ORDINAMENTO

→ **ORDER BY AttrDiOrdinamento [asc | desc]**

permette di specificare un ordinamento delle righe del risultato di una query

ASC → ascending order (default)

DESC → descending order

UNION, INTERSECTION, DIFFERENCE

- UNION : SELECT ...

  UNION [ALL] → di default tutte le righe sono uniche, tranne quando viene utilizzato ALL (multi set union)

  SELECT ...

quando due tabelle hanno schemi diversi si assumono i nomi degli attributi del primo operando

- DIFFERENCE: SELECT ... → è possibile esprimere la differenza anche attraverso query annidate

  FROM ...

  EXCEPT

  SELECT ...

  FROM ...

- INTERSECTION SELECT ...

  FROM ...

  INTERSECT ≡

  SELECT ...

  FROM ...

SELECT R.A

FROM R, R1

WHERE R.A = R1.B

QUERY ANNIDATE

Possono essere formulate utilizzando i predicati ANY o ALL con gli operatori (>, <, =, ...)

Attribute op ANY / ALL (Expr)

- IN : la tupla della query esterna è un match se il suo valore è contenuto tra gli elementi ritornati da Expr

Attribute IN (Expr)

ANY, ALL, IN possono essere negati con IN

→ A IN (Expr) ≡ A = ANY (Expr)

A NOT IN (Expr) ≡ A ≠ ALL (Expr)

■ VISIBILITÀ :

Non è possibile fare riferimento a variabili dichiarate all'interno dei blocchi interni. Se il nome di una variabile è omesso, prendiamo la dichiarazione più "vicina". Possiamo fare riferimento a variabili definite:

- nello scope della query in cui è definita (blocchi esterni)
- nello scope di una query annidata, a qualsiasi livello (blocco interno) all'interno di esso.

La query interna viene eseguita una volta per ogni tupla all'interno della query esterna, l'unico modo per evitarlo è creare una vista, che va però a modificare lo schema del database.

EXISTENTIAL QUANTIFICATION

EXISTS (Expr) Il predicato è vero se Expr (usually query annidata) torna almeno una tupla

Advanced SQL

Nested Queries Positions

- *WHERE*: uso standard
- *FROM*: necessita di una nuova fonte di dati, l'alternativa è creare una vista che va però a modificare lo schema del database.
- *SELECT*: è equivalente ad un JOIN. Richiede necessariamente una tupla come risultato.

Provide the name and the income of Jim's children

```
SELECT Name, Income
FROM PEOPLE P, (SELECT Child
                FROM FATHERHOOD
                WHERE Father='Jim') AS
                JIMCHILD
WHERE Name = JIMCHILD.Child
```

Provide the total shipping charges for each customer in the customer table

```
SELECT CUSTOMER.Num,
       (SELECT SUM(ShipCharge)
        FROM ORDERS
        WHERE CUSTOMER.Num=ORDERS.Num)
       AS TotalShipCharge
  FROM CUSTOMER
```

Funzioni di aggregazione
Nella target list possiamo mettere espressioni che calcolano i valori da un insieme di tuple attraverso funzioni aggregate:

- `COUNT`
- `MIN`
- `MAX`

- `AVG`
- `SUM`
  UPDATING OPERATIONS:
- `INSERT`
- `DELETE`
- `UPDATE`

Vincoli di integrità generici

- `CHECK` (Predicate): le condizioni ammesse sono le stesse della clausola `WHERE`. La condizione deve essere sempre verificata affinchè la base di dati sia corretta, in questo modo è possibile specificare tutti i vincoli intrarelazionali.
- `ASSERTION`: vincoli associati allo schema (no tabella / attributo). E' possibile esprimere tutti i vincoli che coinvolgono più tabelle o che richiedono che una tabella abbia cardinalità minima. Possiedono un nome con il quale possono essere eliminate esplicitamente dallo schema con `drop`.
  ```
  create assertion <name>
  check (predicate)
  ```

I vincoli possono essere:

- *immediati*: sono verificati subito dopo ogni modifica
- *differiti*: sono verificati al termine dell'esecuzione di una serie di operazioni

Viste

Sono tabelle "virtuali" il cui contenuto dipende dalle altre tabelle. Vengono definite con un nome e una lista di attributi al risultato dell'esecuzione

```
create view NomeVista [(ListaAttributi)] as SelectSQL
[with [local | cascaded] check option
```

```
create view ADMINEMPLOYEES
   (Name, Surname, Salary) as
   select Name, Surname, Salary
   from EMPLOYEE
   where Dept = 'Administration' and
         Salary > 10
```

Alcuni sistemi considerano una vista aggiornabile solo se è definita su una sola tabella, altri richiedono anche che l'insieme di attributi della vista contenga almeno una chiave primaria della tabella base.

- La clausola `check option` può essere usata solo in questa categoria di viste, specifica che gli aggiornamenti sono ammessi solo sulle righe della vista e che dopo ogni modifica tutte le righe devono continuare ad appartenere alla vista. Permette di aggiornare la vista, solo se la tupla inserita appartiene alla vista (rispetta le condizioni);

- Nel caso in cui una vista è definita in termini di altre viste:
  - `local`: l'aggiornamento della tupla deve essere eseguito solo all'ultimo livello della vista;
  - `cascade`: l'aggiornamento della tupla deve essere propagato a tutti i livelli di definizione.

Querying a View
Le viste possono essere utilizzate per formulare interrogazioni che non sarebbero esprimibili altrimenti es.
Stipendi totali di ogni dipartimento:

```
        create view BudgetStipendi(Dip,TotaleStipendi) as
        select Dipart, sum(Stipendio)
        from Impiegato
        group by Dipart
```

Dipartimento con lo stipendio massimo:

```
select Dip
from BudgetStipendi
where TotaleStipendi = (select max(TotaleStipendi)
                                        from BudgetStipendi)
```

Query scorretta, la sintassi di SQL non permette l'utilizzo di funzioni di aggregazione nidificate:

```
select avg(count(distinct Office))
from EMPLOYEE
group by Dept
```

Possiamo utilizzare una vista per ottenere il numero medio di uffici per dipartimento:

```
create view DEPTOFFICES(NameDept,OffNum) as
        select Dept, count(distinct Office)
        from EMPLOYEE
        group by Dept;

select avg(OffNum)
from DEPTOFFICES
```

Query Ricorsive
Esempio: per ogni persona restituire il suo antenato:
FATHERHOOD(Father, Child)

```
with recursive ANCESTORS(Ancestor,Descendant) as
( select Father, Son
  from FATHERHOOD
  union all
  select Ancestor, Son
  from ANCESTORS, FATHERHOOD
  where Descendant = Father
  )
```

```
select *
from ANCESTORS
```

`with` definisce la vista *ANCESTORS*, costruita ricorsivamente utilizzando *FATHERHOOD*.

Funzioni Scalari
Funzioni a livello di tuple che forniscono un unico valore per tuple.
Temporali:

- `current_date()` restituisce la data attuale
- `extract(yearExpression)` estrae parte di una data da una determinata espressione
  Manipolazione delle stringhe:
- `char_length` ritorna la lunghezza della stringa
- `lower` converte la stringa in minuscolo
- `upper` converte la stringa in maiuscolo
- `substring` restituisce una parte della stringa identificata da indici
  Conversione di dominio:
- `cast` permette di convertire un valore in un dominio nella sua rappresentazione in un altro dominio (es. `Data as char(10)`)
  Funzioni condizionali:
- `coalesce` prende una sequenza di espressioni e restituisce il primo valore non nullo. Può essere utilizzato anche per convertire i valori nulli in valori definiti dal programmatore.
  es 1. Per ogni impiegato tornare un numero di cellulare valido o il suo numero di telefono:
  ```SQL
  select number, coalesce(Mobile, PhoneHome)
  from EMPLOYEE
```
        es 2. per ogni dipartimento il valore NULL è sostituito da 'none':
```SQL
select Name, Surname,
        coalesce(Dept, 'None')
from EMPLOYEE
```

- `nullif` prende come argomento un'espressione e un valore costante; se l'espressione è pari al valore costante la funzione restituisce il valore nullo, altrimenti restituisce il valore dell'espressione.
- `case` permette di specificare strutture condizionali cui risultato dipende dalla valutazione del contenuto delle tabelle.
  es: calcolare le tasse di circolazione dei veicoli immatricolati dopo il 1975, in base ad un tariffario secondo il tipo di veicolo
  Veicolo (*Targa*, Tipo, Anno, KWatt, Lunghezza, NAssi)

```
select Targa,
        (case Tipo
          when 'Auto' then 2.58 * KWatt
          when 'Moto' then (22.00 + 1.00 *KWatt)
          else null
         end) as Tassa
```

```
from Veicolo
where Anno > 1975
```

Database Security

SQL consente di definire per ogni utente specifico a quali risorse possono avere accesso, questo sistema si basa su un concetto di privilegio, esso è caratterizzato da:

1. la risorsa a cui si riferisce
2. l'utente che concede il privilegio
3. l'utente che riceve il privilegio
4. l'azione che viene permessa sulla risorsa
5. se il privilegio può essere trasmesso o meno ad altri utenti
   Quando viene creata una risorsa il sistema concede automaticamente tutti i privilegi al creatore di essa, in più esiste un utente predefinito `_system` che possiede tutti i privilegi.
   I privilegi disponibili sono:

- `insert`: permette di inserire un nuovo oggetto
- `update`: permette di aggiornare il valore di un oggetto
- `delete`: permette di rimuovere oggetti
- `select`: permette di leggere la risorsa e utilizzarla in un interrogazione
- `references`: permette di definire vincoli d'integrità referenziale
- `usage`: permette che venga usata la risorsa
  I privilegi sull'utilizzo di `drop` e `alter` non possono essere concessi.
  I privilegi vengono concessi o revocati tramite le istruzioni `grant` e `revoke`.

Granting Privileges
Sintassi: `grant Privilegi on Risorsa to Utenti [ with grant option ]`
Permette di concedere i *Privilegi* sulla *Risorsa* agli *Utenti*.
`grant option` permette all'utente di propagare i suoi privilegi ad altri utenti.

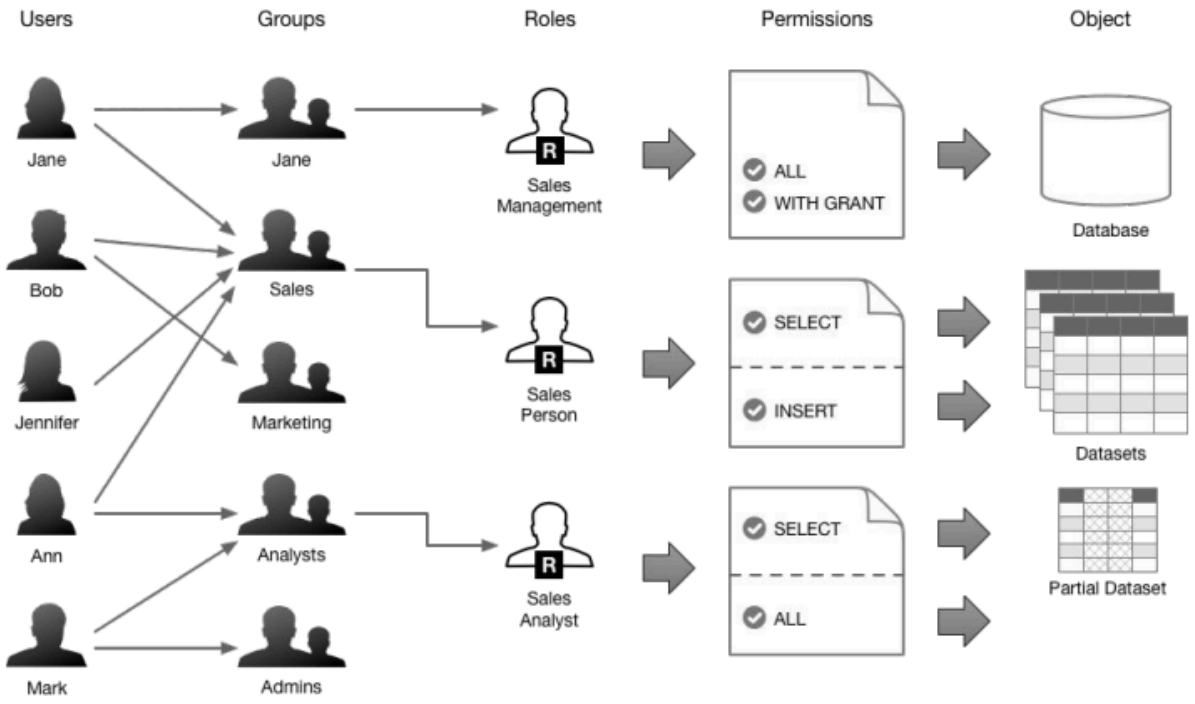
Revoking Privileges
Sintassi: `revoke Privilegi on Risorsa from Utenti [ restrict | cascade ]`
I privilegi possono essere rimossi solo da chi li aveva concessi.

- `restrict` (default), il comando non viene eseguito se la revoca dei privilegi all'utente comporta altre revoche di privilegi.
- `cascade` forza l'esecuzione di `revoke`, tutti i privilegi propagati vengono revocati e tutti gli elementi costruiti sulla base di quei privilegi vengono rimossi.

Role-Based Access Control
RBAC è un modello di controllo dell'accesso basato sui *ruoli*. Un ruolo:

- viene creato con `crete role NomeRuolo`;
- si comporta come un contenitore di privilegi, che gli vengono attribuiti con il comando `grant`, lo stesso comando viene utilizzato anche per concedere agli utenti la possibilità di ricoprire un certo ruolo.
  Per utilizzare i privilegi l'utente deve invocare il comando `set role NomeRuolo`. In ogni momento l'utente dispone dei privilegi che gli sono stati attribuiti direttamente e dei privilegi associati al ruolo

che è stato esplicitamente attivato.



Transazioni

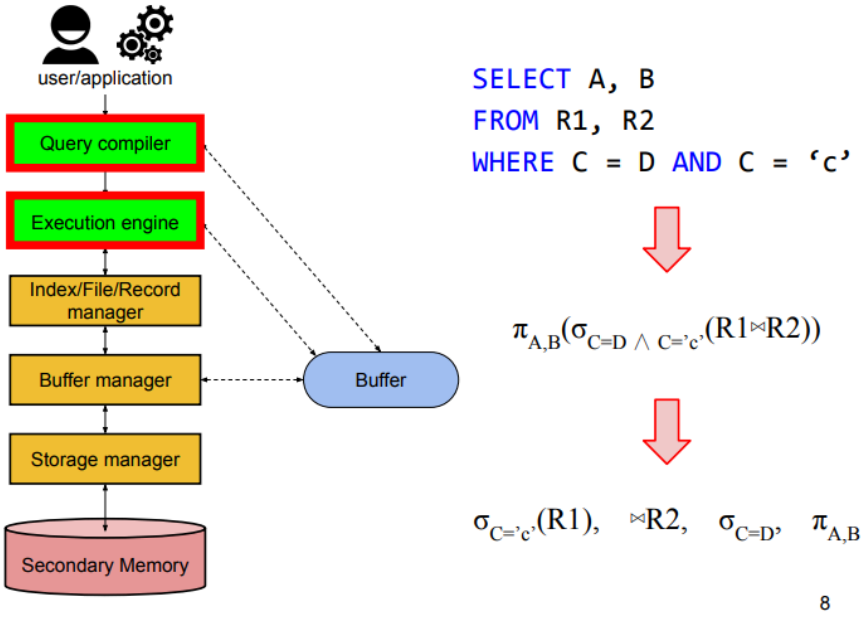
Una *transazione* è una sequenza di operazioni che vengono eseguite come un'unità logica ed ha come obiettivo quello di garantire l'integrità dei dati anche in presenza di errori o guasti.
Rispettano quattro proprietà fondamentali (*ACID*):

- Atomicità (*Atomicity*): una transazione è un'unità *indivisibile* di esecuzione → o vengono resi visibili tutti gli effetti di una transazione, oppure la transazione non deve avere alcun effetto sulla base di dati;
- Consistenza (*Consistency preservation*): l'esecuzione della transazione non deve violare i vincoli di integrità definiti sulla base di dati. Quando il sistema rileva una violazione interviene per annullare la transazione o per correggere la violazione del vincolo.
  - vincolo di integrità di tipo *immediato*: la verifica può essere fatta nel corso della transazione.
  - vincolo di integrità di tipo *differito*: la verifica deve essere effettuata alla conclusione della transazione. In caso di violazione gli effetti della transazione vengono annullati.
- Isolamento (*Isolation*): l'esecuzione di una transazione deve essere indipendente dalla contemporanea esecuzione di altre transazioni;
- Persistenza (*Durability*): l'effetto di una transazione che ha eseguito il commit correttamente deve essere persistente e non deve essere perso

DBMS Architecture

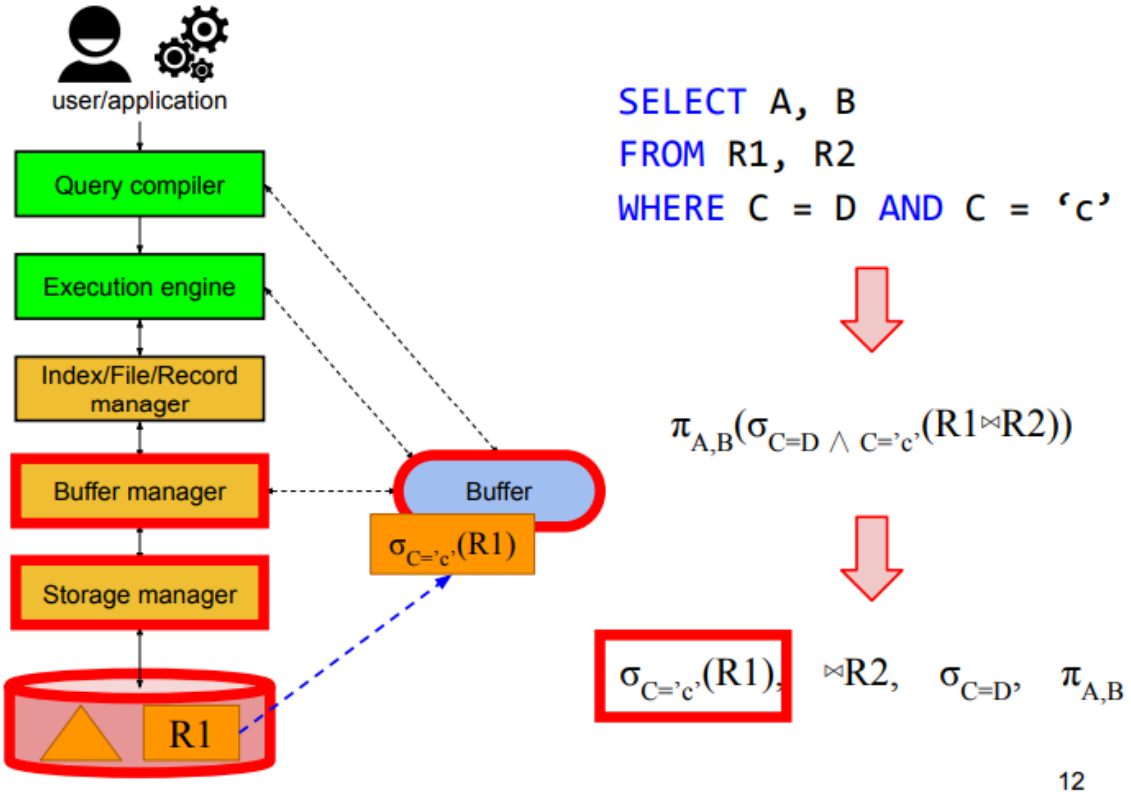
Un database management system (DBMS) è un software per creare, gestire e manipolare grandi quantità di dati in modo efficiente e persistente. L'architettura di un DBMS è composta da:

- **Query Processor**:
  è il componente che interpreta ed esegue la query SQL. Si compone di:
  - *parsing*: verifica la sintassi della query e costruisce un albero sintattico
  - *preprocessing*: verifica la semantica (es. se le tabelle e colonne esistono) e traduce la query in un albero di operazioni algebriche
  - *ottimizzazione*: trova la sequenza più efficiente di operazioni per eseguire la query (es. l'ordine delle tabelle di un join)
  - **Execution Engine**: esegue il piano ottimizzato interagendo con altri componenti, come la memoria e i moduli di gestione dei file.



- **Resource Manager**:
  Si occupa di gestire i file a livello fisico. E' responsabile della persistenza dei dati e dell'efficienza dell'accesso. Include:
  - *Index/File/Record Manager*: conosce la struttura dei dati e utilizza indici per velocizzare le ricerche;
  - *Buffer Manager*: carica i blocchi di dati dalla memoria secondaria alla memoria principale;

- *Storage Manager*: gestisce i file su disco, localizzandoli e leggendo i blocchi richiesti.



- **Transaction Manager**:
  si occupa delle transazioni, unità logiche di lavoro nel database spesso costituite da più query SQL. Esempio: transazione di trasferimento bancario:

```
START TRANSACTION;
UPDATE BANKACCOUNT SET Balance = Balance - 500 WHERE AccountNumber = 42177;
UPDATE BANKACCOUNT SET Balance = Balance + 500 WHERE AccountNumber = 12202;
COMMIT;
```

Concorrenza e Deadlock
In un sistema di database, più transazioni possono essere eseguite contemporaneamente, ma potremmo incontrare problematiche come:

- interferenza tra transazioni (es. una transazione modifica un dato mentre un'altra lo legge).
- inconsistenza: quando l'accesso simultaneo ai dati causa risultati non coerenti.
  Il **Concurrency-Control Manager** utilizza tecniche per garantire che le transazioni parallele non interferiscano negativamente tra loro:

1. *Locking* (blocco):
   - un lock è un meccanismo che impedisce ad altre transazioni di accedere a un dato finchè non viene rilasciato
   - due tipi principali:
     - *shared lock* (lettura): più transazioni possono leggere lo stesso dato
     - *exclusive lock* (scrittura): solo una transazione può modificare il dato
2. *schedulazione*: l'ordine di esecuzione delle transazioni viene determinato per garantire che i risultati siano equivalenti a una loro esecuzione sequenziale (serializzabilità).
   Un *deadlock* si verifica quando due o più transazioni sono bloccate in attesa di risorse detenute dall'altra. Esempio:

- Transazione T1 blocca il record X e vuole accedere al record Y.
- Transazione T2 blocca il record Y e vuole accedere al record X. Nessuna delle due può procedere.
  Il DBMS utilizza diverse strategie per risolvere questo tipo di conflitti:

1. **timeout**: una transazione viene terminata automaticamente se resta bloccata troppo a lungo.
2. **detection e rollback**
   - il sistema rileva i deadlock costruendo un grafo di attesa.
   - identificato il deadlock, una transazione viene abortita (rollback) per liberare le risorse.

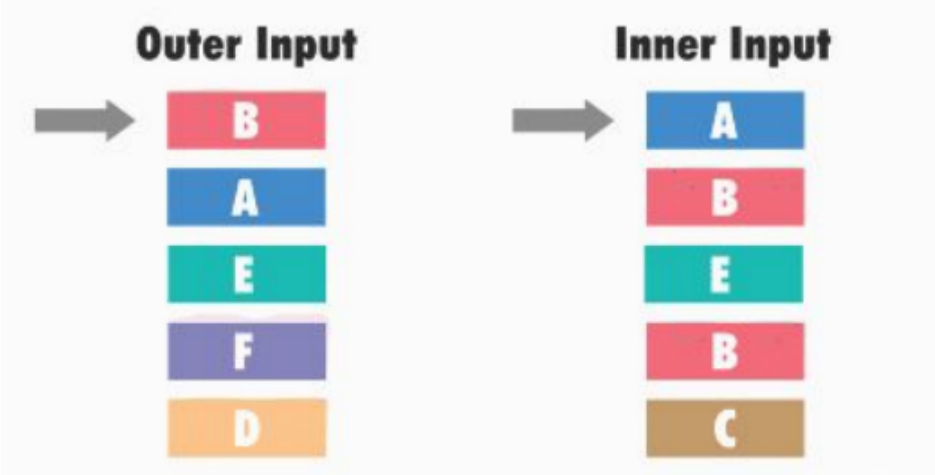
Logging and Recovery
Logging: Il *Log Manager* registra ogni modifica effettuata nel database, creando un registro che può essere utilizzato per recuperare il sistema in caso di guasto.
Recovery: Quando il sistema subisce un guasto. il *Recovery Manager* utilizza il log per riportare il database a uno stato consistente.

Tecniche di JOIN

Il join è un'operazione computazionalmente costosa, quindi il DBMS usa diverse tecniche per ottimizzarla.

Nested-loop JOIN

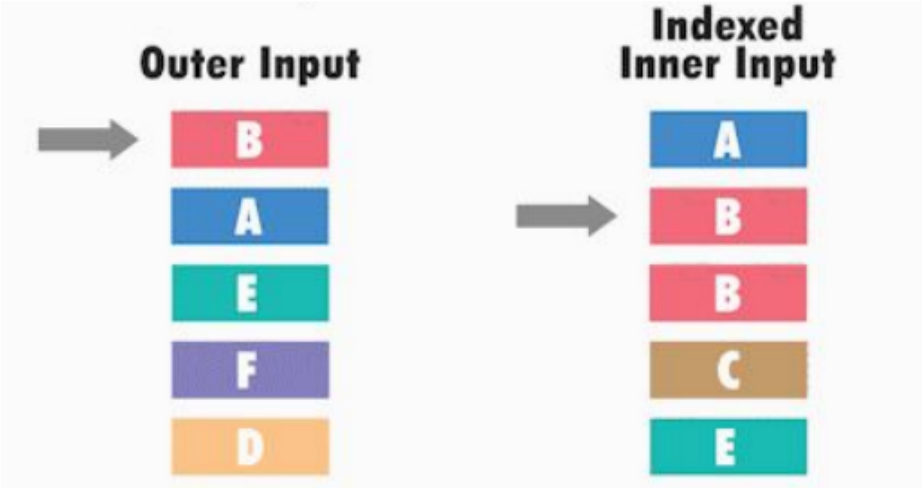
- più semplice, ma meno efficiente.
- per ogni riga della tabella R (outer), vengono scansionate tutte le righe della tabella S (inner)
  PRO: non richiede precondizioni. CONTRO: estremamente lento per tabelle grandi.
  (es. R ha righe 1000 righe e S ne ha 500, vengono effettuate 1000 * 500 = 500000 confronti)



Single-loop JOIN

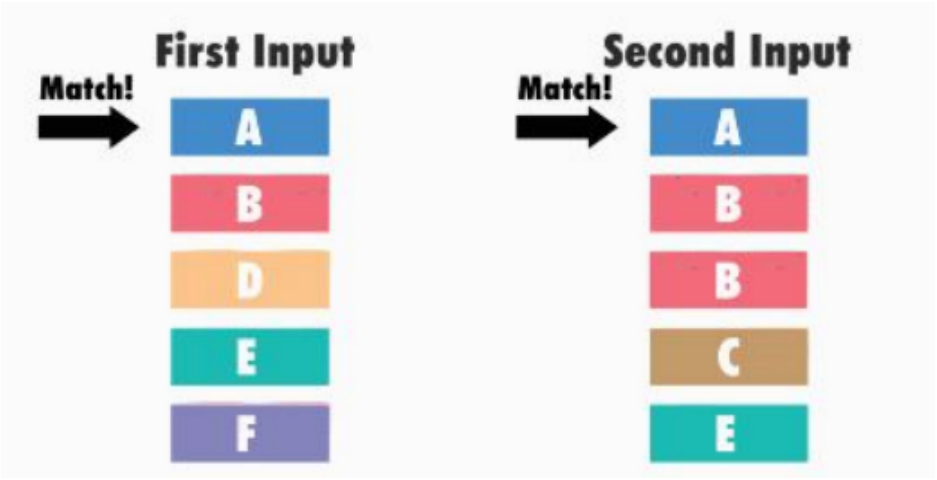
- usa un indice (o struttura hash) su una delle due tabelle.
- per ogni riga di una tabella (es. S), l'indice viene utilizzato per cercare direttamente le righe corrispondenti nell'altra tabella (es. R).
  Se esiste un indice sull'attributo di join di R, l'accesso a ogni riga è diretto, riducendo il numero totale di confronti.
  PRO: più efficiente rispetto al nested-loop se gli indici esistono.

CONTRO: richiede indici preesistenti.



Sort-merge JOIN

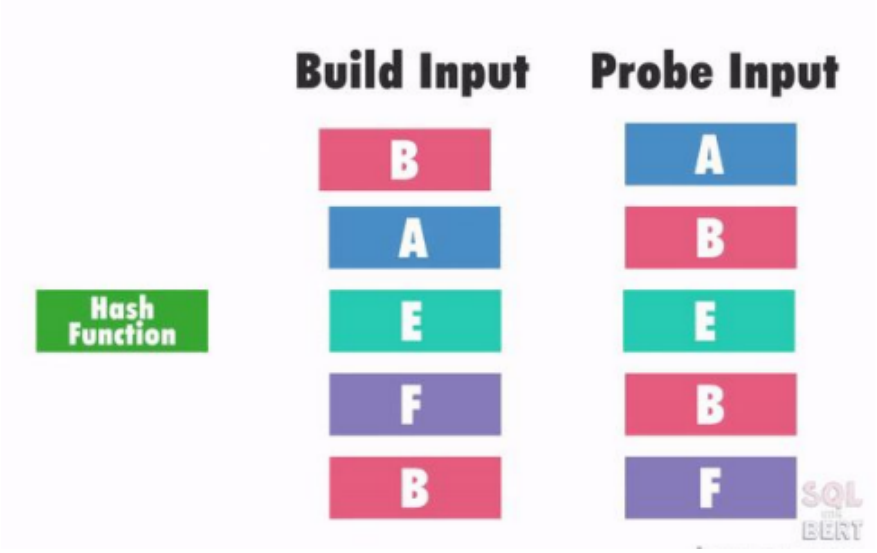
- ordina entrambe le tabelle sugli attributi di JOIN.
- scansiona le tabelle ordinate una sola volta, abbinando le righe con valori uguali.
  PRO: ottimo per tabelle grandi, specialmente quando devono essere ordinate.
  CONTRO: richiede una fase di ordinamento iniziale se le tabelle non sono già ordinate.



Hash-based JOIN

- usa una funzione di hashing per suddividere entrambe le tabelle in *bucket*.
- le righe nei bucket corrispondenti vengono abbinate.
  Esempio: con una funzione hash che divide una tabella in 10 bucket, ogni bucket viene processato separatamente, riducendo il numero di confronti.
  PRO: efficiente per tabelle grandi

CONTRO: richiede memoria sufficiente per gestire i bucket.



Transactions

Un DBMS è un ambiente multiutente in cui vari programmi interagiscono con i dati. Include componenti quali:

- *transaction manager*
- *concurrency control*
- *logging and recovery*
  e altri moduli per gestire memoria, esecuzione di query e sicurezza dei dati.

Un programma è una sequenza di:
read(X): legge un elemento del database chiamato X;
write(X): scrive un valore nell'elemento del database.

Una **transazione** rappresenta l'esecuzione del programma di un utente come unità indivisibile (insieme di operazioni di lettura e scrittura).
Problemi principali:

- esecuzione concorrente: più transazioni possono essere eseguite simultaneamente
  motivazioni:
  - gli accessi al disco sono lenti; la CPU non dovrebbe rimanere inattiva.
  - Per migliorare le prestazioni, più transazioni possono essere eseguite in modo concorrente, mantenendo la CPU occupata.
    - **Problema**: l'utente non deve percepire l'interleaving delle transazioni, cioè l'esecuzione simultanea di più operazioni.
- **Recupero da crash**: il database deve rimanere consistente anche in caso di errori o guasti.
  - **Problema**: il DBMS deve garantire che altre transazioni non siano influenzate dall'errore e deve lasciare il database in uno stato consistente.

Proprietà ACID

Per garantire l'esecuzione sicura e simultanea delle operazioni, ogni transazione deve rispettare le proprietà ACID:

Atomicità

- Una transazione deve essere completata integralmente o non avere alcun effetto sul database.
- **Gestione dell'abort**:
  - Cause di abort:
    1. Anomalie interne all'esecuzione (bloccate dal DBMS).
    2. Condizioni di eccezione rilevate dalla transazione (autosospensione).
    3. Crash di sistema (es. errori hardware, software o di rete).
  - In caso di abort, il DBMS:
    - Utilizza il file di log per annullare la transazione (rollback) e ripristinare lo stato consistente precedente.

Consistenza

- La transazione inizia e termina rispettando tutti i vincoli di integrità del database.

Isolamento

- Ogni transazione deve essere eseguita come se fosse l'unica attiva, indipendentemente dall'interleaving con altre transazioni.

Durabilità

- Una volta che una transazione è confermata (commit), le modifiche apportate diventano permanenti, anche in caso di crash.
  Una transazione:
- parte sempre in uno stato consistente dove tutti i vincoli referenziali sono soddisfatti;
- potrebbe passare attraverso uno stato intermedio di inconsistenza;
- termina sempre in uno stato consistente dove tutti i vincoli referenziali sono soddisfatti.

Schedulazione delle Transazioni

- **Definizione:** Una schedule rappresenta la sequenza cronologica di operazioni (read, write, commit, abort) eseguite da più transazioni.

Esempio

Dati:

- **T1**: `read1(A) → write1(A) → read1(C) → write1(C)`
- **T2**: `read2(B) → write2(B)`
  Schedule:
  `read1(A) → write1(A) → read2(B) → write2(B) → read1(C) → write1(C)`

Tipi di schedule:

1. **Completa:** Include commit o abort per ogni transazione.
2. **Seriale:** Tutte le operazioni di una transazione vengono eseguite consecutivamente (una transazione attiva alla volta).
3. **Serializzabile:** Non è necessariamente seriale, ma produce lo stesso risultato di una schedule seriale.

Anomalie nell'Esecuzione Interleaved

Quando più transazioni sono eseguite simultaneamente, possono verificarsi anomalie che lasciano il database in uno stato inconsistente.

Tipi di conflitti:

1. **Write-Read Conflict (Dirty Read):**
   - **T1** aggiorna `A` ma non esegue il commit.
   - **T2** legge `A` e utilizza un dato potenzialmente errato.

- Se **T1** fa rollback, il dato letto da **T2** è invalido.
  - Given the following two transactions T_1 and T_2 and the schedule S:

    T_1: A := A + 100, B := B - 100

    T_2: A := A * 1.06, B := B * 1.06

  - **Please note**: T_1 writes a value in A that could make the DB inconsistent, and this value is read by T_2
  - **Problem**: A is written by T_1 and read by T_2 before T_1 commits

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
	read(B)
	write(B)
	commit
read(B)	
write(B)	
commit	

2. **Read-Write Conflict (Unrepeatable Read):**
   - **T1** legge `A = 500`.
   - **T2** aggiorna `A` a 600 e conferma.
   - **T1** legge di nuovo `A` e trova 600, ma le letture di **T1** non sono coerenti.
     - Given the following two transactions T_1 and T_2 and the schedule S:

       T_1: A := A + 1

       T_2: A := A - 1

     - **Please note**: if T_1 repeats the reading of A, it will get a different value
     - **Problem**: A is written by T_2 before T_1 commits

T_1	T_2
read(A)	
	read(A)
	write(A)
	commit
write(A)	
commit	

3. **Write-Write Conflict (Lost Update):**
   - **T1** aggiorna `A` a 600.

- **T2** sovrascrive `A` a 450, perdendo l'aggiornamento di **T1**.
  - Given the following two transactions T_1 and T_2 and the schedule S:

    T_1: A := 1000, B := 1000

    T_2: A := 2000, B := 2000

  - **Please note**: the values of A and B are equal at the end of each of the two transactions
  - **Problem**: A and B have different values at the end of the schedule

T_1	T_2
write(A)	
	write(A)
	write(B)
	commit
write(B)	
commit	

4. **Phantom Anomalies:**

- Una transazione legge un insieme di dati (es. una tabella) e successivamente rileva che l'insieme è cambiato (es. inserimento/cancellazione di righe da un'altra transazione).

Aborted Transactions
In linea di principio, tutte le azioni di una transazione aborted devono essere cancellate. Tuttavia non è sempre possibile. Esempio:

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
	read(B)
	write(B)
	commit
read(B)	
write(B)	
abort	

problema: T2 legge il valore di A che non avrebbe dovuto leggere, a causa di un write-read conflict
soluzione non soddisfacente: T2 dovrebbe anche essere abortito, ma violerebbe la proprietà di durabilità delle transazioni.

Transactions Parameters

- **Access Mode** imposta l'autorizzazione per modificare le tabelle utilizzate nella transazione:
  - **read only**: permette soltanto la lettura del DB, il tentativo di modificare il DB causa un errore
  - **read write**: consente le operazioni di lettura e scrittura
- **Statement Mode**: specifica le azioni da eseguire al termine di una transazione

- **Isolation Level**: specifica come gestire le transazione che modificano il DB:
  - **READ UNCOMMITTED**: la transazione richiede blocchi per scrivere oggetti ma non blocchi per la lettura. Consente *dirty read*.
  - **READ COMMITTED**: la transazione richiede blocchi per la scrittura e blocchi condivisi per la lettura. Previene *dirty read*.
  - **REPEATABLE READS**: blocca letture e scritture fino al commit.
  - **SERIALIZABLE**: blocca transazioni che potrebbero causare conflitti.

Level	Dirty read (Write-Read)	Unrepeatable read (Read-Write)	Lost update (Write-Write)	Phantom
READ UNCOMMITTED	may occur	may occur	may occur	may occur
READ COMMITTED	don't occur	may occur	may occur	may occur
REPEATABLE READS	don't occur	don't occur	don't occur	may occur
SERIALIZABLE	don't occur	don't occur	don't occur	don't occur

Concurrency Control Approaches

L'interleaving è necessario e preferibile per migliorare le performance, d'altronde non tutte le schedule sono possibili, alcune azioni devono essere riavvolte (roll back).
Diversi modi di controllare la concorrenza:

- **RESTRICTIVE**: (conflict-serializability) ogni transazione esegue due fasi:
  1. *acquisizione*: richiede blocchi
  2. *rilascio*: libera i blocchi dopo il commit

     Esempio:
     T1 acquisisce un blocco esclusivo su A e lo modifica.
     T2 deve aspettare il rilascio del blocco di A.

Strict Two-phase Locking (Strict 2PL):

- se una transazione vuole leggere/scrivere un oggetto, deve richiedere un accesso esclusivo.
- dopo aver rilasciato un lock, la transazione non può chiederne altre
- quando viene eseguito il commit la transazione rilascia tutti gli accessi esclusivi
- Strict 2PL garantisce la serializzabilità, in particolare:
  - se le transazioni accedono a oggetti diversi, possono essere interleaved
  - altrimenti, se almeno due transazioni voglio l'accesso allo stesso oggetto e almeno una delle due vuole modificarlo, le due transazioni dovranno essere eseguite in serie
    il **Lock Manager** tiene traccia, per ogni oggetto del DB, dell'accesso in una *lock table*.

- **OPTIMISTIC**: esegue tutte le transazioni concorrenti e verifica la presenza di conflitti prima del commit
- **TIMESTAMPING**: assegna timestamps alle transazioni e compara tutti i valori per determinare l'ordine delle operazioni.

Deadlock Prevention

Una concorrenza locking-based potrebbe causare *deadlocks*. Solitamente i DBMS assegnano una priorità alle transazioni in base al tempo di inizio.
Se T1 richiede un lock posseduto da T2, il lock manager può:

- **wait-die**: se T1 è la transazione antecedente, T1 aspetta. Altrimenti T1 viene terminata e ripresa successivamente, con un random delay ma con lo stesso timestamp
- **wound-wait**: se T1 è la transazione antecedente, T2 viene terminata e ripresa successivamente con random delay, ma stesso timestamp. Altrimenti, T1 aspetta.
  Se i deadlock sono rari, il DBMS lascia che si verifichino e li risolve invece di adottare le policy per evitarli. Due approcci più comuni:
- il lock manager mantiene una struttura chiamata **waits-for graph**, che utilizza per identificare il cicli deadlock. Il grafo è periodicamente analizzato e i cicli deadlock sono risolti abortendo alcune transazioni.
- se una transazione aspetta per un periodo più lungo del timeout assegnato, il lock manager assume che la transazione è deadlocked e la abortisce.

Optimistic Concurrency Control

Il protocollo basato sul locking adotta un approccio pessimistico per prevenire i conflitti.
L'approccio ottimistico assume che le transazioni non vadano in conflitto (o che lo facciano raramente).
Validation è un approccio ottimistico dove le transazioni possono accedere ai dati senza lock, e al momento opportuno, controlla che le transazioni si siano comportate in modo seriale.
Le transazioni sono eseguite in tre fasi:

- **read**: la transazione è eseguita leggendo il dato dal DBMS e scrivendo in un area privata
- **validation**: prima del commit, il DBMS controlla che non ci siano stati conflitti. In tal caso, la transazione viene abortita e restartata automaticamente.
- **write**: se la fase di validation viene conclusa con successo, il dato scritto nell'area privata viene copiato nel DBMS.

Timestamping concurrency control

Timestamping è un altro approccio ottimistico che assegna per ogni transazione il timestamp TS del suo start time. Per ogni operazione a1 eseguita da T1:

- se l'operazione a1 è in conflitto con l'operazione a2 eseguita da T2 e TS1 < TS2, allora a1 deve essere eseguita prima di a2.
- se un'operazione eseguita da T viola questo ordine, la transazione T è abortita e ripresa con un TS maggiore.

Crash Recovery

Il **Logging and recovery manager** del DBMS deve assicurare:

- *atomicità*: operazioni eseguite da transazioni non-committed sono rolled back.
- *persistenza*: operazioni eseguite da transazioni committed devono persistere ad un crash di sistema

ARIES Recovery Algorithm

Advanced Recovery and Integral Extraction System è un algoritmo di ripristino eseguito dal Logging and Recovery Manager sugli arresti anomali di sistema. Tre fasi:

- **fase di analisi**: identifica le pagine sporche del buffer pool (cioè, le modifiche non ancora scritte sul disco) e le operazioni attive nel momento del crash
- **fase di redo**: partendo da un dato checkpoint nel log file, ripete tutte le operazioni e riporta il DB nello stato in cui si trovava al momento del crash
- **fase di undo**: cancella le operazioni delle transazioni che erano attive al momento del crash, ma che non erano committed, in ordine inverso.
  Principi:
- **write-ahead logging**: qualunque cambiamento ad un oggetto del DB deve prima essere registrato nel log file. Successivamente, il log file deve essere riportato sulla memoria secondaria. Infine, le pagine modificate possono essere aggiornate.
- **repeating history during redo**: Al riavvio, dopo un arresto, il sistema viene riportato allo stato in cui si trovava prima dell'anomalia. Le operazioni delle transazioni ancora attive durante il crash vengono cancellate.
- **logging changes during redo**: i cambiamenti fatti al DB durante l'annullamento delle transazioni sono registrati per assicurare che quell'azione non venga ripetuta nel caso di un altro arresto anomalo.

Log File

Il log file tiene traccia di tutte le azioni eseguite dal DBMS. E' fisicamente organizzato in registrazioni memorizzate in uno storage stabile, che dovrebbe resistere ad incidenti / guasti dell'hardware. I log records sono ordinati sequenzialmente con un id unico, *Log Sequence Number* (LSN).
La parte più recente del log file, detta **log tail**, è conservata in **log buffers** e salvata periodicamente in storage stabili. Il log file e i dati vengono scritti su disco con gli stessi meccanismi.

Logging Data Structures

La **dirty page table** tiene i record di tutti gli ID delle pagine che sono state modificate e non ancora scritte in uno storage stabile, e della LSN dell'ultimo log entry che l'ha causato.

Log File Record

<LSN, Transaction ID, Page ID, Redo, Undo, Previous LSN>

- i campi Transaction ID e Page ID identificano la transazione e la pagina
- i campi Redo e Undo tengono le informazioni sulle modifiche che il log record salva e sul come annullarle
- il campo Previous LSN è una reference al precedente log record creato per la stessa transazione. Permette il roll back di transazioni abortite.

Creazione di Log File Record

Un record è scritto nel Log File per ognuno dei seguenti eventi:

- **page update**: un record deve essere scritto in memoria stabile prima di modificare effettivamente i dati della pagina. Mantiene sia il vecchio che il nuovo valore della pagina per rendere possibili le

operazioni di undo e redo.

- **commit**: un record traccia che una transazione è stata completata con successo e il log tail viene scritto nello stable storage.
- **abort**: un record traccia una transazione abortita, e la transazione undo viene ripresa
- **end**: dopo un commit/abort, sono necessarie alcune operazioni di finalizzazione al fine della quale viene scritta una registrazione finale
- **undoing operation**: durante un recovery o durante l'undoing delle operazioni, viene un scritto un tipo speciale di file record, il Compensation Log Record (CLR). Un record CLR non viene mai ripristinato e traccia che un operazione e stata già annullata

Checkpoint

Un checkpoint è uno snapshot dello stato del DB. I checkpoint riducono il tempo di ripristini. Invece di dover eseguire l'intero log file, è sufficiente eseguire all'indietro fino ad un checkpoint. ARIES crea checkpoints in tre step:

- **begin-checkpoint**: il record viene scritto nel log file
- **end-checkpoint**: il record, contenente *dirty page table* e *transaction table*, viene scritto nel log file
- alla fine dell'end-checkpoint il record viene scritto nello stable storage, un record, **Master Record**, contenente LSN del *begin-checkpoint* viene scritto in una parte conosciuta dello stable storage Questo tipo di checkpoint viene detto **fuzzy checkpoint** e non è costoso in termini di performance. Non interrompe le normali operazioni del DBMS e non richiede la scrittura delle pagine del buffer pool.

ARIES Crash Recovery

- **analysis phase**:
  - determina la locazione del log file da dove comincia la fase di redo, ovvero l'inizio dell'ultimo checkpoint
  - determina quale pagine del buffer pool contiene il dato modificato che non era ancora stato scritto al momento dell'anomalia.
  - identifica le transazioni che erano in corso al momento dell'anomalia
- **redo phase**: dalla *dirty page table*, ARIES identifica il minimo LSN di una dirty page. Da qui, ripete le azioni fino all'anomalia, nel caso in cui non erano già state rese persistenti
- **undo phase**: i cambiamenti di una transazione uncommitted devono essere annullate in modo da ripristinare il DB in uno stato consistente.