	Luca Bianchi	30
BD001	Anna Neri	28

Table 1: Esami contiene un codice che è lo stesso di corsi

Nella progettazione, bisogna tradurre le informazioni in dati del modello relazionale. Ci si chiede quali dati devono essere gestiti e quante tabelle servono.

[1] Un'ennupla (o tupla) corrisponde ad un istanza della relazione.

2.4 Schema matematico del modello relazionale

DEF: Dati n insiemi $D_1, D_2, ..., D_n$, una relazione matematica su questi insiemi è un sottoinsieme del prodotto cartesiano $D_1 \times D_2 \times ... \times D_n$.

DEF: Il prodotto cartesiano degli insiemi $D_1, D_2, ..., D_n$ è l'insieme di tutte le ennuple ordinate $(d_1, d_2, ..., d_n)$ con $d_i \in D_i, \forall i = 1, ..., n$.

EX: Relazione

$A = \{a, b, c, d, e\}, \ B = \{1, 2, 3\}$

$A \times B = \{(a,1), (a,2), (a,3), (b,1), (b,2),$

$(b,3), (c,1), (c,2), (c,3), (d,1), (d,2), (d,3), (e,1), (e,2), (e,3)\}$

$R_1 \subseteq A \times B = \{(a,1), (a,2), (a,3)\}$

$R_2 \subseteq A \times B = \{(a,2), (b,1), (d,3), (e,3)\}$

A e B sono due tabelle con un solo campo. $a, b, c, ..., 1, 2, 3$ sono le istanze. Il prodotto cartesiano unisce tutte le istanze facendo tutte le combinazioni possibili. Le istanze di $A \times B$ sono tutte le combinazioni di A e B. Le relazioni R_1 e R_2 sono sottoinsiemi del prodotto cartesiano, quindi sono relazioni.

Una ennupla su un 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 dell'ennupla T nell'attributo A.

2.5 Informazioni incomplete

In una relazione le ennuple devono essere omogenee, ossia avere tutte la stessa struttura. Se il valore di un attributo non è noto, si usa il valore NULL. $T[A] \in A \vee$ NULL $\forall$ attributo A.
Per definizione, il valore NULL non è uguale a nessun altro valore, nemmeno a se stesso[2].

2.6 Vincoli di integrità

I vincoli di integrità sono regole che limitano i valori che possono essere inseriti in una relazione.
Un vincolo è una funzione booleana che associa ad una istanza r di una base di dati definita su uno schema $R = \left\{R_1(x_1), ..., R_{k(x_k)}\right\}$ un valore booleano. Un'istanza è lecita se soddisfa tutti i vincoli.

[2] NULL $\neq$ NULL.

2.6.1 Vincoli intra-relazionali

I vincoli intra-relazionali sono regole che limitano i valori che possono essere inseriti in una singola relazione.

Vincoli di ennupla: questi vincoli esprimono condizioni su una ennupla, considerata singolarmente. Possono essere espressi tramite espressioni algebriche o booleane.

EX: Vincolo di ennupla

`((voto>=18) and (voto<=30))`, `not((lode=true) and (voto!=30))`, `(saldo = entrate-uscite)`

Vincoli di chiave: una chiave è un insieme di attributi che consente di identificare in maniera univoca le ennuple di una relazione.

EX: Chiave

La matricola di uno studente: studenti(matricola, cognome, nome, data). Non esistono due studenti con la stessa matricola: data la matricola di uno studente è possibile risalire a tutti i suoi dati.

DEF: **Superchiave**
Un sottoinsieme di k attributi di una relazione è una superchiave se non contiene due ennuple distinte T_1 e T_2 con $T_1[k] = T_2[k]$. Nell'esempio di prima, matricola è una superchiave

DEF: **Superchiave Minimale**
La superchiave k è minimale $\iff \nexists k' \mid k \subseteq k'$. k è la superchiave più piccola, non ne esiste un'altra che la contenga.

In una relazione esiste sempre almeno una superchiave alla peggio si prendono tutti i campi[3].

Le chiavi servono per accedere a ciascuna ennupla della base di dati in maniera univoca e correlare dati tra relazioni (tabelle) differenti.

[3]Per definizione non possono esserci due ennuple uguali.

DEF: **Chiave Primaria**
Chiave di una relazione su cui non sono attesi valori NULL. Gli attributi che formano la chiave primaria sono per convenzione sottolineati.

EX: Chiave primaria

studenti(<u>matricola</u>, nome, cognome)

EX: Chiave primaria con più attributi

partita(<u>squadra1</u>,<u>squadra2</u>,<u>data</u>,punti1, punti2)

Ogni relazione deve disporre di una chiave primaria. Se tutti i campi presentano dei valori NULL, si aggiunge un codice univoco o un identificativo progressivo.

2.6.2 Vincoli inter-relazionali

Una base di dati può essere composta da molte relazioni collegate tra loro. I collegamenti tra relazioni differenti sono espressi tramite valori comuni in attributi replicati.

Ogni riga della tabella referenziante si collega al massimo ad una riga della tabella referenziata, in base ai valori comuni nell'attributo replicato.

tabella referenziata (chiave primaria)
tabella referenziante (chiave secondaria)

Un vincolo di integrità referenziale (chiave esterna) fra gli attributi x di R_1 e un'altra relazione R_2 impone ai valori su x in R_1 di comparire come valori della chiave primaria di R_2.

Il vincolo garantisce che non ci siano riferimenti a elementi inesistenti: ogni valore usato come chiave esterna in una tabella deve esistere come chiave primaria nell'altra tabella a cui si riferisce.

2.7 Problemi

Se un operazione di aggiornamento o modifica causa violazioni dei vincoli di integrità su altre relazioni

- non si consente l'operazione;
- si elimina a cascata

- si inseriscono valori NULL

3 Linguaggio SQL

DEF: **Relational DataBase Management System (RDBMS)**
Un RDBMS è un software che consente di creare, gestire, modificare e interrogare basi di dati strutturate in forma relazionale e usa SQL per operare sui dati.

I linguaggi data-oriented permettono di implementare il modello relazionale in un RDBMS. Essi dispongono di UI, linguaggi basati su proprietà algebrico-logiche. Il più famoso è SQL.

Si applicano i concetti del modello relazionale, ma con delle differenze:
- si parla di tabelle e non di relazioni;
- il sistema dei vincoli è più espressivo;
- ci possono essere tabelle con righe duplicate se non c'è la chiave primaria;
- il vincolo di integrità referenziale è meno stringente.

3.1 SQL-DDL

Contiene i costrutti necessari per la creazione e modifica dello schema della base di dati.

`create database[if not exists] <nome DB>`	crea DB
`drop database[if exists] <nome DB>`	cancella DB

`create table NOMETABELLA(`
`  NOMEATTRIBUTO1 DOMINIO[<val default>][<vincoli>]`
`  ...`
`)`
	crea tabella

Domini	
`character[<lunghezza max>]` `[<lunghezza>]` alternativa `varchar(<lunghezza>)`	Se la lunghezza non è specificata accetta un singolo carattere
- `numeric[(Precisione [,Scala])])` - `decimal[(Precisione [,Scala])])` • `integer` • `smallint`	I tipi numerici esatti consentono di rappresentare valori esatti, interi o con una parte decimale di lunghezza prefissata.
• `integer auto_increment` • `smallint auto_increment`	La keyword `auto_increment` consente di creare campi numerici che si auto-incrementano ad ogni nuovo inserimento nella tabella.
• `float [<precision>]` • `real` • `double [<precision>]`	I tipi numerici approssimati consentono di rappresentare valori reali con rappresentazione in virgola mobile.
• `date [(Precisione)]` • `time [(Precisione)]` • `timestamp`	I domini temporali consentono di rappresentare informazioni temporali o intervalli di tempo.
`boolean`	I domini booleani consentono di rappresentare valori booleani
• `blob` • `cblob`	I domini blob e cblob consentono di rappresentare oggetti di grandi dimensioni come sequenza di valori binari (`blob`) o di caratteri (`cblob`).

`create domain NomeDominio as TipoDati`
`  [Valore di default]`
`  [Vincolo]`
	Tramite il costrutto `domain` , l'utente può costruire un proprio dominio di dati a partire dai domini elementari.
	`CREATE DOMAIN Voto AS SMALLINT DEFAULT NULL CHECK ( value >=18 AND value <= 30 )`

3.1.1 Vincoli

Per ciascun dominio o attributo è possibile definire dei vincoli che devono essere rispettati da tutte le istanze di quel dominio o attributo.
- Vincoli **intra-relazionali**:
  - vincoli generici di ennupla
  - vincolo `not null`
  - vincolo `unique`
  - vincolo `primary key`
- Vincoli **inter-referenziali**:
  - vincolo `references`

Vincoli

`check(<condizione>)`	Il vincolo viene valutato ennupla per ennupla. `VOTO SMALLINT CHECK((VOTO>=18) and (VOTO<=30))`
`not null`	Il vincolo `not null` indica che il valore NULL non è ammesso come valore dell'attributo. `NUMEROORE SMALLINT NOT NULL`
• `Attributo Dominio [<default value>] unique`: la superchiave è un solo attributo • `unique(Attributo1, Attributo2, ...)`: la superchiave è composta da più attributi	Il vincolo `unique` impone che l'attributo/attributi su cui sia applica non presenti valori comuni in righe differenti, ossia che l'attributo/i sia una superchiave della tabella. Con `unique` sono ammessi valori NULL dato che sono considerati diversi tra loro.
• `Attributo Dominio [ValDefault] primary key`: la chiave è un solo attributo. • `primary key(Attributo1, Attributo2, ...)`: la chiave è composta da più attributi.	Il vincolo `primary key` impone che l'attributo/attributi su cui sia applica non presenti valori comuni in righe differenti e non assuma valori NULL: ossia che l'attributo/i sia una chiave primaria. `CREATE TABLE IMPIEGATI ( ARTICOLO INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY, ... )`

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/i cui si fa riferimento nella tabella esterna deve/devono essere soggetto/i al vincolo unique.

```
CREATE TABLE ESAMI (
CORSO VARCHAR(4) REFERENCES CORSI(CODICE)
STUDENTE VARCHAR(20),
PRIMARY KEY(CORSO, MATRICOLA),
...
)
```

Corsi		
Nome corso	**Codice Corso**	**Nome Docente**
Basi di dati	BD001	Mario Rossi
Sistemi informativi	SI002	Giovanni Verdi

Esami		
Corso	**Studente**	**Voto**
BD001	Luca Bianchi	30
BD001	Anna Neri	28

Table 2: Vincoli di integrità referenziale

Il costrutto `foreign key` si utilizza nel caso il vincolo di integrità referenziale riguardi più di un attributo delle tabelle interne/esterne.

```
CREATE TABLE STUDENTE (
  MATRICOLA CHARACTER(20) PRIMARY KEY,
  NOME VARCHAR(20),
  COGNOME VARCHAR(20),
  DATANASCITA DATE,
  FOREIGN KEY(NOME,COGNOME,DATANASCITA) REFERENCES
  ANAGRAFICA(NOME,COGNOME,DATA)
);
```

Se un valore nella tabella esterna viene cancellato o viene modificato il vincolo di integrità referenziale nella tabella interna potrebbe non essere più valido.

Si possono associare azioni specifiche da eseguire sulla tabella interna in caso di violazioni del vincolo di integrità referenziale.
`on (delete | update) (cascade | set null | set default| no action)`
- `cascade`: elimina/aggiorna righe (della tabella interna)
- `set default`: ripristina il valore di default
- `no action`: non consente l'azione (sulla tabella esterna)
- `set null`: setta i valori a NULL

È possibile modificare gli schemi di dati precedentemente creati tramite le primitive di `alter` (modifica) e `drop` (cancellazione).

- `drop (schema|domain|table|view) NomeElemento`

- `drop (restrict|cascade) NomeElemento`

```
alter NomeTabella
  alter column NomeAttributo
  add column NomeAttributo
  drop column NomeAttributo
  add constraint DefVincolo
```

3.2 SQL-DML

Contiene i costrutti per le interrogazioni, inserimento, eliminazione e modifica dei dati.

3.2.1 Query

```
select attributo1, ... , attributoM
from tabella1, ... , tabellaN
where condizione
```

Fa il prodotto cartesiano tra tabella1,..., tabellaN. Da queste, estrai le righe che rispettano la condizione. Di quest'ultime, preleva solo le colonne corrispondenti ad attributo1, ... ,AttributoM.

Il risultato è una tabella con le righe e colonne richieste dalla query.

Nella clausola `where`, è possibile fare confronti tra stringhe usando l'operatore `like` e l'utilizzo di wildcard:

- `_`: singolo carattere arbitrario
- `%`: sequenza di caratteri arbitraria

EX: Wildcard

```
SELECT CODICE
FROM IMPIEGATI
WHERE (NOME LIKE "M_R%O")
```

`between` consente di verificare l'appartenenza ad un certo insieme di valori.

EX: `between`

Trovare i codici degli impiegati il cui stipendio sia compreso tra i 24000 ed i 34000 euro annui:

```
SELECT NOME
FROM IMPIEGATI
WHERE (STIPENDIO BETWEEN 24000 AND 34000)
```

È possibile ridenominare le colonne del risultato di una query attraverso il costrutto `as`.

EX: `as`

```
SELECT NOME as Name, Cognome as LastName
FROM IMPIEGATI
WHERE (NOME="Marco")
```

EX: `select` su più tabelle

```
SELECT TELEFONO AS TEL
FROM IMPIEGATI, SEDI
WHERE (UFFICIO=UFFNUM) AND (CODICE=145)
```

1. Si effettua il prodotto cartesiano delle due tabelle
2. Si selezionano le righe dove `UFFICIO=UFFNUM` nelle tue tabelle
3. Si selezionano le righe r_{145} relative all'impiegato con `codice=145`
4. Si seleziona la colonna dell'attributo Telefono (c_{tel})
5. Si costruisce il risultato finale ($r_{145} \cap c_{\text{tel}} = 2035434$)

Se le tabelle della clausola `from` hanno attributi con nomi uguali si può utilizzare la notazione `NomeTabella.NomeAttributo` per far riferimento ad un attributo in maniera non ambigua.

```
SELECT TELEFONO AS TEL
FROM IMPIEGATI, SEDI
WHERE (IMPIEGATI.UFFICIO=SEDI.UFFICIO) AND (CODICE=145)
```

Il costrutto `distinct` (nella `select`) consente di rimuovere i duplicati nel risultato. Il comportamento di default `all` non lo fa.

EX: `select distinct`

```
SELECT DISTINCT NOME AS NAME
FROM IMPIEGATI AS I
WHERE (STIPENDIO >20000)
```

Il costrutto `order by` consente di ordinare le righe del risultato di un'interrogazione in base al valore di un attributo specificato.
`order by Attributo1 [asc|desc], ... , AttributoN [asc|desc]`

EX: `order by`

```
SELECT *
FROM IMPIEGATI
WHERE (UFFICIO=“A”)
ORDER BY STIPENDIO
```

3.2.2 Operatori Aggregati

Gli operatori aggregati producono come risultato un solo valore. Solitamente sono inseriti nella `select` e valutati dopo il `where` e `from`.

Operatori Aggregati	
`count (* | [distinct|all] Lista Attributi)`	`*` si applica su tutti gli attributi, in pratica conta il numero di righe
• `sum(Lista Attributi)` • `avg(Lista Attributi)` • `min(Lista Attributi)` • `max(Lista Attributi)`	

EX: `count(*)`

```
SELECT COUNT(*) AS CONTATORE
FROM STRUTTURATI
WHERE (DIPARTIMENTO="FISICA")
```

1. Si considerano le tabelle indicate nella clausola `FROM`
2. Si effettua il prodotto cartesiano delle tabelle.
3. Si selezionano le righe che soddisfano la condizione del `WHERE`.
4. Si considera l'Attributo della `SELECT` e si applica l'operatore aggregato su tutti i valori della colonna.
5. Dalla colonna si calcola un solo valore come risultato della query

L'operatore di raggruppamento 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.

`groupby` combina
- `SELECT ATTRIBUTI FROM WHERE` che valuta i valori di ciascuna riga in isolamento.
- `SELECT OP(ATTRIBUTI) FROM WHERE` che valuta i valori delle righe corrispondenti alle colonne della `SELECT` in modo aggregato.

EX: `groupby`

```
SELECT DIPARTIMENTO AS DIP, COUNT(*) AS NUMERO
FROM STRUTTURATI
GROUPBY DIPARTIMENTO
```

1. Partizionamento della tabella (in questo caso si raggruppano quelle con lo stesso dipartimento)
2. Si applica la `select` su ciascun gruppo
3. Si costruisce il risultato finale (si riuniscono le tabelle restituite dalla `select`)

È possibile filtrare i gruppi in base a determinate condizioni, attraverso il costrutto `having`.

EX: `having`

```
SELECT ListaAttributi1
FROM ListaTabelle
WHERE Condizione
GROUPBY ListaAttributi2
HAVING Condizione
```

1. Prodotto cartesiano delle tabelle
2. Estrazione delle righe che rispettano la condizione della clausola `WHERE`
3. Partizionamento della tabella (`groupby ...`)
4. Selezione dei gruppi (`having ...`)
5. Selezione dei valori delle colonne o esecuzione degli operatori aggregati su ciascuno dei gruppi, e composizione della tabella finale. (`select ...`)

Nel `where` non si possono mettere gli operatori aggregati, nell' `having` si. `where` valuta riga per riga, `having` valuta su ciascun gruppo a livello aggregato.

3.2.3 Operazioni Insiemistiche

È possibile effettuare operazioni insiemistiche o risultati di `select` se gli attributi hanno tipi compatibili.

Operazioni Insiemistiche	
`union`	Estrarre nome e cognome di tutto il personale universitario (strutturati + tecnici)

	`SELECT NOME, COGNOME`
	`FROM STRUTTURATI`
	`UNION`
	`SELECT NOME, COGNOME`
	`FROM TECNICI`

`intersect`	Estrarre nome e cognome degli strutturati che hanno degli omonimi che lavorano come tecnici.
	`SELECT NOME, COGNOME`
	`FROM STRUTTURATI`
	`INTERSECT`
	`SELECT NOME, COGNOME`
	`FROM TECNICI`

`except`	Estrarre nome e cognome degli strutturati che non hanno degli omonimi che lavorano come tecnici
	`SELECT NOME, COGNOME`
	`FROM STRUTTURATI`
	`EXCEPT`
	`SELECT NOME, COGNOME`
	`FROM TECNICI`

3.2.4 Modifica dell'istanza

Modifica dell'Istanza	

`insert`	Inserire una riga esplicitando i valori degli attributi oppure estraendo le righe da altre tabelle del database.
	`INSERT INTO IMPIEGATI(Codice, Nome, Cognome, Ufficio)`
	`values ("8","Vittorio","Rossi","A")`
	I valori non specificati di default sono NULL.

`delete`	Cancellare tutte le righe che soddisfano una condizione(cancella tutto se non specificata).
	`DELETE FROM IMPIEGATI`
	`DELETE FROM IMPIEGATI WHERE (UFFICIO="A")`
	`DELETE FROM TABELLA WHERE NOME IN (`
	`SELECT NOME FROM IMPIEGATICOMUNE)`

`update`	Aggiornare il contenuto di uno o più attributi di una tabella che rispettano una certa condizione.

	`update NomeTabella`
	`set attributo = expr|SELECT|null|default`
	`[where Condizione]`
	`UPDATE IMPIEGATI`
	`SET NOME="Mario"`
	`WHERE (CODICE=5)`
	`UPDATE IMPIEGATI SET NOME=(SELECT NOME FROM IMPIEGATICOMUNE`
	`WHERE CODICE=5) WHERE (CODICE=5)`

3.2.5 Join

È possibile implementare il `join` tra tabelle in due modi distinti (ma equivalenti nel risultato):

- Inserendo le condizioni del `JOIN` direttamente nella clausola del `WHERE`;
- Attraverso l'utilizzo dell'operatore di `inner JOIN` nella clausola `FROM`.

EX: `inner join`

```
SELECT Modello
FROM GUIDATORI JOIN VEICOLI
ON GUIDATORI.NrPatente
=VEICOLI.NrPatente
WHERE (Nome="Sara")
```

Varianti del `join`	

`left join`	Risultato dell'`inner join` + 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`	Risultato dell'`inner join` + 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`	Risultato dell'inner join + righe della tabella di sinistra/destra che non hanno un corrispettivo a destra/sinistra (completate con valori NULL).

	`SELECT ListaAttributi`
	`FROM Tabella FULL JOIN Tabella ON CondizioneJoin`
	`[WHERE Condizione]`
	`...`

3.3 Interrogazioni Annidate

Nella clausola `where`[4], oltre ad espressioni semplici, possono comparire espressioni complesse in cui il valore di un attributo viene confrontato con il risultato di un'altra query.

EX: Query annidate

Estrarre il codice dello strutturato che riceve lo stipendio più alto.

```
SELECT CODICE
FROM STRUTTURATI
WHERE (STIPENDIO = SELECT MAX(STIPENDIO) # query interna
FROM STRUTTURATI)
```

EX: Query interna restituisce più di un valore

Nome e cognome dei dipendenti del dipartimento di INFORMATICA, il cui stipendio è uguale a quello di un dipendente del dipartimento di FISICA

```
SELECT NOME, COGNOME
FROM STRUTTURATI
WHERE (DIPARTIMENTO="INFORMATICA") AND
  (STIPENDIO = (SELECT STIPENDIO # query interna
  FROM STRUTTURATI              # può restituire più di una riga
  WHERE (DIPARTIMENTO="FISICA")))
```

Operatori speciali di confronto per query annidate:

- `any`: la riga soddisfa la condizione se è vero il confronto tra il valore dell'attributo ed almeno uno dei valori ritornati dalla query annidata.
- `all`: la riga soddisfa la condizione se eè 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 interrogazione nidificata, `false` altrimenti.

EX: `in`

```
SELECT ListaAttributi
FROM TabellaEsterna
WHERE Valore/i IN SELECT ListaAttributi2
    FROM TabellaInterna
    WHERE Condizione
```

Il costrutto `exists` restituisce true se l'interrogazione nidificata restituisce un risultato non vuoto (≥ 1 elemento trovato).

EX: `exists`

```
SELECT ListaAttributi
FROM TabellaEsterna
WHERE EXISTS SELECT ListaAttributi2 # controlla se il numero di
    FROM TabellaInterna             # righe della query interna >0
    WHERE Condizione
```

3.3.1 Interrogazioni annidate semplici

Non c'è passaggio di binding[5] tra un contesto all'altro. Le interrogazioni vengono valutate dalla più interna alla più esterna..

EX: Query annidate semplici

Estrarre chi, tra gli informatici, guadagna più di qualunque fisico.

```
SELECT NOME, COGNOME
FROM STRUTTURATI
WHERE (DIPARTIMENTO="INFORMATICA") AND
    (STIPENDIO > ALL (SELECT STIPENDIO #
    FROM STRUTTURATI                   # query interna
    WHERE (DIPARTIMENTO="FISICA")))    #
```

1. Viene valutata la query più interna;
2. Viene confrontata ciascuna riga della tabella più esterna con il risultato della query interna

3.3.2 Interrogazioni annidate complesse

C'è passaggio di binding attraverso variabili condivise tra le varie interrogazioni. In questo caso, le interrogazioni più interne vengono valutate su ogni tupla.

[4] Non è possibile annidare due query nelle altre clausole.

[5] Momento in cui il DBMS "capisce" a quale colonna o tabella ti riferisci quando scrivi, ad esempio, STIPENDIO o DIPARTIMENTO.

EX: Interrogazioni annidate complesse

Estrarre nome/cognome degli impiegati che hanno omonimi.

```
SELECT NOME, COGNOME
FROM IMPIEGATI AS I
WHERE (I.NOME,I.COGNOME) = ANY  (SELECT NOME, COGNOME
    FROM IMPIEGATI AS I2
    WHERE (I.NOME=I2.NOME)
    AND (I.COGNOME=I2.COGNOME)    # stesso nome
    AND (I.CODICE <> I2.CODICE))  # codice diverso
```

La query più interna viene valutata su ciascuna tupla della query più esterna

3.4 Viste

Le viste rappresentano "tabelle virtuali" ottenute da dati contenuti in altre tabelle del DB. Ogni vista ha associato un nome ed una lista attributi, e si ottiene da una `select`.

```
create view nomeview [lista attributi]
as SELECTSQL [with [local | cascade] check option]
```

Le viste esistono a livello di schema, ma non hanno istanze proprie. Sono utilizzate per:

- Far visualizzare solamente certi dati ad alcuni utenti

EX: `view`

```
CREATE VIEW STUDENTI(CODICE,NOME,COGNOME,
DATANASCITA) AS
SELECT CODICE,NOME,COGNOME,NASCITA
FROM PROFESSORI
```

- tabella di appoggio per semplificare una query
- garantire la retro-compatibilità con versioni precedenti dello schema del DB

L'opzione `WITH CHECK OPTION` consente di definire viste aggiornabili, a condizione che le tuple aggiornate continuino ad appartenere alla vista (in pratica, la tupla aggiornata non deve violare la clausola `WHERE`)

EX: `Viste with check option`

```
CREATE VIEW
PROFESSORIRICCHI(CODICE,NOME,COGNOME,STIPENDIO) AS
SELECT CODICE,NOME,COGNOME,STIPENDIO
FROM PROFESSORI
WHERE (STIPENDIO>=30000)
```

3.5 Common Table Expression (CTE)

Le CTE rappresentano viste temporanee che possono essere usate in una query come se fossero una vista a tutti gli effetti. A differenza delle viste, le CTE non esistono a livello di schema del DB.

```
WITH
NAME(Attributi) AS
SQL Query
```

Una CTE è valida solo per le query fatte dopo la sua creazione.

3.6 Assertion

Le asserzioni sono un costrutto per definire vincoli generici a livello di schema.

```
create assertion NomeAsserzione check Condizione
```

Consentono di definire vincoli non altrimenti definibili con i costrutti visti fin qui. Il vincolo può essere immediato o differito (ossia verificato al termine di una transazione).

EX: Asserzione

La tabella STUDENTI non può essere vuota.

```
CREATE ASSERTION TabellaValida
CHECK (
  (SELECT COUNT(*) FROM STUDENTI) >= 1
);
```

3.7 Costrutti Avanzati

3.7.1 Procedure/Stored Procedures

Frammenti di codice SQL, con la possibilità di specificare un nome, dei parametri in ingresso e dei valori di ritorno. Con le stored procedure si ha maggior efficienza, espressività e sicurezza.

```
CREATE PROCEDURE myPROC (IN param1 INT, OUT param2 INT)
SELECT COUNT(*) INTO param2
FROM tabella
WHERE name = param1;
mysql>> CALL myPROC("Test",@variable);
```

Le estensioni procedurali consentono di:

- Aggiungere strutture di controllo al linguaggio SQL (es. cicli, strutture condizionali `if then else`, etc).
- Dichiarare variabili e tipi di dato user-defined.
- Definire procedure sui dati avanzate, che sono ritenute "sicure" dal DBMS.

3.7.2 Trigger

Si usano per implementare comportamenti automatici.

EX: Trigger

Ogni mese, vengono rimosse tutte le righe presenti dalla tabella `ORDINI` e spostate nella tabella `ORDINI_PENDENTI`.

```
Create trigger Nome
[before/after] [insert/delete/update] on Tabella  #evento
[referencing Referenza] # variabili globali per aumentare
l'espressività del trigger
[for each Livello] # Livello può essere row (Il trigger agisce a
livello di righe) o statement (Il trigger agisce globalmente a
livello di tabella)
[when (IstruzioneSQL)]  #condizione
Istruzione/ProceduraSQL #azione
```

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
- Azione: sequenza di istruzioni SQL, talvolta procedure SQL specifiche del DBMS.

Ci sono due modalità di esecuzione:

- immediata: il trigger viene attivato e completato subito, prima o dopo l'operazione che lo ha causato.
- differita: il trigger non viene eseguito immediatamente, ma alla fine della transazione corrente. Attenderà fino a quando tutte le modifiche della transazione saranno completate per poi essere eseguito come passaggio finale.

EX: Trigger

```
CREATE TRIGGER CHECKAUMENTO
BEFORE UPDATE OF CONTO ON IMPIEGATO
FOR EACH ROW
WHEN (NEW.STIPENDIO > OLD.STIPENDIO * 1.2)
SET NEW.STIPENDIO=OLD.STIPENDIO * 1.2
```

- Modo: `before`.
- Evento: `update`.
- Livello: `row`.

3.7.3 Permessi

I permessi sono meccanismi di controllo di accesso alle risorse dello schema del DB. Di default, ogni risorsa appartiene all'utente che l'ha definita Su ciascuna risorsa sono definiti dei privilegi (grant):

Privilegi	
`insert` / `update` / `delete`	tabelle/viste
`select`	tabelle/viste
`references`	tabelle/attributi
`usage`	domini

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]
```

L'opzione `with grant option` consente di propagare il privilegio ad altri utenti del sistema.

Il comando `revoke` consente di revocare privilegi su una certa risorsa ad utenti specifici.

```
revoke Privilegio on Risorsa/e from Utente/i [cascade|restrict]
```

In SQL3 è possibile definire dei ruoli per regolare l'accesso alle risorse di un database.

Ruolo: raccoglitore di privilegi.

```
create role/set ruolo
```

EX: Creazione ruolo

```
create role analyst
grant select, insert on tabella clienti to analyst
grant analyst to marco
```

Il ruolo `analyst` ha accesso in lettura e inserimento sulla tabella `clienti`

4 MySQL

MySQL è un DBMS basato sul modello relazionale (RDBMS). Supporta gran parte dei costrutti di SQL 2.0, con trigger e viste aggiornabili. Supporta l'esecuzione di transazioni su un tipo particolare di tabelle (INNODB). Dispone di un proprio linguaggio di estensione procedurale per gestire le stored procedures. Non ha limiti espliciti sulla dimensione massima di un DB. Non esistono problemi dal punto di vista della concorrenza in termini di numeri massimi di connessioni simultanee al server MySQL.

Come interagire con MySQL:
- Riga di comando
- Interfaccia grafica
- Applicazioni/linguaggi di programmazione

4.1 Autenticazione

Via shell: `mysql –u <utente> –p <password>`.
La gestione dei dati degli utenti avviene attraverso la tabella `mysql.user`.
- Creare un nuovo utente (locale): `CREATE USER nome@localhost;`
- Impostare la password di un utente:

`SET PASSWORD FOR nome@localhost=PASSWORD("passwd");`
- Creare un nuovo utente con password:
  `CREATE USER nome@localhost IDENTIFIED BY "passwd";`
- Cancellare un utente: `DROP USER nome@localhost;`

4.2 Creazione database

- Creare un nuovo database: `CREATE DATABASE [IF NOT EXIST] nome_db;`
- Rimuovere un database: `DROP DATABASE [IF EXISTS] nome_db;`
- Vedere quali db sono presenti nel sistema: `SHOW databases;`
- Impostare il db corrente: `USE nome_database;`
- Assegnare/rimuovere un privilegio ad un determinato utente:
  - Comandi SQL: `REVOKE | INSERT`
  - Aggiornare tabelle `mysql.user`, `mysql.db`, `mysql.tables_priv` attraverso `INSERT/UPDATE`.

Per creare una tabella

```
CREATE [TEMPORARY] TABLE
    nome_tabella | nome_db.nome_tabella
    [definizione attributi]
    [opzioni]
    [select]
```

Con `temporary` si crea una tabella valida solo per la sessione corrente. Si può popolare una tabella con il risultato di una `select` da altre tabelle.

Ci sono due tipo di tabelle/*storage engine*:
- INNODB
  - supporta il sistema transazionale
  - supporta i vincoli di chiave esterni
  - maggiore robustezza ai guasti
- MyISAM
  - non supporta il sistema transazionale
  - maggiore efficienza
  - minore consumo di spazio su memoria secondaria

Un sistema transazionale garantisce le proprietà ACID.

Esempi di opzioni valide sulle tabelle:
- `ENGINE = TIPO_TABELLA (INNODB/MyISAM)`
- `AUTO_INCREMENT = <VALORE>` (valore da cui si inizia a contare)
- `AVG_ROW_LENGTH = <VALORE>`
- `CHECKSUM = {0|1}`
- `COMMENT = <STRING>`
- `MAX_ROWS = <VALORE>`

Definizione di attributi

```
Nome_colonna TIPO
[NOT NULL | NULL] [DEFAULT valore]
[AUTO_INCREMENT]
[UNIQUE | PRIMARY KEY]
[COMMENT "commento"]
```

Per definire i vincoli di integrità referenziale (solo con INNODB):

```
FOREIGN KEY (nome_colonna_interna)
REFERENCES nome_tabella_esterna
(nome_colonna_esterna)
[ON DELETE | ON UPDATE
RESTRICT | CASCADE | SET NULL |
NO ACTION ]
```

4.3 Tipi di dato

- numerici
  - `BIT`
  - `TINYINT [UNSIGNED][ZEROFILL]`
  - `SMALLINT [UNSIGNED][ZEROFILL]`
  - `MEDIUMINT [UNSIGNED][ZEROFILL]`
  - `INT [UNSIGNED][ZEROFILL]`
  - `BIGINT [UNSIGNED][ZEROFILL]`
  - `FLOAT [UNSIGNED][ZEROFILL]`
  - `DOUBLE [UNSIGNED][ZEROFILL]`
  - `DECIMAL [UNSIGNED][ZEROFILL]`
- temporali
  - `DATE`
  - `DATETIME`
  - `TIMESTAMP [M]`
  - `TIME`
  - `YEAR [(2,4)]`

Per conoscere data/timestamp correnti: `SELECT NOW();`, `SELECT CURTIME();`

- stringa di caratteri o byte
  - `CHAR(M) [BINARY | ASCII | UNICODE]`
  - `VARCHAR(M) [BINARY]`
  - `BINARY(M)`
  - `VARBINARY(M)`
  - `TINYBLOB`
  - `TINYTEXT`
  - `BLOB(M)`
  - `TEXT(M)`
  - `LONGBLOB`

EX: Creazione tabella in MySQL

```
CREATE TABLE IMPIEGATI (
Codice smallint auto_increment primary key,
Nome varchar(200) not null,
Cognome varchar(100) not null,
Salario double default 1000,
Anno date)
engine=innodb;
```

4.4 Comandi CRUD

- `insert`: popolamento dati

```
INSERT [LOW_PRIORITY|DELAY|HIGH_PRIORITY]
[INTO] nome_tabella [(nome_colonne,...)]
VALUES ({espressione | DEFAULT}, ...)
[ON DUPLICATE KEY UPDATE nome_colonna=espressione, ...]
```

- `replace`: popolamento dati

```
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] nome_tabella [(nome_colonna, ...)]
VALUES ({espressione | DEFAULT}, ...)
```

Estensione MySQL del costrutto `INSERT`. Consente di rimpiazzare delle righe preesistenti con delle nuove righe, qualora si verifichi un problema di inserimento con chiave doppia.

- `load`: popolamento dati

```
LOAD DATA [LOCAL] INFILE 'file.txt'
[REPLACE | IGNORE]
INTO TABLE nome_tabella
[FIELDS
[TERMINATED BY 'stringa']
[ENCLOSED BY 'stringa']
[ESCAPED BY 'stringa'] ]
[LINES
[STARTING BY 'stringa]
[TERMINATED BY 'stringa']]
[IGNORE numero LINES]
```

- `select`: ricerca dati

```
SELECT [ALL | DISTINCT | DISTINCTROW]
lista_colonne
[INTO OUTFILE 'nome_file' |
INTO DUMPFILE 'nome_file' ]
FROM lista_tabelle
[WHERE condizione]
[GROUP BY {nome_colonna}]
[HAVING condizione]
[ORDER BY {nome_colonna}]
[LIMIT [offset,] numero_righe]
```

- `delete`: cancellazione dati

```
DELETE [LOW_PRIORITY][IGNORE][QUICK]
FROM nome_tabella
[WHERE condizione]
[LIMIT numero_righe]
```

- `truncate`: rimuove tutto il contenuto

```
TRUNCATE nome_tabella
```

- `update`: aggiornamento di dati

```
UPDATE [LOW_PRIORITY][IGNORE]
SET {nome_colonna=espressione, ….}
WHERE condizione
```

4.5 Creazione di Trigger

Creazione di regole attive attraverso il costrutto di `TRIGGER`.

```
CREATE TRIGGER nome tipo
ON tabella FOR EACH ROW istruzioniSQL
```

Il tipo specifica l'evento che attiva il trigger: `BEFORE INSERT`, `BEFORE UPDATE`, `BEFORE DELETE`, `AFTER INSERT`, `AFTER UPDATE`, `AFTER DELETE`

EX: Definizione di trigger

```
CREATE TRIGGER upd_check
BEFORE INSERT ON Impiegati
FOR EACH ROW
BEGIN
IF NEW.Salario > 300 THEN
SET NEW.Salario=300;
END IF;
END;
```

4.6 Creazione di Viste

Creazione di regole attive attraverso il comando `VIEW`.

```
CREATE [OR REPLACE]
[ALGORITHM = (UNDEFINED | MERGE | TEMPTABLE)]
VIEW nome [(lista colonne)]
AS selectSQL
[WITH [CASCADED|LOCAL] CHECK OPTION]
```

`WITH CHECK OPTION` definisce visite aggiornabili.

4.7 Creazione di Stored Procedures

```
CREATE PROCEDURE nomeProcedura
([IN|OUT] nomeParametro tipo)
BEGIN
[dichiarazione di variabili locali]
[istruzioni SQL]
END;
```

Insieme di istruzioni SQL memorizzate nel DBMS, cui è associato un nome. Può ricevere parametri in input, può restituire più di un valore in output. Il corpo contiene istruzioni SQL.

```
CREATE PROCEDURE nomeImpiegato
(IN cod INT, OUT nomeI VARCHAR(200))
BEGIN
SELECT NOME AS NOMEI
FROM IMPIEGATI
WHERE (CODICE=cod);
END;

CALL nomeImpiegato(200,@var);
SELECT @var;
```

Creazione di Stored procedures	
Dichiarazione di variabili locali	`DECLARE a INT DEFAULT 0;`
Costrutti di selezione (`if ... then ... else`)	`Condizione THEN IstruzioniSQL [ELSE IstruzioniSQL] ENDIF;`
Costrutti iterativi (`WHILE` / `LOOP` / `REPEAT`):	`[nome] WHILE Condizione DO IstruzioniSQL END WHILE [nome];`

Cursori di query	`DECLARE nomeCursore CURSOR FOR selectSQL;`
	`OPEN nomeCursore`
	`FETCH nomeCursore INTO nomeVariabili;`
	`CLOSE nomeCursore`

I cursori consentono di eseguire query SQL e salvare il risultato (result set) in una lista. La lista risultante può essere iteratamente visitata attraverso il comando di `FETCH`.

EX: Stored procedure con cursori

```
CREATE PROCEDURE nomeImpiegato
(IN salarioMax INT, OUT valido BIT)
BEGIN
    DECLARE fine INT DEFAULT 0;
    DECLARE cur CURSOR FOR
    SELECT salario FROM IMPIEGATI;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET
fine=1;

SET valido=1;
OPEN cur;
ciclo: WHILE NOT fine DO
    FETCH cur INTO salarioCor;
    IF salarioCor > salarioMax THEN
        valido=0;
    END IF;
END WHILE ciclo;
END;
```

Questa procedura verifica se esistono impiegati con un salario maggiore di `salarioMax`, passato come input. Se trova almeno un impiegato con salario maggiore di `salarioMax`, imposta il parametro di output `valido` a 0, altrimenti a 1.

4.8 Gestione delle transazioni per tabelle INNODB.

Di default, la modalità `autocommit` è abilitata, quindi tutti gli aggiornamenti sono effettuati immediatamente sul database. Nel caso in cui gli autocommit siano disabilitati, è necessario indicare l'inizio della transazione con `START TRANSACTION` e terminarla con un comando di `COMMIT` o `ROLLBACK`.

EX: Transazione

```
SET AUTOCOMMIT = 0;
START TRANSACTION
INSERT INTO IMPIEGATO (Nome, Cognome, Salario)
VALUES ('Michele','Rossi',1200);
INSERT INTO IMPIEGATO (Nome, Cognome, Salario)
VALUES ('Carlo','Bianchi',1000);
COMMIT
```

MySQL offre quattro livelli di isolamento:

- `READ UNCOMMITTED`: sono visibili gli aggiornamenti non consolidati fatti da altri.
- `READ COMMITTED`:aggiornamenti visibili solo se consolidati (ossia solo dopo `COMMIT`).
- `REPEATABLE READ`: tutte le letture di un dato operate da una transazione leggono sempre lo stesso valore (comportamento di default).
- `SERIALIZABLE`: lettura di un dato blocca gli aggiornamenti fino al termine della transazione stessa che ha letto il dato (lock applicato ad ogni `SELECT`).

4.9 Utilities

```
set[global|session] transaction isolation level {
    READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE
}
```

Il tool MySQLDump consente di effettuare backup del contenuto di un database (o di tutti).

MySQLDump	
Backup di tutti i database	`mysqldump -single-transaction -all-database > nomefile`
Backup di uno specifico database	`mysqldump -single-transaction nomedb > nomefile`
Ripristino di un database (o tutti) da un file di backup	`mysql [nomedb] < nomefile`

5 Gestione delle Transazioni

EX: Sistema va in crash tra due operazioni che modificano tabelle

```
UPDATE ITEM SET Quantita=Quantita-1 WHERe (Codice=CodiceScelto));
! X-> CRASH !
INSERT INTO ORDINE(Data,Ordinante,ItemOrdinato) VALUES (NOW(),
```

Causa un problema di coerenza tra i dati

Le transazioni sono unità di lavoro elementari (insiemi di istruzioni SQL) che modificano il contenuto di unabase di dati.

EX: Transazione

```
start transaction
update SalariImpiegati
set conto=conto-10
where (CodiceImpiegato = 123)
if var > 0 then commit work;
else rollback work;
```

Le transazioni sono comprese tra una `start transaction` ed una `commit/rollback`.

Proprietà ACID delle transazioni:

- Atomicità: La transazione deve essere eseguita con la regola del "tutto o niente".
- Consistenza: La transazione deve lasciare il DB in uno stato consistente, eventuali vincoli di integrità non devono essere violati.
- 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 log e checkpoint. Il gestore della concorrenza carantisce isolamento in caso di esecuzione concorrente di più transazioni

.

DEF: **Schedule**
Dato un insieme di transazioni $T_1, T_2, ..., T_n$ di cui ciascuna formata da un certo insieme di operazioni di *writing* w_i e *reading* r_i[6], si definisce schedule la sequenza di operazioni di tutte le transazioni così come eseguite sulla base di dati:
$r_1(x)\ r_2(y)\ r_1(y)\ w_4(y)\ w_2(z)$

[6] EX: $T_1 = r_1(x)\ r_2(x)\ r_3(x)\ w_1(x)\ ...$

Lo schedule è l'insieme di tutte le operazioni delle transazioni eseguite sulla base di dati. Nell'insieme possono essere mescolate operazioni di transazioni diverse, ma deve essere sempre rispettato l'ordine interno delle singole transazioni. Lo schedule rappresenta l'esecuzione reale, mentre le transazioni rappresentano solo le intenzioni logiche di ciascun utente o processo.

DEF: **Schedule Seriale**
Uno schedule si dice seriale se le azioni di ciascuna transazione appaiono in sequenza, senza essere inframezzate da quelle di altre transazioni.

$$S = \{T_1, T_2, ...T_n\}$$

Le transazioni devono essere eseguite una alla volta ed essere completamente indipendenti l'una dall'altra[7]. In un sistema reale, le transazioni vengono eseguite in concorrenza per ragioni di efficienza / scalabilità.
L'esecuzione concorrente determina un insieme di problematiche che devono essere gestite.

Perdita di aggiornamento ($x = 3$)	
T_1	T_2
read(x)	
x=x+1	
	read(x)
	x=x+1
	write(x)
	commit work
write(x)	
commit work	

Table 3: Sia T_1 che T_2 scrivono 4

[7] Scenario non realistico e non probabile

Lettura Sporca ($x = 4$)	
T_1	T_2
read(x)	
x=x+1	
write(x)	
	read(x)
	commit work
rollback work	

Table 4: T_2 legge 4

Lettura Sporca ($x = 3$)	
T_1	T_2
read(x)	
	read(x)
	x=x+1
	write(x)
	commit work
read(x)	

Table 5: T_1 legge prima 3 poi 4

DEF: **Schedule Serializzabile**
Uno schedule S si dice serializzabile quando lo schedule corrente porta allo stesso risultato che otterebbe con uno schedule seriale S'.

Per implementare il controllo della concorrenza si usano i lock: per poter effettuare un'operazione su una risorsa è necessario aver precedentemente acquisito il controllo (lock) su di essa.
- Lock in **lettura**: accesso condiviso a più transazioni
- Lock in **scrittura**: mutua esclusione, solo una transazione alla volta

Su ogni lock possono essere definite due operazioni:
- Richiesta del lock in lettura/scrittura.
- Rilascio del lock (unlock) acquisito in precedenza.

5.1 Lock Manager

DEF: **Lock Manager**
Componente del DBMS responsabile di gestire i lock alle risorse del DB, e di rispondere alle richieste delle transazioni.

Metodi di ciascun oggetto x del DBMS:
- `State(x)` : stato dell'oggetto (libero/r_locked/w_locked)
- `Active(x)` : lista transazioni attive sull'oggetto
- `Queued(x)` : lista transazioni bloccate sull'oggetto

Azioni del lock manager
1. riceve una richiesta da una transazione T su un oggetto x
2. controlla la tabella stato
3. se la risposta è OK, aggiorna lo stato della risorsa, e concede il controllo alla transazione T
4. se la risposta è NO, inserisce la transazione T in una coda associata ad x.

5.2 Gestione delle transazioni

DEF: **2 Phase Lock** (2PL)
Una transazione, dopo aver rilasciato un lock, non può acquisirne un altro.

Una transazione prima acquisisce tutti i lock delle risorse di cui necessita.
Ogni schedule che rispetta 2PL è anche serializzabile perché 2PL garantisce l'ordine delle operazioni delle transazioni in modo tale da evitare conflitti che potrebbero portare a inconsistenze nei dati.
Si evita aggiornamento fantasma, lettura inconsistente, perdita di aggiornamento, ma non la lattura sporca.

DEF: **Strict 2PL**
I lock di una transazione sono rilasciati solo dopo aver effettuato le operazioni di `commit` / `abort` .

S2PL è usata solo in alcuni DBMS commerciali.
Uno schedule che rispetta lo S2PL eredita tutte le proprietà del 2PL, ed inoltre NON presenta anomalie causate da problemi di lettura sporca.

5.3 Deadlock

I protocolli 2PL e S2PL possono generare schedule con situazioni di deadlock. Per gestirli si usano 3 tecniche:

1. Uso dei timeout: ogni operazione di una transazione ha un timeout entro il quale deve essere completata, pena annullamento (`abort`) della transazione stessa. T_1: `r_lock(x,4000), r(x), w_lock(y,2000), w(y), commit, unlock(x), unlock(y)`
2. Deadlock avoidance: prevenire le configurazioni che potrebbero portare ad un deadlock tramite:
   - Lock/Unlock di tutte le risorse allo stesso tempo.
   - Utilizzo di time-stamp o di classi di priorità tra transazioni (può causare starvation!)
3. Deadlock detection: utilizzare algoritmi per identificare eventuali situazioni di deadlock, e prevedere meccanismi di recovery dal deadlock.

5.4 Timestamp (TS)

TS è un metodo alternativo al 2PL per la gestione della concorrenza che utilizza i time-stamp delle transazioni.

1. Ad ogni transazione si associa un timestamp che rappresenta il momento di inizio della transazione.
2. Ogni transazione non può leggere o scrivere un dato scritto da una transazione con timestamp maggiore.
3. Ogni transazione non può scrivere su un dato già letto da una transazione con timestamp maggiore.

Ad ogni oggetto x si associano due indicatori:

1. `WTM(x)` : timestamp della transazione che ha fatto l'ultima scrittura su x.
2. `RTM(x)` : timestamp dell'ultima transazione (ultima=con t più alto) che ha letto x.

Livelli di Isolamento	
read uncommitted	La transazione non emette lock in lettura, e non rispetta i lock esclusivi delle altre transazioni.
read committed	Richiede lock condivisi per effettuare le letture.
repeatable read	Applica S2PL anche in lettura
serializable	Applica S2PL con lock di predicato

6 NoSQL

DEF: **NoSQL**
Movimento che promuove l'adozione di DBMS non basati sul modello relazionale.
I sistemi NoSQ in genere:
- sono database distribuiti
- sono tool open source
- non dispongono di uno schema
- non supportano operazioni di join
- non implementano le proprietà ACID delle transazioni
- sono scalabili orrizzontalmente
- sono in grado di gestire grandi basi di dati
- supportano le repliche dei dati

6.1 Motivi della diffusione dei database NoSQL

- Gestione dei big data[8]. Le quattro V dei big data
  - Volume: grossi moli di dati
  - Velocità: stream di dati
  - Varietà: dati eterogenei, multi-sorgente
  - Valore: possibilità di estrarre conoscenza dai big-data
- Limitazioni del modello relazionale:
  - SQL assume che i dati siano strutturati
  - alcune operazioni non possono essere implementate in SQL
  - scalabilità orrizzontale dei DBMS relazionali

DEF: **Scalabilità**
Capacità di un sistema di migliorare le proprie prestazioni per un certo carico di lavoro, quando vengono aggiunte nuove risorse al sistema.
- Scalabilità **orrizzontale**: aggiungere più nodi al cluster.
- Scalabilità **verticale**: aggiungere più potenza di calcolo (RAM, CPU) a i nodi che gestiscono il DB.

- Teorema Cap

DEF: **Cap Theorem**
Il teorema di Brewer (CAP Theorem) afferma che un sistema distribuito può soddisfare al massimo solo due di queste proprietà:
- Consistency: Tutti i nodi della rete vedono gli stessi dati. Se l'utente A modifica il dato X sul server 1, e B legge X dal server 2, B legge l'ultima versione disponibile di X.
- Availability: Il servizio è sempre disponibile. Se un utente effettua una query sul server A o B, la query restituisce un risultato.

[8] Big data: moli di dati, eterogenei, destrutturati, difficili da gestire attraverso tecnologie tradizionali.

- Partion Tolerance: Il servizio continua a funzionare correttamente anche in presenza di perdita di messaggi o di partizionamenti della rete.

6.2 Proprietà Base dei Sistemi NoSQL

- *Basically Available*: i nodi del sistema distribuito possono essere soggetti a guasti.
- *Soft State*: la consistenza dei dati non è garantita in ogni istante
- *Eventually Consistent*: il sistema diventa consistente dopo un certo intervallo di tempo, se le attività di modifica dei dati cessano.

Il termine NoSQL identifica una varietà di DBMS non relazionali, basati su modelli logici differenti:

- database chiave-valore
- database document oriented (MongoDB)
- database graph oriented

7 MongoDB

MongoDB è un DBMS non relazionale basato su DB document-oriented. È un database organizzato in collezioni, le collezioni contengono liste di documenti. Ogni documento è un insieme di campi.

MongoDB	RDBMS
collezione	tabella
documento	riga
campo	colonna di una riga

Si usano comandi javascript tramite shell MongoDB oppure applicazioni che si collegano a mongoDB. Utilizza il linguaggio JSON come input/output delle query di aggiornamento o selezione.

EX: Documeto JSON in MongoDB

```
{"nome": "Mario",
"cognome": "Rossi",
"eta": 45,
"impiegato": false,
"salario": 1205.50,
"telefono": ["0243434", "064334343"],
"ufficio": [
    {"nome": "A", "via": "Zamboni", "numero": 7},
    {"nome": "B", "via": "Irnerio", "numero": 49}
]}
```

Comando	Azione
`mongod`	avvio server
`mongo`	avvio shell
`use provaDB`	utilizzo/creazione di un DB
`db.createCollection("circoli")`	creazione di una collezione vuota
Comandi shell MongoDB	
`show DBS`	mostra DB disponibili
`show collections`	mostra collezioni del DB
`show users`	mostra gli utenti del sistema
`show rules`	mostra il sistema di accessi
`show logs`	mostra i log disponibili

Un documento in MongoDB è un oggetto JSON.

Nella stessa collezione, è possibile inserire documenti eterogenei, ossia con strutture campo/valore differenti.

EX: Inserimento di un documento in una collezione

`db.NOMECOLLEZIONE.insert(DOCUMENTO)`

```
db.anagrafica.insert({"name": "Marco", "cognome":
"Rossi", "eta": 22})
db.anagrafica.insert({"cognome: "Rossi", "eta": 22,
"domicilio":["Roma", "Bologna"]})
db.anagrafica.insert({"name": "Maria", "eta": 25})
```

Ogni documento contiene un campo `_id`, che corrisponde alla chiave primaria della collezione. Il campo `_id` può essere definito esplicitamente, o viene aggiunto in maniera implicita da MongoDB. Può essere inserito esplicitamente, o aggiunto in automatico da MongoDB.

Rimozione di un documento da una collezione	
Comando	**Azione**
`db.collezione.remove`	svuota la collezione
`db.NOMECOLLEZIONE.remove(SELETTORE)`	Elimina dalla collezione tutti i documenti che fanno matching con il selettore

Selettore: Il selettore è un documento json che specifica un campo e un valore per filtrare i documenti in base a corrispondenze esatte.

EX: Cercare documenti con il campo nome "Mario" o cognome "Rossi"

```
db.utenti.find({
  $or: [
    { "name": "Mario" },
    { "cognome": "Rossi" }
  ]
})
```

Operatori di Confonto:

`$GT`	maggiore	`{"età":{$GT:30}}`
`$LT`	minore	`{"età":{$LT:30}}`
`$GTE`	maggiore o uguale	`{"età":{$GTE:30}}`
`$LTE`	minore o uguale	`{"età":{$LTE:30}}`
`$NE`	diverso	`{"età":{$NE:30}}`
`$IN`	uno tra molti valori	`{"nome":{$IN:["Mario","Lucia"]}}`

EX: Cercare utenti con età compresa tra i 18 e 30 anni

```
db.utenti.find(
    {"età":{
        $GTE: 18,
        $LTE: 30
    }
})
```

Selettori Logici

`$AND`	e	`{$AND:[{"età":{$GTE:18}},{"sesso":"M"}]}`
`$OR`	o	`{$OR:[{"età":{$GTE:18}},{"sesso":"M"}]}`
`$NOT`	non	`{"età":{$NOT:{$GT:30}}`

EX: Cercare utenti che siano uomini con almento 30 anni, oppire con nome Luca

```
db.utenti.find(
    {
        $OR:[
            {$AND:[
                {"sesso":"M"},
                {"età":{$GTE:30}}
            ]},
            {"nome":"Luca"}
        ]
    }
)
```

`$exists` controlla se un campo esiste o meno.

EX: Cerca documenti con il campo `telefono`

```
db.utenti.find(
    {"telefono":{
        $EXIST:true
    }
})
```

`update`	

`db.nomecollezione.update(`
`  selettore,campi`
`  )`
	Trova i documenti filtrati poi modifica tutti i campi inseriti (se non esiste lo aggiunge), ma cancella tutti quelli non specificati nell' `update`.
	`db.anagrafica.update(`
	`  {"name":"Mario"},`
	`  {"età":45}`
	`)`
	Il campo `"name":"Mario"` è cancellato[9] e viene inserito `"età":35`

`db.NOMECOLLEZIONE.update(`
`  SELETTORE,{$SET: CAMPI}`
`  )`
	Uguale alla forma sopra, ma i campi non specificati non vengono cancellati. Si modificano soli i camp inseririti.
	`db.anagrafica.update(`
	`  {"name":"Mario"},`
	`  {$set:{"eta":45}})`
	Nel documento relativo all'impiegato Mario, aggiorna il campo età ponendolo pari a 45

`db.NOMECOLLEZIONE.update(`
`  SELETTORE,{$PUSH:CAMPI}`
`  )`
	Aggiunge un elemento alla fine di un array specificato nel campo definito. Se il campo è un array, l'operatore `push` aggiunge un nuovo valore all'array esistente, altrimenti ne crea uno nuovo e aggiunge l'elemento.
	`db.anagrafica.update(`
	`  {"name":"Mario"},`
	`  {$push:{"eta":45}})`
	Nel documento relativo all'impiegato Mario, si aggiunge un nuovo campo età (array), settandolo pari a 45.

`$EACH` serve per inserire più valori nell'array in una sola operazione.

EX: `$EACH`

```
db.utenti.update(
    {"_id":"1"},
        {$PUSH:{
            "hobby":{
                $EACH: ["scrittura", "sport"]
            }
        }
    }
)
```

`find`	
`db.nomecollezione.find()`	Restituisce tutti i documenti presenti nella collezione.
`db.nomecollezione.find(condizone)`	Restituisce tutti i documenti, i cui campi rispettino la condizione espressa nella query.

`db.NOMECOLLEZIONE.find(`
`  SELETTORE,PROJECTION`
`)`
	Filtra e con il `projection` si inseriscono i campi che si vuole vengano mostrati.

EX: `projection`

Voglio solo i campi nomi ed età escludendo `_id`

```
db.utenti.find(
    {"nome":"Mario"},
    {"nome":true,"età":true,"_id":false}
)
```

Se voglio che il campo sia incluso, lo imposto a `true` altrimenti a `false`.

[9] Perché non specificato nell' `update`

Operatori	
`db.nomecollezione.find(...).sort(campi)`	`1` : ordinamento crescente, `-1` : ordinamento decrescente. In caso di più campi si guarda il primo. `db.anagrafica.find(...).sort({"età":1})`
`db.collezione.find(...).count()`	Conta i documenti
`db.collezione.find(...).distinct(campo,condizione)`	Restituisce un array con i valori distinti del campo `<campo>` per tutti i documenti che soddisfano la condizione `<condizione>`. `db.anagrafica.distinct("età", {"nome":"Marco"})`

È possibile i comandi MongoDB in uno script javascript, eseguito nella shell di MongoDB o in un applicazione esterna tipo node.js. Il file di script può contenere costrutti iterativi come `while`, `if`, `else`, `...`. I cursori vengono usati per scorrere il risultato di una query.

EX: Codice javascript che comunica con MongoDB

```
conn = new Mongo();
db = conn.getDB("tennis2");
db.createCollection("soci");
cursor = db.collection.find({"name"="mario"});
while (cursor.hasNext()) {
    printjson(cursor.next());
}
cursor = db.collection.find({"name"="mario"});
if (cursor.hasNext()) {
    print("Trovato!");
}
```

7.1 Correlazioni tra collezioni

Non esistono vincoli di integrità referenziale tra collezioni/tabelle.

- Le correlazioni possono essere costruite esplicitamente mediante campi "replicati" tra più collezioni.

```
db.circoli.insert({"nome":"tennis2000", "citta": "Bologna" })
db.soci.insert({"nome":"Mario", "cognome":"Rossi",
"nomeCircolo":"tennis2000"})
```

- Le associazioni uno-a-molti, o molti-a-molti, tra documenti di diverse collezioni possono essere rappresentate sfruttando il fatto che in MongoDB il valore di un campo può essere anche un array, o una struttura complessa (es. documento annidato).

```
db.soci.insert({"name:"Mario", cognome:"Rossi",
circolo:{...})
```

Come fare query che implementano il join tra collezioni

1. usi lookup table
2. usi due query (prima trovi l'id, poi con quello fai la seconda)

7.2 Aggregazione di dati

L'operatore `aggregate` consente di implementare una pipeline di operazioni da eseguire su una base di dati. Ad ogni passo, vengono eseguite operazioni che prednono in input dei documenti json e producono in output documenti json.
`collezione->operatore1->operatore2->risultato`

Operazioni	
`$geonear`	Ordina i documenti dal più lontano al più vicino rispetto ad una posizione data.
`$match`	Seleziona solo alcuni documenti che soddisfano le condizioni
`$project`	Seleziona i campi prescelti
`$group`	Raggruppa in base a uno o più campi.
`$sort`	Ordina il json in base ad alcuni campi.
`$out`	Scrive l'output su una collezione.
`$lookup`	Consente di effettuare il `join` tra collezioni che appartengono allo stesso DB. `{ "lookup":{ "from": collezione su cui fare il join, "local field": campo dalla collezione di partenza, "foreign field": campo della collezione del from, "as": nome del campo di destinazione } }`

EX: `aggregate`

```
db.anagrafica.aggregate([
  {
      $match: { "name":"A"}
  },
  {
      $group: {
          "_id":"$customId",
          "total":{$sum: "$amount"}
      }
])
```

1. Fa il `match` prima: prende tutti i documenti con nome `A`

```
{"_id":1,"name":"A","customId":101,"amount":10},
{"_id":3,"name":"A","customId":101,"amount":15},
{"_id":4,"name":"A","customId":103,"amount":5},
```

2. Raggruppa i documenti in base al campo `customId`. Verrà creato un documento per ogni valore unico di `customId`.

```
{"_id":101,"total":25},
{"_id":103,"total":5},
```

8 Progettazione di basi di dati

Le basi di dati implementate fino ad ora si basavano su uno schema relazionale già definito. Come si realizza un sistema informativo da zero?

- Problema 1: **dimensionamento del problema**
  Un DB di un sistema informativo di medie dimensioni può contenere decine di tavelle
- Problema 2: **analisi dei requisiti**
  Identificare specifiche, dati di interesse e operazioni da gestire
- Problema 3: **Traduzione nel modello logico** (relazionale)
  Passare da specifiche informali (testo scritto) a delle tabelle vere e proprie

Senza una buona progettazione, possono emergere anomalie ed errori nella fase di trazione del modello logico, come le ridondanze.

Esistono metodologie consolidate per progettare una buona base di dati a partire dai suoi requisiti.

Schema di vita di un sistema informativo
Studio di fattibilità
Raccolta/analisi dei requisiti
Progettazione
Implementazione
Validazione
Funzionamento

Si raccolgono le informazioni sulle specifiche dei requisiti sui dati (testo grezzo), poi le informazioni sulle specifiche delle operazioni sui dati.

8.1 Analisi dei requisiti

Analisi dei Requisiti	
Fasi della progettazione	**Risultati**
Progettazione concettuale	schema concettuale
Progettazione logica	schema logico
Progettazione fisica	schema fisico

8.1.1 Progettazione Concettuale

Ci si focalizza sul contenuto informativo dei dati ad alto livello di astrazione (senza concentrarsi sull'implementazione). Si produce un modello concettuale indipendente dallo schema logico e dal DBMS in uso.
Uno schema concettuale può essere prodotto con un modello E-R o con UML

8.1.2 Progettazione Logica

Traduzione dello schema concettuale in tabelle, ottimizzazione dello schema logico ottenuto.
Dopo aver ottenuto lo schema logico, è neessario analizzare la qualità del prodotto finale:

- si rimuovono le ridondanze[10]
- si analizzano le prestazioni: si controlla se il costo delle singole operazioni rendono il prodotto efficiente.

[10] Questo processo si chiama normalizzazione

8.1.3 Progettazione fisica

Si descrivono le strutture per la memorizzazione di dati su memoria secondaria, e l'accesso efficiente ai dati:

- strutture ad albero
- strutture sequenziali
- strutture ad acesso diretto (hash)

8.2 Raccolta dei requisiti

La raccolta dei requisiti consiste nella completa indiviudazione dei problemi che il sistema informativo da realizzare deve risolvere e le caratteristiche che il sistema informativo deve avere: quelle dei **dati** e quelle delle **applicazioni**.

Queste informazioni vengono raccolte

- dagli utenti dell'applicazione con interviste e documentazioni
- da documentazioni esistenti
  - normative
  - procedure aziendali
  - regolamenti interni
- realizzazioni/applicazioni preesistenti

8.2.1 Come trovare i dati da gestire e le operazioni sui dati consentite

1. Produrre un documento di specifica (testo grezzo). Dato che il linguaggio naturale è fonte di ambiguità e fraintendimenti bisogna:
   - scegliere il corretto livello di astrazione
   - standardizzare la struttura delle frasi
   - evitare frasi contorte
   - individuare omonimi/sinonimi
   - esplicitare il riferimento tra termini

   Può essere utile scomporre le specifiche in frasi omogenee, relative agli stessi concetti
2. Costruire un glossario dei termini che contiene descrizioni, sinonimi e collegamenti

Termine	Descrizione	Sinonimi	Collegamenti
Partecipante	…	…	…
Docente	…	…	…
Corso	…	…	…

3. Definire le operazioni sui dati. Questa fase è utile per
   - Verificare la completezza dei modelli sviluppati nella fase di progettazione
   - Valutare le prestazioni dei modelli sviluppati nella fase di progettazione
   - Fornire linee guida per l'implementazione dei dati (procedure per le operazioni)

9 Diagramma Entità-Relazionale

Sviluppato nella fase di progettazione concettuale, è indipendente dal modello logico (relazionale e non) e dal DBMS in uso.