APPUNTI **SQL**

Francesco De Lucchini
Università di Pisa
2022

Concetti di base

Introduzione

SELECT Attributi → Se li voglio tutti uso *
FROM Relazione
WHERE Condizioni; → **Attenzione al ; (terminatore di default)**

Ordine di esecuzione: FROM → WHERE → GROUP BY → HAVING → SELECT → DISTINCT

Una condizione più articolata può essere formata tramite i connettivi logici **AND OR NOT** oppure specificando un intervallo di valori con *X* **BETWEEN** *Inf* **AND** *Sup*. Per gestire i **valori nulli**, che in un confronto falliscono sempre e rendono il codice poco leggibile, posso usare **IS NOT NULL** oppure **IS NULL**.

In SQL le **stringhe** sono racchiuse tra singoli apici e possono essere confrontate tramite l'operatore l'uguale = , diverso <> oppure tramite la clausola **LIKE** con i placeholder **%** e **_** i quali esprimono rispettivamente *n* caratteri qualsiasi ed *un* caratttere qualsiasi, ad esempio la condizione **WHERE** Targa **LIKE** 'AA___AA' **AND** Nome **LIKE** 'F%' restituisce i proprietari di auto targate AA con tre numeri qualsiasi il cui nome inizia per F.

Un result set non è una relazione e può dunque contenere **duplicati**, che possono essere rimossi utilizzando la clausola **DISTINCT** posizionata dopo il SELECT, questa va utilizzata solo quando strettamente necessario poiché ha complessità quadratica.

Per **ridenominare** una colonna di un result set posso usare la clausola ColVecchia **AS** ColNuova, mentre per **combinare** più colonne posso usare l'operatore +, ad esempio Città + ', ' + Via + ', ' + CAP **AS** Indirizzo

Per **ordinare** i valori assunti da delle colonne di un result set posso usare **ORDER BY** Col_1 **ASC**, ..., Col_n **DESC** → In ordine di precedenza!

Gestione delle date

Una data (tipo **DATE**) è una stringa del formato 'YYYY-MM-DD' a cui è possibile applicare la clausola **BETWEEN** o gli operatori di confronto matematici per stabilirne un ordine cronologico, se volessi un orario più preciso posso usare il tipo di dato **TIMESTAMP** che in aggiunta al tipo DATE offre anche 'HH:MM:SS', per ottenere la data attuale basta semplicemente usare la variabile di sistema **CURRENT_DATE**.

Per estrarre facilemente informazioni da una data posso usare le funzioni **YEAR()**, **MONTH()** e **DAY()**

In SQL i tipi DATE non si possono né sommare né sottrarre tramite gli operatori matematici + e - , se voglio la differenza tra due date devo utilizzare la funzione **DATEDIFF(**Recent, Old**)** che restituisce il numero di giorni che separa le due date specificate. Se invece volessi sommare o sottrarre un **intervallo di tempo** posso farlo con *Data* +/- **INTERVAL** *X* **one of[YEAR**, **MONTH**, **DAY]** dove *X* è un intero, esistono anche le funzioni **DATE_ADD(**DATE, INTERVAL**)** e **DATE_SUB(**DATE, INTERVAL**)** che portano lo stesso esatto risultato.

Operatori di aggregazione

Eventualmente combinati con il DISTINCT, sono operatori con il compito di ridurre l'intero result set in un unico scalare:

- **Conteggio** **COUNT(**Attributes**)**
- **Somma** **SUM(**Attributes**)**
- **Media** **AVG(**Attributes**)**
- **Massimo / minimo** **MAX** / **MIN** **(**Attributes**)**

Operatori di join

Più dettagli sul comportamento di questi operatori nel documento di teoria

- A **CROSS JOIN** B
- A **NATURAL JOIN** B → Se volessi usare solo un sottoinsieme degli attributi omonimi dovrei scrivere A **JOIN** B **USING Attributes**
- A **INNER JOIN** B **ON Condition**
- A ***one*** **of[LEFT**, **RIGHT**, **FULL]** **OUTER|opt** **JOIN** B **ON Condition**

Nel caso il join comporti uno o più attributi omonimi nel result set per accedervi è necessario utilizzare **alias**, ossia sinonimi dichiarati nella clausola FROM, ad esempio: FROM RELAZIONE Alias.

Operatori di raggruppamento

Partizionano un'insieme di record in gruppi di record in cui uno o più attributi *(detti di raggruppamento)* sono uguali, ogni gruppo collassa sempre in un unico record perciò tutti gli attributi che non sono in quelli di raggruppamento *(o non ne dipendono funzionalmente)* non possono essere proiettati se non tramite un'operatore di aggregazione.

Possono essere espresse delle **condizioni sui gruppi**, controllate gruppo per gruppo e non record per record, nella clausola **HAVING** solamente tramite operatori di aggregazione.

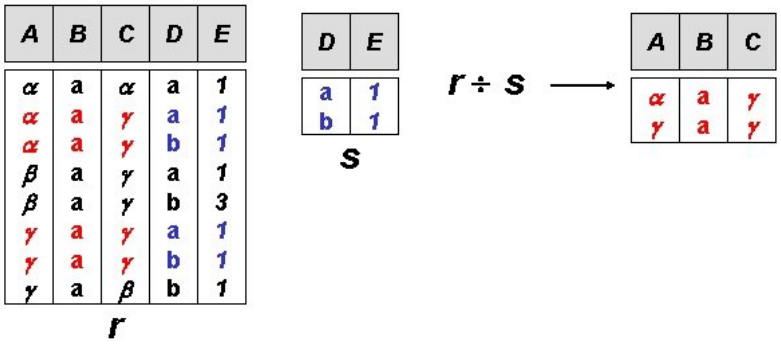
Operatori sugli insiemi

È possibile esprimere query utilizzando gli operatori insiemistici **UNION**, **INTERSECT** ed **EXCEPT**, i quali intuitivamente svolgono le omonime operazioni algebriche su due result set.

DIVISIONE INSIEMISTICA

Salva nel result set i record del numeratore che si legano con tutti i record del denominatore, è utile per interrogazioni che contengono condizioni esaustive *(tutti)* ed in SQL può essere replicata con:

- Doppio NOT EXISTS
- Raggruppamento e conteggio



DIFFERENZA INSIEMISTICA

Esprime il completamento di un insieme rispetto ad un altro, in SQL può essere replicata con:

- NOT EXISTS
- Outer join e IS NULL
- Subquery e NOT IN

Costrutti dell'SQL

Query

Una **query** è un'interrogazione al database ed esiste solamente quando viene eseguita.

View

Una *(virtual)* **view** è una query precompilata che viene memorizzata nel server, pronta per essere eseguita sull'istanza corrente del database, una vota terminata l'esecuzione ne viene subito cancellato il risultato.

Una **materialized view** è una view il cui risultato viene memorizzato nel server, è immediatamente disponibile per le interrogazioni e dunque garantisce prestazioni superiori a discapito della ridondanza e della pesentezza dei dati, la vedremo più in dettaglio in seguito.

Derived table

È una query incapsulata nel FROM che viene **calcolata all'inizio dello statement**, salvata in memoria ed eliminata alla fine di esso, per essere utilizzata deve essere necessariamente dotata di alias.

Subquery

È una query annidata nelle clausole WHERE o SELECT di una query esterna, è utilizzata in alternativa ai join per creare query articolate mantenendo il codice leggibile, ogni subquery può infatti vedere solamente tutto ciò che è stato dichiarato al di fuori di essa.

NONCORRELATED SUBQUERY

Produce un result set non dipendente dalla query esterna, si dice perciò non correlata ad essa, vengono eseguite per prime.

CORRELATED SUBQUERY

Produce un result set per ogni record della query esterna, è utile se combinata con le clausole EXISTS o NOT EXISTS.

Per legare la subquery alla query esterna posso usare:

- Se la subquery restituisce un insieme di record posso usare **IN** o **NOT IN** per controllare se un record ne appartiene o meno oppure posso usare i modificatori **ANY** o **ALL** per imporre che una condizione valga per almeno uno o tutti i record di un insieme
- Se la subquery restituisce un singolo record posso usare gli operatori **=** e **<>** per controllare che un record sia uguale o diverso dal risultato
- Se la subquery usa un operatore di aggregazione *(**scalar subquery**)* utilizzo i classici operatori di confronto in base al tipo del valore restituito
- Se la subquery è di tipo correlato ha senso usare i costrutti **EXISTS** o **NOT EXISTS** grazie ai quali un record va nel result set se esiste o non esiste un insieme

Common table expression (CTE)

Sono result set dotati di identificatore che possono essere usati prima di una query per costruire risultati intermedi, vengono dunque salvati in memoria con un nome, utilizzati immediatamente e distrutti al termine dello statement.

```
WITH  Name_1 AS (query_1) , ...
      Name_n AS (query_n) → Può a sua volta usare le CTE precedenti!
Main query;
```

Temporary table

Sono tabelle accessibili da qualsiasi parte del codice che hanno **durata della sessione**, sono solitamente usate per evitare di impiegare cursori in pezzi di codice dichiarativi.

Stored procedure

Sono programmi compilati, salvati nel DBMS ed eseguibili su chiamata, questo consente il passsaggio di parametri, il mascheramento del codice ed inoltre è ottimale per la rete, dato che non viene passato alcun codice.

```
-- Procedura standard ogni volta che si utilizza la clausola CREATE
DROP PROCEDURE IF EXISTS Procedura;
DELIMITER !! → Finchè non lo ritrova non compila
CREATE PROCEDURE Procedura() → Inserire qui gli eventuali parametri
BEGIN
    ...
END
    !! DELIMITER ; → Reimposto al valore di default
CALL Procedura(); → Eseguo la procedura
```

I **parametri in ingresso** sono dichiarati dopo la clausola IN (che è la clausola di default e può essere omessa), sono solitamente nominati con un underscore all'inizio e possono essere letti ma non modificati.

I **parametri in uscita** sono dichiarati dopo la clausola OUT e sono solitamente nominati con un underscore alla fine.

Cosa posso mettere tra le clausole begin ed end ?

--Query e CTE → Stampate a video o redirette dentro a variabili

--VARIABILI LOCALI

Vengono dichiarate tutte all'inizio della procedura tramite la clausola

```
DECLARE var TYPE(Size|opt) [DEFAULT value]; → Se non specificato è NULL
```

Per modificarne una alla volta uso **SET** var = Letterale o (Query scalare)
Per modificarle in gruppo uso:

```
SELECT A, ..., N FROM ...
INTO varA, ..., varN
LIMIT 1; → Tronca il result set ad una riga se la query non è scalare
```

--VARIABILI USER DEFINED

Sono variabili globali che esistono fintantoché l'utente che le ha definite è connesso al server, si definiscono tarmite **SET** @var = Letterale; Posso usare una variabile user defined come argomento di una chiamata ad una stored procedure senza prima vaerla definita da nessuna parte, sarà definita automaticamente dal momento della chiamata.

Per stampare una variabile o un letterale uso **SELECT** ...;

--ISTRUZIONI CONDIZIONALI

```
IF ... THEN ...;
ELSIF ... THEN ...;
ELSE ...;
END IF;
```

```
CASE WHEN ... THEN ...; → Non si usa il BREAK ma il ;
     ...
     WHEN ... THEN ...; → Non è previsto un WHEN di default
END CASE; → Se non entro in nessun WHEN è un errore di compilazione !
```

--ISTRUZIONI ITERATIVE

```
WHILE ... DO ... END WHILE;
```

Prima controlla la condizione e poi, se è vera, esegue il corpo del ciclo

Se ad ogni ciclo volessi aggiungere dei caratteri ad una stringa posso farlo tramite la funzione **CONCAT**(Destinazione, ' ', Sorgente)

```
REPEAT ... UNTIL ... END REPEAT;
```

Prima esegue il corpo del ciclo poi controlla la condizione e **se è falsa continua**, questo perché rappresenta una condizione di uscita (se è vera esce dal ciclo)

```
loop_label: LOOP ... END LOOP;
```

Esegue il corpo del ciclo aspettandosi sempre una tra le seguenti istruzioni: **LEAVE** loop_label oppure **ITERATE** loop_label

--CURSORI

Un cursore è una struttura dati mascherata che aggancia un result set dal primo record come farebbe un puntatore, si dichiarano solo immediatamente dopo la dichiarazione di tutte le variabili locali tramite:

```
DECLARE cursore CURSOR FOR query; → Non scalare
OPEN cursore;
FETCH cursore INTO variabili; → Legge una riga e passa alla prossima
CLOSE cursore;
```

--HANDLER

Sono dei gestori che al verificarsi di una certa situazione eseguono del codice, si dichiarano solo immediatamente dopo i cursori tramite:

```
DECLARE CONTINUE HANDLER FOR situazione → Una situazione da gestire sempre
```

nel caso di cursori è **NOT FOUND**, la quale segnala al fine del cursore.

CONTINUE significa che il programma non si interrompe quando viene attivato

```
--SEGNALAZIONE DI ERRORI

SIGNAL SQLSTATE '45000' → Codice di errore generico, provoca l'abort!
SET MESSAGE_TEXT = 'Errore: descrizione di esempio...'
```

STORED FUNCTION

È una stored procedure che restituisce sempre uno scalare, per essere chiamata non si scrive più CALL ma solamente il suo identificatore, un esempio di stored function sono gli operatori di aggregazione.

Se due chiamate con gli stessi parametri ritornano sempre lo stesso risultato la funzione è **deterministic**, in questo caso le prestazioni sono maggiori poiché il DBMS salva nella cache i risultati ed i parametri con i quali sono stati ottenuti.

```
CREATE FUNCTION Funzione() → Inserire qui gli eventuali parametri
RETURNS TYPE one of[DETERMINISTIC, NOT DETERMINISTIC]

BEGIN
    ...
    RETURN var o letterale del tipo TYPE;
END
Funzione(); → Eseguo la funzione
```

TRIGGER

I trigger sono dei componenti che reagiscono a cambiamenti scatenati da istruzioni DML, hanno il compito di gestire i vincoli di integrità referenziale e di aggiornare le ridondanze con sincronizzazione.

```
CREATE TRIGGER Trigger
BEFORE      Controlla il trigger PRIMA della modifica sulla base di dati
AFTER       Rappresenta una conseguenza su un'altra relazione da eseguire
            DOPO la modifica sulla base di dati
one of[INSERT, UPDATE, DELETE] ON Relazione → Al massimo un trigger a
FOR EACH ROW                                  relazione per tipologia

BEGIN
    NEW.Colonna → Usato per riferirsi al record preso in considerazione
END
```

EVENT

Gli event sono dei componenti che reagiscono al raggiungimento di certi istanti temporali, servono per svolgere operazioni che risulterebbero troppo dispendiose se eseguite in maniera sincrona con un trigger, ci si accontenta dunque di un valore differito, ossia non sincronizzato con il valore che effettivamente avrebbe calcolandolo ogni istante.

```
SET GLOBAL EVENT_SCHEDULER = ON; → Componente che gestisce gli event, se
                                   non è attivato non funzione nessun event
CREATE EVENT Evento

    Configurazione ricorrente
ON SCHEDULE EVERY X [YEAR, MONTH, DAY, HOUR, MINUTE, SECOND]
STARTS|opt 'Data inizio'
ENDS|opt 'Data fine'

    Configurazione ad esecuzione singola
ON SCHEDULE AT 'Data esecuzione'
ON COMPLETITION PRESERVE → Tiene il codice quando terminato

DO BEGIN ... END
```

MATERIALIZED VIEW

Sono delle tabelle **permanenti**, ossia delle normali tabelle salvate in memoria ed utilizzate per memorizzare risultati di operazioni che impiegherebbero troppo tempo per essere eseguite al momento della richiesta. Sono **ridondanti** in quanto salvano in memoria elaborazioni dei dati grezzi della base di dati (**raw data**), sono tuttavia necessarie per ottenere una risposta **rapida** anche se **non aggiornata** a interrogazioni complesse.

La base di dati è in costante aggiornamento dunque **più passa il tempo più una materialized view**, che rappresenta un certo istante nel tempo della base di dati, **diventa inutile**. È dunque necessario aggiornare periodicamente le materialized view, questo può essere fatto in vari modi:

- **IMMEDIATE (IN SYNC)**
  La materialized view viene mantenuta **costantemente aggiornata** tramite trigger sulle tabelle coinvonlte.

- **DEFERRED**
  L'aggiornamento avviene ad **intervalli regolari** nel tempo tramite un evento che chiama una stored procedure la quale legge i cambiamenti da apportare da una tabella di log ausiliaria.

- **ON DEMAND**
  L'aggiornamento avviene **solamente su richiesta** tramite chiamata ad una stored procedure.

Questi ultimi due possono seguire due possibili politiche di aggiornamento:

- **FULL REFRESH**
  La materialized view viene svuotata e viene ripopolata **da zero**.

- **INCREMENTAL REFRESH**
  I cambiamenti della base di dati che riguardano dati nella materialized view vengono salvati in una **tabella di log**, in questo modo è possibile **aggiornare solo la parte vecchia** e non buttare via tutto. Quest'utlimo si suddivide in tre modalità:

  - **PARTIAL**
    Viene **processata solo una parte del log**, la materialized view rimane vecchia ma meno vecchia di quanto lo era prima, è realizzato tramite una stored procedure che prende come input una data, la quale rappresenta fino a che momento aggiornare la materalized view.

  - **COMPLETE**
    Viene **processato l'intero log**, ossia applico tutti i cambiamenti

  - **REBUILD**
    Corrisponde ad un full refresh

La tabella di log deve contenere i dati sufficienti per poter aggiornare la materialized view, l'accesso ai raw data deve essere limitato il più possibile, al più può essere un join su chiave primaria (costo costante).

In una log table **vengono copiati tramite trigger tutti i record che hanno subito modifiche che rendono la materialized view non più aggiornata**, più altri attributi calcolati a seconda dello scopo della materialized view.

È buona norma **separare le log table in base al tipo di modifica** effettuata, ad esempio crearne una specifica per inserimenti ed una per cancellazioni, e **separare le log table in base alle tabelle delle quali tengono traccia**.

Funzionalità di Data Analytics in SQL

Si basano sul concetto di **funzioni a finestra**, ossia funzioni che affiancano a ciascun record un valore scalare ottenuto tramite un operazione eseguita su un'insieme di record logicamente connessi ad esso, questo insieme è detto **partition** e normalmente è diverso per ogni record, ma può anche essere uguale per tutti i record e può anche essere definito su attributi non proiettati.

Ne fanno parte le **funzioni di aggregazione**, con l'aggiunta del modificatore **WITH ROLLUP**, usato in combinazione con group by, il quale aggiunge al termine del result set un valore **super aggregato**, ossia la somma di tutti i valori aggregari, non può essere combinato con order by.

Ne fanno parte anche **funzioni di non aggregazione**, ossia che guardano il result set senza però farlo collassare, queste sono sempre espresse tramite la clausola **OVER**, il senso logico di questa clausola è:
Funzione scalare (applicata) **OVER** (Window (definita da **PARTITION BY**))

Ad esempio ... AVG(Parcella) OVER (PARTITION BY Specializzazione) affianca a ciascun record la media della sua specializzazione, ossia esegue la funzione scalare non sull'intero result set ma solo su una piccola finestra, che in questo caso è la specializzazione del record che si considera. Le principali funzioni scalari che possiamo applicare sono:

ROW_NUMBER() Come suggerisce il nome numera le righe della partizione.

RANK() Stila una classifica in base ad un criterio, spesso ordinato tramite ORDER BY, più alto è il criterio e più basso è il rank, dunque più alta è la posisione in classifica. Ai parimerito viene assegnata la stessa posizione (rank), che poi viene saltata nelle posizioni successive, ad esempio ci possono essere due primi ed un terzo (saltando il secondo), nel caso non volessi saltare il secondo dovrei usare la funzione **DENSE_RANK**(). È anche possibile effettuare rank multipli semplicemente ripetendo due o più volte la funzione.

Se il result set è dotato di ordinamento posso usare la funzione **LAG**(Attributo, k) per affiancare ad ogni record il valore di un secondo record posto k posizioni prima nella partizione (o k posizioni dopo se uso **LEAD**(Attributo, k)), se il valore spostato di k posizioni non esiste viene insertito NULL.

Le ultime due funzioni che vediamo operano su **frame**, ossia sottoinsiemi di partition creati prendendo *n* righe prima ed *m* righe dopo la riga corrente. Se non specificato dall'utente di **default** il frame è tutta la partizione se non viene usato ORDER BY, tutte le righe prima della corrente altrimenti.

FIRST_VALUE() e **LAST_VALUE**() come dice il nome prendono il primo o l'ultimo record di un frame, che può essere definito in due modi:

- Per righe **ROWS BETWEEN ... AND ...**
- Per range **RANGE BETWEEN ... AND** ...

Nota: Il loro utilizzo è mutualmente esclusivo.

Al posto dei puntini posso esprimere:

- La riga corrente CURRENT ROW
- La riga finale della partition UNBOUNDED FOLLOWING
- La riga iniziale della partition UNBOUNDED PRECEDING
- Un numero di righe arbitrario, ad esempio 3 PRECEDING o 2 FOLLOWING

Data Manipulation Language in SQL

Tipi di dati

CARATTERI

- **CHAR** Singolo
- **VARCHAR** (len) Vengono allocati al massimo len (255 MAX) caratteri

Se volessi più caratteri uso il tipo **TEXT** (proprietario di MySQL)

NUMERI ESATTI (Rappresentati come letterali)

- **TINYINT o BOOLEAN**
- **SMALLINT** (len)
- **INTEGER**
- **BIGINT**
- **NUMERIC** (Precisione, Scala) La precisione è in numero di cifre totali
- **DECIMAL** (Precisione, Scala) La precisione è il minimo numero di cifre

La scala è in entrambi il numero esatto di cifre dopo la virgola

NUMERI APPROSSIMATI (Rappresentati in virgola mobile) → Cercare di evitare

- **FLOAT** (Precisione binaria) È il numero di cifre della mantissa
- **REAL**
- **DOUBLE PRECISION**

TEMPO

- **DATE** Rappresenta Anno-Mese-Giorno
- **TIME** (Precisione) Rappresenta Giorno-Minuto-Secondo
- **TIMESTAMP** (Precisione) Rappresenta Date e Time uniti

La precisione è in il numero di cifre dopo la virgola dei secondi

- **INTERVAL A**(Precisione) **TO B**(Precisione)

Dove A e B variano da **YEAR** to **MONTH** a **DAY** to **SECOND**

OGGETTI BINARI

- **BLOB** Binary Large OBject
- **CLOB** Character Large OBject

Gestione degli schemi

CREATE

- **SCHEMA** Crea lo schema di un database
- **DOMAIN** Crea nuovi tipi di dato
- **TABLE** Crea una relazione nella quale è possibile specificare:
  - **Vincoli intrarelazionali**
    - **UNIQUE**
    - **NOT NULL**
    - **PRIMARY KEY**
  - **Vincoli interrelazionali**
    - Attribute **REFERENCES** Relation(SuperKey)
    - **FOREIGN KEY** Attribute **REFERENCES ...**
      - **ON [DELETE|UPDATE]**
        - **CASCADE**
        - **SET NULL**
        - **SET DEFAULT**
        - **NO ACTION**
  - **Valori di default**
    - **NULL**
    - **USER**
    - Valore personalizzato del dominio
  - Motore di query e set di caratteri consentiti, di default sono:
    - **ENGINE =** InnoDB
    - **DEFAULT CHARSET =** latin1

ALTER

Permette la modifica dei costrutti creati con l'istruzione CREATE, se modifico un vincolo l'istanza della relazione deve già soddisfarlo.

DROP

Permette l'eliminazione dei costrutti creati con l'istruzione CREATE, questa può essere a cascata oppure ristretta.

TRUNCATE

Elimina ogni record da una relazione senza eliminare la relazione stessa.

Gestione delle istanze

INSERT

```
INSERT INTO Relation[(AttributeList)] [VALUES ('Record1'),... | Query];
```

Se non specifico degli attributi questi prendono il valore di default

DELETE

```
DELETE FROM Relation [ WHERE Condizione ]
```

UPDATE

```
UPDATE Relation SET Attribute = ... [ WHERE Condizione ]
```

REPLACE

```
REPLACE INTO Relation[(AttributeList)] [VALUES ('Record1'),... | Query];
```

Esegue un inserimento se non collide su chiave primaria, altrimenti esegue un aggiornamento.

Nelle istruzioni DELETE e UPDATE se nella condizione uso una subquery non posso mettere nel from la stessa tabella che elimino per il concetto di lock esclusivo, allora posso usare una derived table o il join anticipato

Gestione delle transazioni

Una transazione è un insieme di operazioni da considerarsi indivisibile, in SQL può comprendere interrogazioni, inserimento di nuovi valori, aggiornamento di valori esistenti e molto altro... Le istruzioni fondamentali per gestire una transazione sono:

- **BEGIN** Specifica l'inizio della transazione
- **COMMIT** Esegue tutte le operazioni dall'ultimo BEGIN
- **ROLLBACK** Rinuncia ad eseguire le operazioni dall'ultimo BEGIN

Per le transazioni che compiono solo operazioni di lettura solitamente il DBMS permette di scegliere quale **livello di isolamento** adottare, in base alla precisione che deve avere la transazione:

- **Read uncommitted**

  Permette tutte le anomalie: non tiene conto di alcun lock
- **Read committed**

  Evita le letture sporche: richiede lock rilasciati subito dopo
- **Repeatable read**

  Evita quasi tutte le anomalie: applica strict 2PL su singole tuple
- **Serializable**

  Lettura perfetta: applica strict 2PL standard