Basi di Dati

modulo Tecnologie

Alessio Gjergji

Anno accademico 2022 - 2023

Indice

Capitolo 1

Transazioni

1.1 Tecnologie per le basi di dati

Un DBMS (*Database Management System*) o sistema di gestione dei basi di dati, è un sistema software in grado di gestire collezioni di dati che siano, grandi, condivise e persistenti, assicurando la loro affidabilità e privatezza.

Un DBMS in quanto sistema software deve essere **efficiente** ed **efficace**, caratteristiche dipendenti dalla bontà di progettazione della base di dati. Deve inoltre estendere la funzionalità del file system, garantendo la gestione di collezioni di dati presenti in memoria secondaria. Una base di dati è quindi di una collezione di dati gestita da un DBMS.

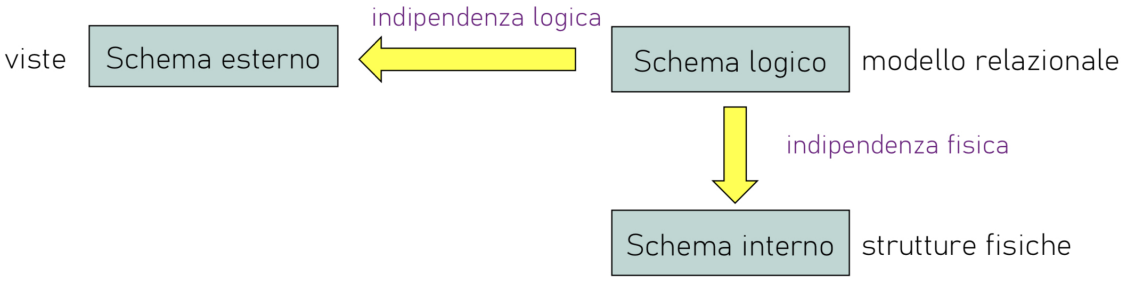


Figura 1.1.1: Tecnologia dei DBMS

Un DBMS basato sul modello relazionale è nella maggior parte dei casi anche un **sistema transazionale**: fornisce quindi un meccanismo per la definizione ed esecuzione delle transazioni.

1.1.1 Transazione

La **transazione** è un'unità di lavoro svolta da un programma applicativo per la quale si vogliono garantire proprietà di correttezza, robustezza e isolamento.

La principale caratteristica di una transazione è che solamente se va a buon fine ha effetto sulla base di dati, se non va a buon fine le operazioni vengono abortite, non sono quindi ammesse esecuzioni parziali.

La transazione va a buon fine all'esecuzione di un `commit work`. Se Non ha alcun effetto sulla base di dati viene eseguito un `rollback work`.

La transazione è ben formata se:

- inizia con un `begin transaction`;
- termina con un `end transaction`;
- a sua esecuzione comporta il raggiungimento di un `commit` o di un `rollback work` senza che vengano eseguiti altri accessi alla base di dati successivamente.

Esempio

```
begin transaction;
    update CONTO set saldo = saldo - 1200
        where filiale = '005' and numero = 15;
    update CONTO set saldo = saldo + 1200
        where filiale = '005' and numero = 205;
    commit work;
end transaction;
```

Proprietà delle transazioni

Una transazione ha quattro proprietà:

- Atomicità (*Atomicity*): una transazione è una esecuzione **indivisibile**. O viene eseguita completamente o non viene eseguita affatto. Se una transazione viene interrotta prima del commit, il lavoro fin qui eseguito dalla transazione deve essere disfatto ripristinando la situazione in cui si trovata la base di dati prima dell'inizio della transazione. Se viene interrotta dopo l'esecuzione del commit, il sistema deve assicurare che la transazione abbia effetto sulla base di dati.
- Consistenza (*Consistency*): l'esecuzione di una transazione non deve violare i **vincoli d'integrità**:
  - Verifica immediata: fatta durante il la transazione, viene abortita solo l'ultima operazione e il sistema restituisce all'applicazione una segnalazione d'errore, l'applicazione può quindi reagire alla violazione.

  - Verifica differita: al commit se un vincolo di integrità viene violato, la transazione viene abortita senza possibilità da parte dell'applicazione di reagire alla violazione.

- Isolamento (*Isolation*): l'esecuzione di una transazione deve essere **indipendente** dalla contemporanea esecuzione di altre transazioni. Il rollback di una transazione non deve creare rollback a catena di altre transazioni che si trovano in esecuzione contemporaneamente. Il sistema, inoltre, deve regolare l'esecuzione concorrente con meccanismi di controllo dell'accesso alle risorse.
- Persistenza (*Durability*): l'effetto di una transazione che ha eseguito il commit non deve andar perso. Il sistema deve essere in grado, in caso di guasto, di garantire gli effetti delle transazioni che al momento del guasto avevano già eseguito un commit.

Un DBMS che gestisce transazioni dovrebbe garantire per ogni transazione che esegue tutte queste proprietà.

1.1.2 Architettura di un DBMS

L'architettura mostra i **moduli principali** che possiamo individuare nei DBMS attuali, considerando le diverse funzionalità che il DBMS svolge durante l'esecuzione dell transazione.

Per ogni modulo dell'architettura presentiamo le **funzionalità** che esso svolge e alcune tecniche che applica.

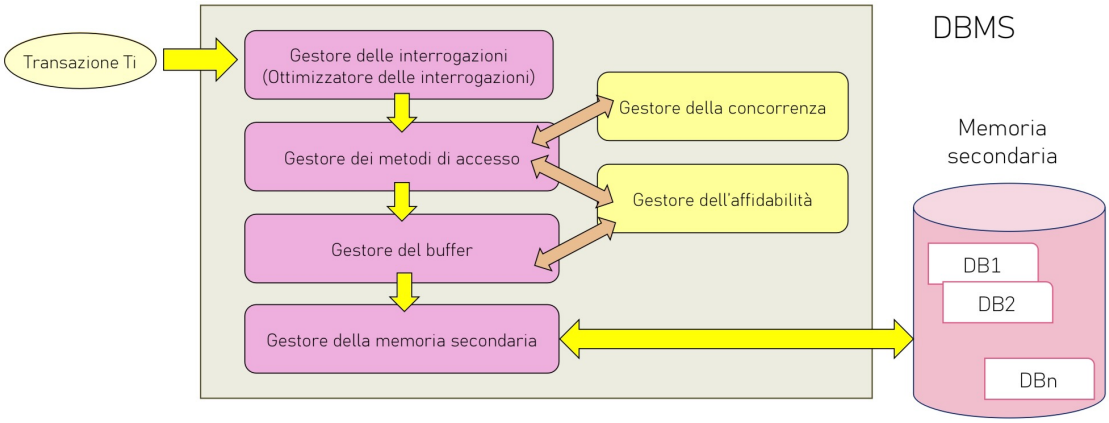


Figura 1.1.2: Architettura di riferimento di un DBMS

Sequenza

- Il gestore delle interrogazioni prende in input una transazione, quindi l'espressione DML e calcolerà un piano di esecuzione.
- Il piano di esecuzione sarà l'input per il gestore dei metodi d'accesso, che eseguirà operazioni di:
  - Richieste d'accesso verso il gestore dell'esecuzione corrente.

  - Riceverà permessi di accesso dal gestore dell'esecuzione corrente

  In output fornirà richieste di pagine (*o blocchi della memoria secondaria*) contenente dati e/o indici.

- Le richieste di blocchi contenenti dati/indici sarà l'input per il gestore dell'affidabilità (*o dei guasti*) che manderà in output richieste di blocchi contenenti dati/indici o blocchi del `LOG`.
- Le richieste di blocchi contenenti dati/indici o log andranno in input al gestore del buffer, in comunicazione con la memoria secondaria.

I moduli che garantiscono le proprietà delle transazioni sono:

- Gestore dei metodi d'accesso: consistenza.
- Gestore dell'esecuzione concorrente: atomicità e isolamento.
- Gestore dell'affidabilità: atomicità e persistenza.

1.2 Architettura di un DBMS

Le basi di dati gestite da un DBMS risiedono in memoria secondaria, in quanto sono:

- Grandi: non possono essere contenute in memoria centrale
- Persistenti: hanno un tempo di vita che non è limitato all'esecuzione dei programmi che le utilizzano.

La memoria secondaria non è direttamente utilizzabile dai programmi, l'organizzazione dei dati è strutturata in pagine, le uniche operazioni possibili sono la lettura e la scrittura di un intera pagina e il costo di tali operazioni è in ordini di grandezza maggiore per accedere ai dati in memoria centrale.

1.2.1 Gestore del Buffer

L'interazione tra memoria secondaria e memoria centrale avviene attraverso il trasferimento di pagine della memoria secondaria in una zona appositamente dedicata della memoria centrale detta **buffer**. Tale zona è condivisa dalle applicazioni e quando uno stesso dato viene utilizzato più volte in tempi ravvicinati il buffer evita l'accesso alla memoria secondaria, poiché dispendiosa. La gestione ottimale del buffer è strategica per ottenere buone prestazioni nell'accesso ai dati.

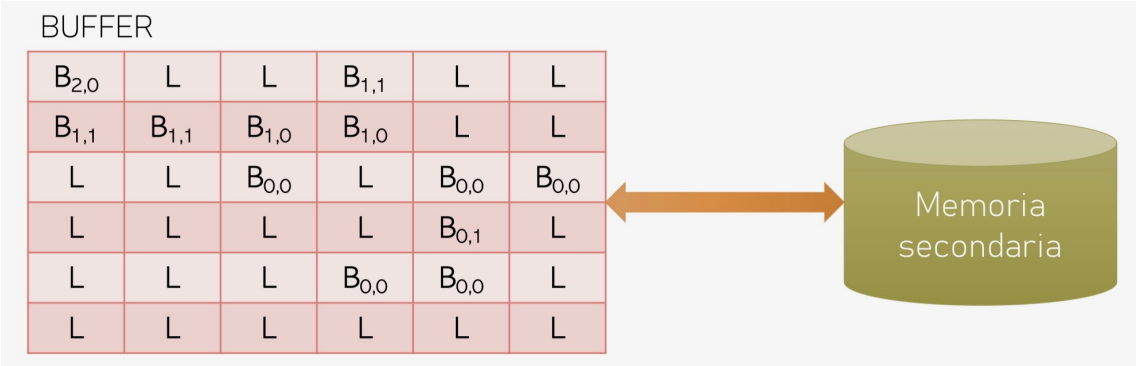


Figura 1.2.1: $\texttt{B}_{i,j}$ indica che nella pagina del buffer è caricato il blocco `B`, inoltre `i` indica che il blocco è attualmente utilizzato da `i` transazioni, mentre `j` è 1 se il blocco è stato modificato, 0 altrimenti. `L` indica pagina libera.

Il buffer è organizzato in pagine e una pagina ha le dimensioni di un blocco della memoria secondaria. Il gestore del buffer si occupa del caricamento/salvataggio delle pagine in memoria secondaria a fronte di richieste di lettura/scrittura.

Politica del gestore dei buffer

- Lettura di un blocco: se il blocco è presente in una pagina del buffer allora non si esegue una lettura su memoria secondaria e si restituisce un puntatore alla pagina del buffer. Altrimenti si cerca una pagina libera e si carica il blocco nella pagina, restituendo il puntatore alla pagina stessa.
- Scrittura di un blocco: in caso di richiesta di scrittura di un blocco precedentemente caricato in una pagina del buffer, il gestore del buffer può decidere di differire la scrittura su memoria secondaria in un secondo momento.

In entrambi i casi l'obiettivo è quello di aumentare la velocità di accesso ai dati.

Tale comportamento del gestore dei buffer si basa sul **principio di località**, i dati referenziati di recente hanno maggiore probabilità di essere referenziati nuovamente in futuro. Inoltre, una nota **legge empirica** dice che il 20% dei dati e acceduto dall'80% delle applicazioni.

Tutto ciò rende conveniente dilazionare la scrittura su memoria secondaria delle pagine del buffer.

Gestione delle pagine

Per ogni pagina del buffer si memorizza il **blocco B** contenuto indicando il file e l'offset del blocco. Per ogni pagina del buffer si memorizza un insieme di **variabili di stato** tra cui si trovano sicuramente:

- Un contatore `i` per indicare il numero di transazioni che utilizzano le pagine.
- Un bit di stato `j` per indicare se la pagina è stata modificata o meno.

1.2.2 Primitive per la gestione del buffer

La primitiva `fix` viene usata dalle transazioni per richiedere l'accesso ad un blocco richiesto.

Passi della primitiva:

- Il blocco richiesto viene cercato nelle pagine del buffer, in caso sia presente, si restituisce un puntatore alla pagina contenente il blocco
- Altrimenti, viene scelta una pagina libera (*in base a diversi criteri*) e si carica il blocco aggiornando file e numero di blocco corrispondenti.
- Se non esistono pagine libere, il gestore del buffer può adottare due politiche:
  - Steal: ruba una pagina ad un'altra transazione (*primitiva* `flush`).
  - No Steal: sospende la transazione inserendola in una cosa d'attesa.

Quando una transazione accede ad una pagina per la prima volta il contatore si incrementa.

La primitiva `setDirty` viene usata dalle transazioni per indicare al gestore del buffer che il blocco della pagina è stato modificato. L'effetto è la modifica del bir di stato `J` a 1.

La primitiva `unfix` viene usata dalle transazioni per indicare che la transazione ha terminato di usare il blocco. L'effetto è il decremento del contatore I che indica l'uso della pagina.

La primitiva `force` viene usata per salvare in memoria secondaria in modo sincrono il blocco contenuto nella pagina. L'effetto è il salvataggio in memoria secondaria del blocco e il bit di stato J posto a zero.

La primitiva `flush` viene usata dal gestore del buffer per salvare blocchi sulla memoria secondaria in modo asincrono. Tale operazione libera *dirty* (*il bit di stato viene posto a zero*).

1.2.3 Gestore dell'affidabilità

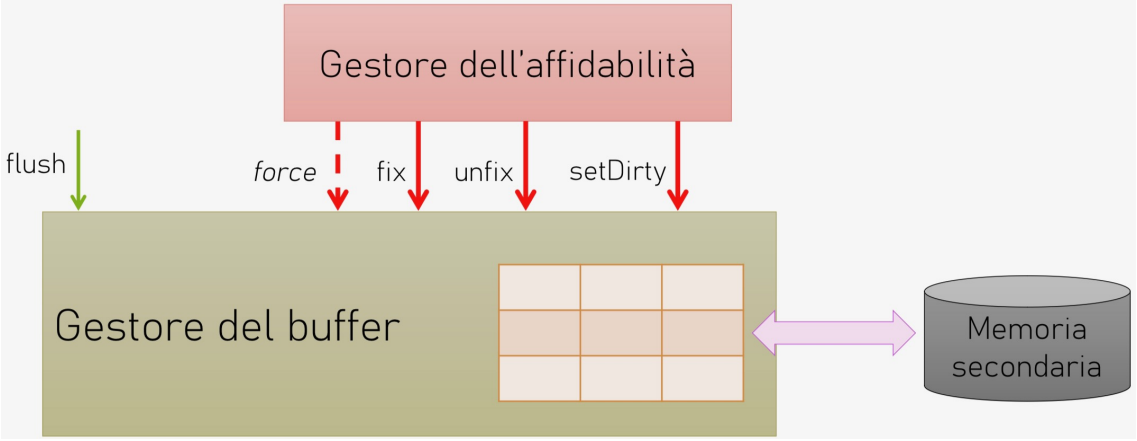


Figura 1.2.2: Gestore dell'affidabilità

È il modulo responsabile di ciò che riguarda l'esecuzione delle istruzioni per la gestione delle transazioni e la della realizzazione delle operazioni necessarie al ripristino della base di dati dopo eventuali malfunzionamenti.

Per il suo funzionamento il gestore dell'affidabilità deve disporre di un dispositivo di memoria stabile, cioè resistente a guasti.

1.2.4 Gestione del file di `LOG`

In memoria stabile viene memorizzato il file di `LOG` che registra in modo sequenziale le operazioni eseguite dalle transazioni sulla base di dati. I record memorizzati sul file di `LOG` di suddividono in record di transazione e di sistema.

I record di transazione si classificano in:

- Begin della transazione `T`: `B(T)`.
- Commit della transazione `T`: `C(T)`.
- Abort della transazione `T`: `A(T)`.
- Insert, Delete e Update eseguiti dalla transazione `T` sull'oggetto `O`:
  - `I(T,O,AS)`: dove `AS` indica After State.
  - `D(T,O,BS)`: dove `BS` indica Before State.
  - `U(T,O,BS,AS)`

I record di transazione salvati nel LOG consentono di eseguire, in caso di ripristino, le seguenti operazioni:

- `UNDO`: per disfare un'azione su un oggetto O è sufficiente ricopiare in O il valore BS; l'insert/delete viene disfatto cancellando/inserendo O.
- `REDO`: per rifare un'azione su un oggetto O è sufficiente ricopiare in O il valore AS; l'insert/delete viene rifatto cancellando/inserendo O.

Gli inserimenti controllano sempre l'esistenza di O (*non si inseriscono duplicati*).

I record di sistema:

- Operazioni di `DUMP` della base di dati: `DUMP`.
- Operazioni di `CheckPoint`: `CK`$(T_1, \ldots, T_n)$ indica che l'esecuzione del CheckPoint le transazioni attive erano $T_1, \ldots, T_n$.
- L'operazione svolta periodicamente dal gestore dell'affidabilità prevede:
  - Sospensione delle operazioni di scrittura, commit e abort delle transazioni.
  - Esecuzione della primitiva force sulle pagine *dirty* di transazioni che hanno eseguito il commit.

- Scritture sincrona sul file `LOG` del record di CheckPoint con gli identificatori delle transazioni attive.
- Ripresa delle operazioni di scrittura, commit e abort delle transazioni.

Regole di scrittura sul `LOG`

- Regola WAL (*Write Ahead Log*): i record di log devono essere scritti sul `LOG` prima dell'esecuzione delle corrispondenti operazioni sulla base di dati (*garantisce la possibilità di fare sempre* `UNDO`).
- Regola Commit-Precedenza: i record di log devono essere scritti prima dell'esecuzione del commit della transazione (*Garantisce la possibilità di fare* `REDO`).

Tali regole consentono inoltre di salvare i blocchi delle pagine "dirty" in modo totalmente asincrono rispetto al commit delle transazioni.

Azione di commit

Una transazione `T` sceglie in modo atomico l'esito di commit nel momento in cui scrive nel file di log in modo sincrono (*primitiva* `force`) il suo record di commit `C(T)`.

- Se il guasto avviene prima della scrittura del record di commit `UNDO`.
- Se il guasto avviene dopo la scrittura del record di commit `REDO`.

Gestore dell'affidabilità

Il gestore dell'affidabilità gestisce due tipologie di guasto:

- Guasto di sistema: perdita del contenuto della memoria centrale, gestibile attraverso la ripresa a caldo.
- Guasto di dispositivo: perdita di parte o tutto il contenuto della base di dati in memoria, gestibile attraverso la ripresa a freddo.

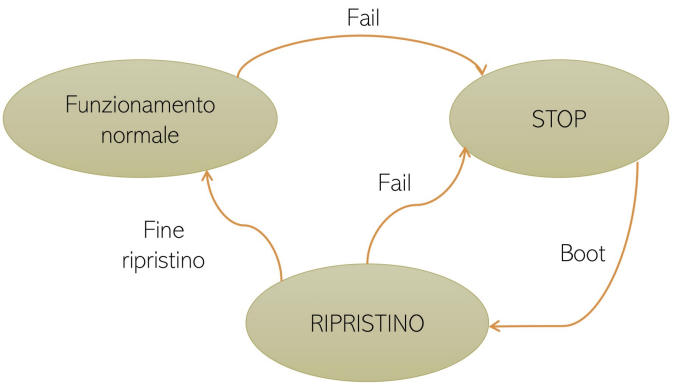


Figura 1.2.3: Modello di funzionamento

1.2.5 Ripresa a caldo

- Si accede all'ultimo blocco di `LOG` e si ripercorre all'indietro il `LOG` fino al più recente record `CK`.
- Si decidono le transazioni da rifare/disfare inizializzando l'insieme `UNDO` con le transazioni attive al `CK` e l'insieme `REDO` con l'insieme vuoto.
- Si ripercorre in avanti il `LOG` e per ogni record `B(T)` incontrato si aggiunge `T` a `UNDO` e per ogni record `C(T)` incontrato si sposta `T` da `UNDO` a `REDO`.
- Si ripercorre all'indietro il `LOG` disfacendo le operazioni eseguite dalle transazioni in `UNDO` risalendo fino alla prima azione della transazione più vecchia.
- Si rifanno le operazioni delle transazioni dell'insieme `REDO`.

1.2.6 Ripresa a freddo

- Si accede al `DUMP` della base di dati e si ricopia selettivamente la parte deteriorata della base di dati.
- Si accede al `LOG` risalendo al record `DUMP`.
- Si ripercorre in avanti il `LOG` rieseguendo tutte le operazioni relative alla parte deteriorata comprese le azioni di commit e abort.
- Si applica una ripresa a caldo.

Capitolo 2

Strutture fisiche e strutture di accesso ai dati

2.1 Gestore dei metodi di accesso

Il gestore dei metodi d'accesso è un modulo del DBMS che esegue il piano di esecuzione prodotto dall'ottimizzatore e produce sequenze di accessi ai blocchi della base di dati presenti in memoria secondaria.

Metodi d'accesso

Sono software che implementano gli algoritmi di accesso e manipolazione dei dati organizzati in specifiche strutture fisiche.

Ogni metodo d'accesso ai dati conosce:

- L'organizzazione delle tuple/record di indice nei blocchi `DATI/INDICE` salvati in memoria secondaria (*come una tabella/indice viene organizzata in blocchi `DATI` della memoria secondaria*);
- L'organizzazione fisica interna dei blocchi sia quando contengono `DATI` (*vale a dire, tuple di una tabella*) sia quando contengono strutture fisiche di accesso o `INDICI` (*vale a dire, record di un indice*).

2.1.1 Organizzazione di un blocco DATI

In un blocco `DATI` sono presenti informazioni utili e informazioni di controllo:

- Informazioni utili: tuple della tabella;
- Informazioni di controllo: dizionario, bit di parità, altre informazioni del file system o della specifica struttura fisiche.

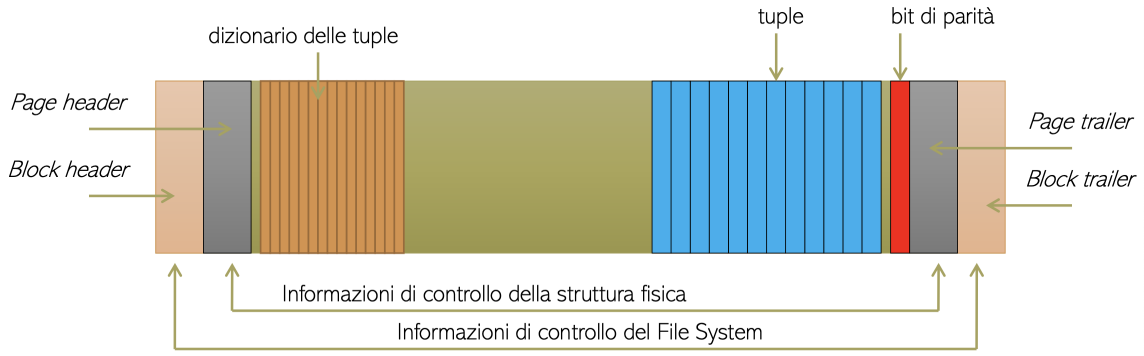


Figura 2.1.1: Struttura della pagina

2.1.2 Struttura del dizionario

Vi sono due opzioni:

- Tuple di **lunghezza fissa**: il dizionario non è necessario, si deve solo memorizzare la dimensione delle tuple e l'offset del punto iniziale.
- Tuple di **lunghezza variabile**: il dizionario memorizza l'offset di ogni tuple presente nel blocco e di ogni attributo di ogni tupla.

La lunghezza massima della tupla è pari alla dimensione massima dell'area disponibile su un blocco, altrimenti va gestito il caso di tuple memorizzate su più blocchi.

2.1.3 Operazioni sulle pagine

- Inserimento di una tupla:
  - Se esiste spazio contiguo sufficiente: inserimento semplice;
  - Se non esiste spazio contiguo, ma esiste spazio sufficiente: riorganizza lo spazio ed esegue un inserimento semplice;
  - Se non esiste spazio sufficiente: operazione rifiutata.
- Cancellazione: sempre possibile anche senza riorganizzare;
- Accesso ad una tupla;
- Accesso ad un attributo di una tupla;
- Accesso sequenziale (*di solito in orine di chiave primaria*);
- Riorganizzare.

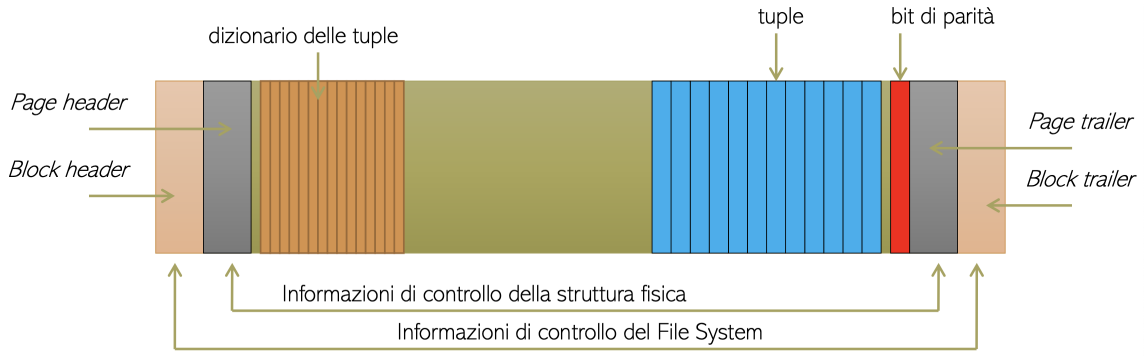


Figura 2.1.2: Rappresentazione di tabella a livello fisico

2.2 Struttura sequenziale ordinata

Una struttura sequenziale ordinata è un **file sequenziale** dove le tuple sono ordinate secondo **chiave di ordinamento**.

Esempio

Figura 2.2.1: Esempio

2.2.1 Operazioni

L'inserimento di una tupla avviene seguendo i seguenti passi:

- Individuazione del blocco che **contiene la tupla che precede**, nell'ordine della chiave, la tupla da inserire.
- Inserimento della tupla nuova in B; se l'operazione non va a buon fine si aggiunge un nuovo blocco (*detto, overflow page*) alla struttura: il blocco contiene la nuova tupla, altrimenti si prosegue.
- Aggiustamento della catena di puntatori.

La scansione sequenziale ordinata avviene secondo la chiave (*seguendo i puntatori*).

La cancellazione di una tupla avviene seguendo i seguenti passi:

- Individuazione del blocco `B` che contiene la tupla da cancellare.
- Cancellazione della tupla `B`.
- Aggiustamento della catena di puntatori.

La riorganizzazione avviene assegnando le tuple ai blocchi ed opportuni coefficienti di riempimento, riaggiustando i puntatori.

2.3 Indici

Per aumentare le prestazioni degli accessi alle tuple memorizzate nelle strutture fisiche (**file sequenziale**), si introducono strutture ausiliarie (*dette strutture ai dati o indici*); tali strutture velocizzano l'accesso casuale via chiave di ricerca. La chiave di ricerca è un insieme di attributi utilizzati dall'indice nella ricerca.

Gli indici su file sequenziali possono essere di due categorie:

- **Indice primario**: in questo caso la **chiave di ordinamento** del file sequenziale coincide con la **con la chiave di ricerca** dell'indice.
- **Indice secondario**: in questo caso invece la **chiave di ordinamento** e la **chiave di ricerca** sono **diverse**.

2.3.1 Indice primario

L'indice primario usa una **chiave di ricerca** che coincide con la chiave di ordinamento del file sequenziale.

Ogni record dell'indice primario contiene una coppia $\langle v_i, p_i \rangle$ dove:

- v_i: valore della chiave di ricerca.
- p_i: puntatore al primo record del file sequenziale con chiave v_i.

Esistono due varianti dell'indice primario:

- Indice denso: per ogni occorrenza della chiave presente nel file esiste un corrispondente record nell'indice.
- Indice sparso: solo per alcune occorrenze della chiave presente nel file esiste un corrispondente record nell'indice, tipicamente una per blocco.

Operazioni di ricerca

La ricerca di una tupla con chive di ricerca K si differenzia per le due modalità:

- Denso: in questo caso K è presente nell'indice e si esegue una scansione sequenziale dell'indice per trovare il record (K, p_k) e si accede al file attraverso puntatore p_k. Il costo dell'operazione è di un accesso all'indice più un accesso al blocco dati.

- Sparso: in questo caso k potrebbe non essere presente nell'indice e si esegue una scansione sequenziale dell'indice fino al record (K', p'_k) dove k' è il valore più grande che sia minore o uguale a K. Si esegue un accesso attraverso il puntatore p'_k e si scansiona il file (*blocco corrente*) per trovare le tuple con chiave K. Il costo dell'operazione è di un accesso all'indice più un accesso al blocco dati.

Operazioni di inserimento

L'inserimento nel file sequenziale si differenzia per le due modalità:

- Denso: l'inserimento nell'indice avviene solo se la tupla inserita nel file ha un valore di chiave K che non è già presente.
- Sparso: l'inserimento avviene solo quando, per effetto dell'inserimento di una nuova tupla, si aggiunge un blocco dati alla struttura; in tutti gli altri casi l'indice rimane invariato.

Operazioni di cancellazione

La cancellazione di un record nell'indice si differenzia per le due modalità:

- Denso: la cancellazione nell'indice avviene solo se la tupla cancellata nel file è l'ultima tupla con valore di chiave K.
- Sparso: la cancellazione nell'indice avviene solo quando K è presente nell'indice e il corrispondente blocco viene eliminato; altrimenti, se il blocco sopravvive, va sostituito K nel record dell'indice con il primo valore K' presente nel blocco.

2.3.2 Indice secondario

L'indice secondario una una chiave che non coincide con la chiave di ordinamento del file sequenziale. Ogni record dell'indice secondario contiene una coppia $\langle v_i, p_i \rangle$:

- v_i: valore nella chiave di ricerca.
- p_i: puntatore al bucket di puntatori che individuano nel file sequenziale tutte le tuple con valore di chiave v_i.

A differenzia degli indici primari, gli indici secondari sono sempre densi.

2.3.3 Operazioni di ricerca

L'operazione di ricerca di una tupla con chiave di ricerca K avviene eseguendo una scansione sequenziale dell'indice per trovare il record (K, p_k), viene eseguito l'accesso al bucket di puntatori attraverso il puntatore p_k e successivamente un accesso al file attraverso i puntatori del bucket B.

Il costo dell'operazione è di un accesso all'indice più l'accesso al bucket e più gli n accessi alle pagine dati.

Operazioni di inserimento e cancellazione

Tali operazioni vengono eseguite come per l'indice primario denso con più l'aggiornamento dei bucket.

2.4 B+-Tree

Quando l'indice aumenta di dimensioni, non può risiedere in memoria centrale: di conseguenza deve essere in grado di essere gestito in memoria secondaria. È possibile utilizzare un **file sequenziale ordinato** per rappresentare l'**indice in memoria secondaria**, le prestazioni di accesso a tale struttura fisica a fronte di inserimenti/cancellazioni tendono a denigrare e richiedono frequenti riorganizzazioni. Inoltre è disponibile solo l'accesso sequenziale.

Per superare tale problema si introducono per gli indici strutture fisiche diverse, tra cui **strutture ad albero** e le **strutture ad accesso calcolato**.

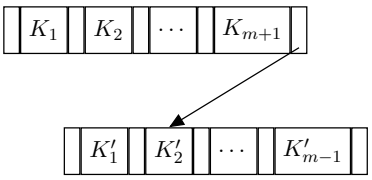
Nel B+-tree ogni nodo corrisponde ad una pagina della memoria secondaria e i legami tra nodi diventano puntatori a pagina. Ogni nodo ha un numero elevato di figli, quindi l'albero ha tipicamente **pochi livelli e molti nodi foglia**.

L'albero è inoltre bilanciato, ciò significa che la lunghezza dei percorsi che collegano la radice ai nodi foglia è costante, inoltre gli inserimenti e le cancellazioni non alterano le prestazioni dell'accesso ai dati: l'albero si mantiene bilanciato.

2.4.1 Struttura

La struttura di un B+-tree può contenere fino a $n-1$ valori ordinati di chiave di ricerca e fino a n puntatori.

Il puntatore p_m del nodo al livello superiore L_i punta al nodo del livello inferiore L_{i+1} se esiste.



Se un puntatore è compreso tra i valori K_{i-1} e K_i tale puntatore punterà al sottoalbero con chiavi k tali che: $K_{i-1} < k < K_i$

2.4.2 Vincoli di riempimento

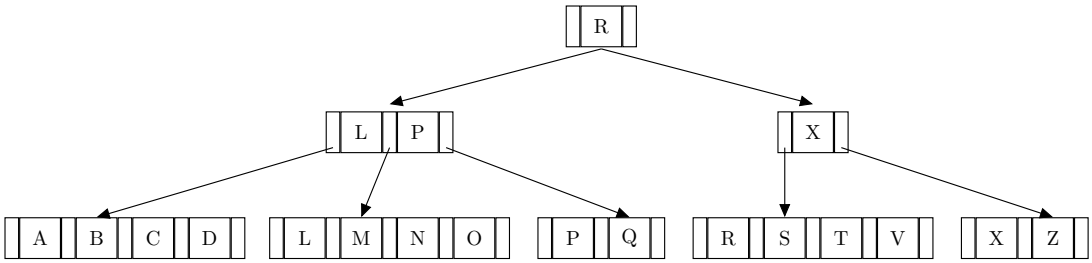
Ogni nodo foglia contiene un numero di valori chiave ($\#chiavi$) vincolato come segue:

$$\left\lceil \frac{n-1}{2} \right\rceil \leq \#chiavi \leq n-1$$

Ogni nodo intermedio contiene un numero di puntatori (*#puntatori*) vincolato come segue (*per la radice non vale il minimo*):

$$\left\lceil \frac{n}{2} \right\rceil \leq \#puntatori \leq n$$

Esempio fan-out 4



2.4.3 Operazioni

Ricerca

La ricerca si effettua attraverso diversi passi:

1. Cercare nel nodo radice il più piccolo valore di chiave maggiore di `K`.
   - Se esiste allora si segue il puntatore p_i.
   - Se tale valore non esiste si segue il puntatore p_m.
2. Se il nodo raggiunto è un nodo foglia cercare il valore `K` nel nodo e seguire il corrisponde puntatore verso le tuple, altrimenti riprendere il passo (1).

Inserimento con chiave `K`

1. Ricerca del nodo foglia `NF` dove il valore `K` va inserito
2. Se `K` è presente in `NF`, allora:
   - Indice primario: nessuna azione;
   - Indice secondario: aggiornamento del bucket di puntatori.

   Altrimenti, inserire `K` in `NF` rispettando l'ordine e:
   - Indice primario: inserire il puntatore alla tupla con valore K della chiave;
   - Indice secondario: inserire un nuovo bucket di puntatori contenente il puntatore alla tupla con valore `K` della chiave.

Se non è possibile inserire `K` in `NF`, allora si esegue uno split di `NF`.

Nel nodo da dividere esistono n valori chiave, si procede come segue:

- Creare due nodi foglia;
- Inserire i primi $\lceil (n-1)/2 \rceil$ valori nel primo;
- Inserire i rimanenti nel secondo;
- Inserire nel nodo padre un nuovo puntatore per il secondo nodo foglia generato e riaggiustare i valori chiave presenti nel nodo padre.
- Se anche il nodo padre è pieno lo spit si propaga al padre e così via.

Cancellazione con chiave `K`

1. Ricerca del nodo foglia `NF` dove il valore `K` va cancellato.
2. Cancellare `K` da `NF` insieme al suo puntatore:
   - Indice primario: nessuna ulteriore azione;
   - Indice secondario: liberare il bucket di puntatori.

Se dopo la cancellazione di `K` da `NF` viene violato allora si esegue un merge di `NF`.

2.5 Hash

Le caratteristiche generali delle strutture ad accesso calcolato:

- Si basano su una funzione di hash che mappa i valori della chiave di ricerca sugli indirizzi di memorizzazione delle tuple nelle pagine dei dati della memoria secondaria.

Uso pratico di una funzione di hash negli indici

- Si stima il numero di valori chiave che saranno contenuti nella tabella.
- Si alloca un numero di bucket ($\mathcal{B}$) di puntatori uguale al numero stimato.
- Si definisce una funzione di folding che trasforma i valori chiave in numeri interi positivi ($\mathbb{K} \to \mathbb{Z}^+$).
- Si definisce una funzione di hashing ($\mathbb{Z}^+ \to \mathcal{B}$).

La funzione di hashing distribuisce in modo uniforme e casuale i valori della chiave nei bucket. È pesante cambiare la funzione di hashing dopo che la struttura d'accesso è stata riempita; si deve costruire l'indice da capo.

2.5.1 Operazioni

Ricerca

Dato un valore di chiave $\mathcal{K}$ trovare la corrispondente tupla:

- Calcolare $b = h(f(K))$ (*costo zero*);
- Accedere al bucket b (*costo: 1 accesso a pagina*);

- Accedere a n tuple attraverso i puntatori del bucket (*costo: m accessi a pagina $m \leq n$*).

Inserimento e cancellazione

Di complessità simile alla ricerca.

2.5.2 Osservazioni

La struttura ad accesso calcolato funziona se i buckets conservano un basso coefficiente di riempimento. Infatti il problema delle strutture ad accesso calcolato è la gestione delle collisioni.

La collisione si verifica quando, dati due valori di chiave $\mathtt{K}_1$ e $\mathtt{K}_2$, con $\mathtt{K}_1 \neq \mathtt{K}_2$, risulta

$$h(f(\mathtt{K}_1)) = h(f(\mathtt{K}_2))$$

Un numero eccessivo di collisioni porta alla saturazione del bucket corrispondente.

2.5.3 Collisioni

La probabilità che un bucket riceva t chiavi su n inserimenti:

$$p(t) = \binom{n}{t} \left(\frac{1}{B}\right)^t \left(1 - \frac{1}{B}\right)^{(n-t)}$$

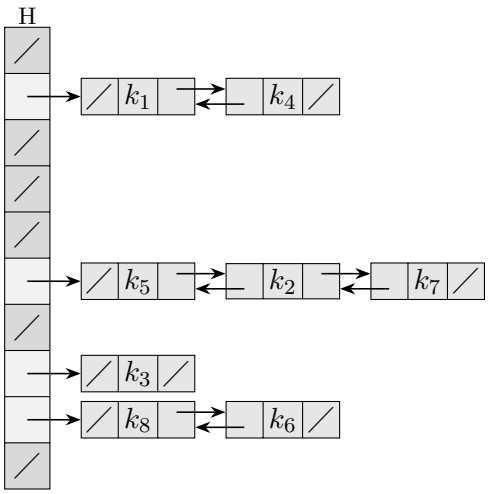
Dove B è il numero totale di buckets.

La probabilità di avere più di F collisioni, dove F è il numero di puntatori nel bucket è:

$$p_k = 1 - \sum_{i=0}^{F} p(i)$$

2.5.4 Gestione delle collisioni

Un numero eccessivo di collisioni sullo stesso indirizzo porta alla saturazione del bucket corrispondente. Per gestire tale situazione si prevede la possibilità di allocare buckets di overflow, collegati di base:



2.6 Confronto tra B+-tree e Hashing

Per quanto riguarda la ricerca con selezioni basate su uguaglianza , ovvero $A = cost$, l'hashing (senza overflow buckets) è a tempo costante, mentre il B+-tree è a tempo logaritmico nel numero di chiavi.

Le selezioni basate su intervalli (*range*), l'hashing: ha un numero elevato di selezioni su condizioni si uguaglianza per scandire tutti i valori del range, mentre il B+-tree è a tempo logaritmico per accedere al primo valore dell'intervallo, scandendo i nodi foglia fino all'ultimo valore compreso.

Gli inserimenti e le cancellazioni nell'hashing sono a tempo costante, più l'eventuale gestione dell'overflow, mentre per il B+-tree sono a tempo logaritmico nel numero di chiavi, più eventuali split e merge.

Capitolo 3

Esecuzione concorrente di transazioni

Per gestire con prestazioni accettabili il carico di lavoro tipico delle applicazioni gestionali, un DBMS deve eseguire le transazioni in modo concorrente. L'esecuzione concorrente di transazioni senza controllo può generare anomalie o problemi di correttezza. È quindi necessario introdurre dei meccanismi di controllo nell'esecuzione delle transazioni per evitare tali anomalie.

3.1 Anomalie di esecuzione concorrente

Le anomalie tipiche sono:

- Perdita di aggiornamento (*update*): gli effetti di una transazione concorrente sono persi.
- Lettura inconsistente: accessi successivi ad uno stesso dato all'interno di una transazione ritornano valori diversi.
- Lettura sporca: viene letto un dato che rappresenta uno stato intermedio nell'evoluzione di una transazione.
- Aggiornamento/inserimento fantasma: valutazioni diverse di valori aggregati a causa di inserimenti intermedi.

Notazione

- Indicheremo con t_i una transazione.
- $r_i(x)$: è un'operazione di lettura eseguita dalle transazioni t_i sulla risorsa x.
- $w_i(x)$: è un'operazione di scrittura eseguita dalle transazioni t_i sulla risorsa x.

3.2 Schedule

Lo schedule è una sequenza di istruzioni di lettura e scrittura eseguite su risorse della base di dati da diverse transazioni concorrenti.

$$\mathcal{S} : r_1(x)r_2(z)w_1(x)w_2(z)$$

Esso rappresenta una possibile esecuzione concorrente di diverse transazioni.

3.2.1 Schedule seriale

Lo schedule seriale è uno schedule dove le operazioni di ogni transazione compaiono in sequenza **senza essere inframezzate** da operazioni di altre transazioni.

$$\mathcal{S}_1 : r_1(x)r_2(x)w_2(x)w_1(x) \qquad \textit{Non è seriale}$$

$$\mathcal{S}_2 : r_1(x)w_1(x)r_2(x)w_2(x) \qquad \textit{È seriale}$$

3.2.2 Schedule serializzabile

Lo schedule serializzabile è uno schedule equivalente ad uno schedule seriale.

A questo punto occorre definire il concetto di **equivalenza** tra schedule.

Equivalenza

L'idea è che si vogliono considerare equivalenti due schedule che **producono gli stessi effetti sulla base di dati**. Quindi accettare schedule serializzabili significa accettare schedule che hanno gli stessi effetti di uno schedule seriale.

3.3 Equivalenza tra schedule

3.3.1 Ipotesi di commit-proiezione

Si suppone che le transazioni abbiano esito noto. Quindi si tolgono dagli schedule tutte le operazioni delle transazioni che non vanno a buon fine.

Richiedono questa ipotesi le seguenti tecniche di gestione della concorrenza:

- Gestione della concorrenza basata sulla **view-equivalenza**
- Gestione della concorrenza basata sulla **conflict-equivalenza**.

View-equivalenza

Prima di definire la view-equivalenza vediamo delle definizioni preliminari.

Legge da

Dato uno schema $\mathcal{S}$ si dice che un'operazione di lettura $r_i(x)$ che compare in $\mathcal{S}$, fa parte dell'insieme `LEGGE_DA`, se $w_j(x)$ precede $r_i(x)$ in $\mathcal{S}$ e non vi è alcuna operazione $w_k(x)$ tra le due.

Esempio

$$\mathcal{S}_1 : r_1(x)r_2(x)w_2(x)w_1(x) \qquad \texttt{LEGGE\_DA}(\mathcal{S}_1) = \varnothing$$
$$\mathcal{S}_1 : w_1(x)r_2(x)w_2(x)w_1(x) \qquad \texttt{LEGGE\_DA}(\mathcal{S}_1) = \{(r_2(x), w_1(x))\}$$

Scritture finali

Dato uno schema $\mathcal{S}$ si dice che un'operazione di scrittura $w_i(x)$, che compare in $\mathcal{S}$, appartiene all'insieme `SCRITTURE_FINALI` se è l'ultima operazione di scrittura della risorsa x in $\mathcal{S}$.

Esempio

$$\mathcal{S}_1 : r_1(x)r_2(x)w_2(x)w_2(y)w_1(x) \qquad \texttt{SCRITTURE\_FINALI}(\mathcal{S}_1) = \{w_3(y), w_1(x)\}$$
$$\mathcal{S}_1 : r1(x)w_1(x)r_2(x)w_2(x)w_3(y) \qquad \texttt{SCRITTURE\_FINALI}(\mathcal{S}_1) = \{(w_2(x), w_3(y))\}$$

View-equivalenza

Da schedule S_1 e S_2 sono view-equivalenti ($\mathcal{S}_1 \approx_V \mathcal{S}_2$) se possiedono le stesse relazioni `LEGGE_DA` e lo stesso insieme `SCRITTURE_FINALI`.

View-serializzabilità

View-serializzabilità

Uno schedule $\mathcal{S}$ è view-serializzabile (VSR) se esiste uno schedule seriale $\mathcal{S}'$ tale che $\mathcal{S}' \approx_V \mathcal{S}$.

Gli schedule seriali $\mathcal{S}'$ si generano considerando tutte le possibili permutazioni delle transazioni che compaiono in $\mathcal{S}$. Inoltre, non si deve cambiare l'ordine delle operazioni di una transazione, ciò significherebbe modificare le transazione.

3.3.2 View-equivalenza

L'algoritmo per il test di view-equivalenza tra due schedule è di complessità lineare. Invece, l'algoritmo per il test di view-serializzabilità di uno schedule è di complessità esponenziale (*è necessario generare i più possibili schedule seriali*).

In conclusione la view-serializzabilità richiede algoritmi di complessità troppo elevata, richiedendo inoltre l'ipotesi di commit proiezione, perciò non è applicabile nei sistemi reali.

3.3.3 Conflict-equivalenza

Conflitto

Dato uno schedule $\mathcal{S}$ si dice che una coppia di operazioni (a_i, a_j), dove a_i e a_j compaiono nello schedule $\mathcal{S}$, rappresentano un conflitto se:

- $i \neq j$ (*transazioni diverse*).
- Entrambe operano sulla stessa risorsa.
- Almeno una di esse è una operazione di scrittura.
- a_i compare in $\mathcal{S}$ di a_j.

Definito il concetto di conflitto, possiamo definire la conflict-equivalenza.

Conflict-equivalenza

Due schedule $\mathcal{S}_1$ e $\mathcal{S}_2$ sono conflict-equivalenti ($\mathcal{S}_1 \approx_C \mathcal{S}_2$) se possiedono le stesse operazioni e gli stessi conflitti (*le operazioni in conflitto sono nello stesso ordine nei due schedule*).

Conflict-serializzabilità

Due schedule $\mathcal{S}$ è conflict-serializzabile (*CSR*) se esiste uno schedule seriale $\mathcal{S}'$ tale che $\mathcal{S}' \approx \mathcal{S}$.

Osservazioni

L'algoritmo per il test di conflict-serializzabilità di uno schedule è di **complessità lineare**, e segue i seguenti passi:

- Si costruisce il grafo dei conflitti `G(N,A)` dove $N = \{t_1, \ldots, t_n\}$ con $t_1, \ldots, t_n$ transazioni di $\mathcal{S}$.
- $(t_{i,j}) \in A$ se esiste almeno un conflitto $(a_i, a_j) \in \mathcal{S}$.
- Se il grafo così costruito è aciclico allora $\mathcal{S}$ è conflict-serializzabile.

3.3.4 Verifica algoritmo sul grafo

Se $\mathcal{S}$ è CSR allora il suo grafo è aciclico.

Dimostrazione. Se uno schedule $\mathcal{S}$ è CSR allora è conflict-equivalente ad uno schedule seriale $\mathcal{S}_{ser}$. Supponiamo che le transazioni in $\mathcal{S}_{ser}$ siano ordinate secondo il `TID` : $t_1, t_2, \ldots, t_n$. Poiché $\mathcal{S} \approx_C \mathcal{S}_{ser}$, $\mathcal{S}_{ser}$ ha tutti i conflitti di $\mathcal{S}$ esattamente nello stesso ordine.

Ora nel grafo di $\mathcal{S}_{ser}$ poiché ci possono essere solo archi (i, j) con $i < j$ e quindi il grafo non può avere cicli, perché un ciclo richiede almeno un arco (i, j) con $i > j$.

Poiché $\mathcal{S} \approx_C \mathcal{S}_{ser}$ ha come già detto gli stessi conflitti di quest'ultimo grafo che risulterà anch'esso aciclico. □

Se il grafo di $\mathcal{S}$ è aciclico allora $\mathcal{S}$ è CSR.

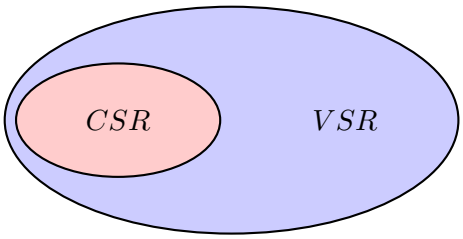
Dimostrazione. Se il grafo di $\mathcal{S}$ è aciclico. allora esiste fra i nodi un ordinamento topologico, vale a dire una numerazione dei nodi tale che il grafo contiene solo archi (i, j) con $i < j$. Per dimostrare che $\mathcal{S}$ è CSR occorre generare a partire da questa osservazione uno schedule seriale che abbia gli stessi conflitti di $\mathcal{S}$.

Lo schedule seriale le cui transazioni sono ordinate secondo l'ordinamento topologico è conflict-equivalente a $\mathcal{S}$, in quanto ha lo stesso grafo di $\mathcal{S}$ e quindi gli stessi conflitti di $\mathcal{S}$.

Quindi l'aciclicità del grafo di $\mathcal{S}$ assicura la possibilità di ottenere uno schedule conflict-equivalente a $\mathcal{S}$ e quindi assicura che $\mathcal{S}$ sia CSR. □

CSR vs VSR

Sappiamo che la conflict-serializzabilità è condizione sufficiente ma non necessaria per la view-serializzabilità.



Le tecniche per il controllo della concorrenza che non richiedono di conoscere l'esito delle transazioni sono:

- Timestamp per il controllo della concorrenza (*TS buffer*).
- Locking a due fasi stretto (*2PL stretto*)

3.4 Locking a due fasi

Si tratta di un metodo applicato nei **sistemi reali** per la gestione dell'esecuzione concorrente di transazioni che non richiede di conoscere a priori l'esito delle transazioni. Tre sono gli aspetti che caratterizzano il locking a due fasi:

- Il meccanismo di base per la gestione dei lock.

- La politica di concessione dei lock sulle risorse.
- La regola che garantisce la serializzabilità.

3.4.1 Meccanismo di base

Si basa sull'introduzione di alcune primitive di lock che consentono alle transazioni di bloccare (*lock*) le risorse sulle quali vogliono agire con operazioni di lettura e scrittura.

- `r_lock`$_k$`(x)`: richiesta di un lock condiviso da parte della transazione t_k sulla risorsa x per eseguire una lettura.
- `w_lock`$_k$`(x)`: richiesta di un lock esclusivo da parte della transazione t_k sulla risorsa x per eseguire una scrittura.
- `unlock`$_k$`(x)`: richiesta da parte della transazione t_k di liberare la risorsa x da un precedente lock.

Regole per l'uso delle primitive da parte delle transazioni

- $\mathbf{R}_1$: ogni lettura deve essere preceduta da un `r_lock` e seguita da un `unlock`
- $\mathbf{R}_2$: ogni scrittura deve essere preceduta da un `w_lock` e seguita da un `unlock`. Non sono ammessi più `w_lock` (oppure `w_lock` e `r_lock`) contemporanei sulla stessa risorsa (*lock esclusivo*).

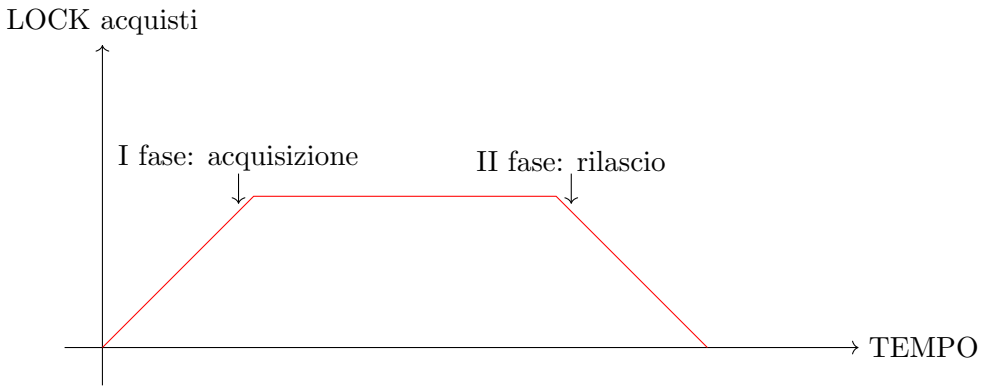
Se la transazione segue le regole R_1 e R_2 si dice **ben formata** rispetto al locking.

3.4.2 Politica di concessione dei lock

<table>
<tr><th>Stato x / Richiesta</th><th colspan="2">LIBERO</th><th colspan="2">R_LOCK</th><th colspan="2">W_LOCK</th></tr>
<tr><td>r_lock$_K$(x)</td><td>Esito
OK</td><td>Operazioni
s(x) = r_lock
c(x) ={k}</td><td>Esito
OK</td><td>Operazioni
c(x) = c(x)∪{k}</td><td>Esito
Attesa</td><td>Operazioni
-</td></tr>
<tr><td rowspan="3">w_lock$_K$(x)</td><td rowspan="3">Esito
OK</td><td rowspan="3">Operazioni
s(x) = w_lock</td><td colspan="2">if |c(x)|=1 and k∈c(x) then</td><td rowspan="3">Esito
Attesa</td><td rowspan="3">Operazioni
-</td></tr>
<tr><td>Esito
OK</td><td>Operazioni
s(x) = w_lock</td></tr>
<tr><td>else
Attesa</td><td>-</td></tr>
<tr><td>unlock$_K$(x)</td><td>Esito
Errore</td><td>Operazioni
-</td><td>Esito
OK</td><td>Operazioni
c(x) = c(x) – {k}
if c(x)=∅ then
s(x)=Libero
verifica coda</td><td>Esito
OK</td><td>c(x)=∅
s(x)=Libero
verifica coda</td></tr>
</table>

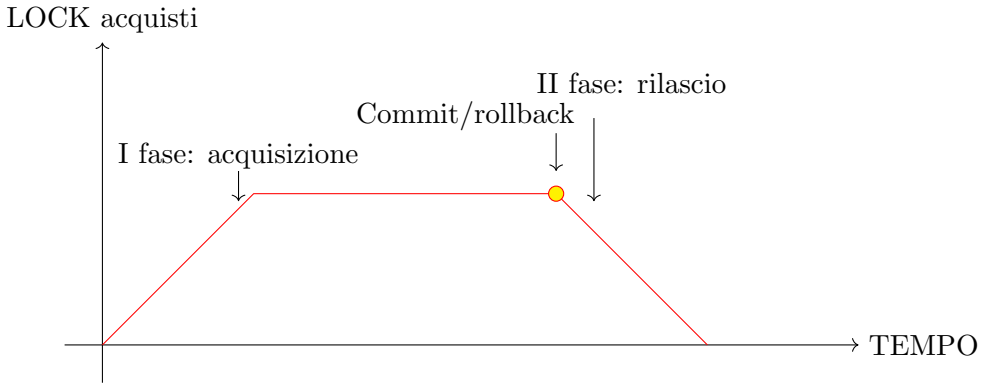
3.4.3 Serializzabilità

La regola che garantisce la serializzabilità (*da cui prende il nome il metodo 2PL*), una transazione dopo aver rilasciato un `LOCK` non può acquisirne altri.

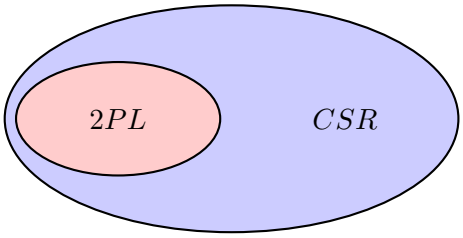


3.4.4 2PL stretto

Per rimuovere completamente l'ipotesi di commit-proiezione, una transazione può rilasciare i `LOCK` solo quando ha eseguito correttamente un `COMMIT` o un `ROLLBACK`.



Relazione tra 2PL e CSR



3.4.5 Blocco critico

Il blocco critico è una situazione di blocco che si verifica nell'esecuzione di transazioni concorrenti quando:

- Due transazioni hanno bloccato delle risorse: t_1 ha bloccato r_1 r t_2 ha bloccato r_2.
- Inoltre t_1 è in attesa sulla risorsa r_2 e t_2 è in attesa sulla risorsa r_1.

Se il numero medio di tuple per tabella è n e la granularità dei lock è la tupla, la probabilità che si verifichi un lock tra due transazioni è pari a $\frac{1}{n^2}$.

Tecniche di risoluzione

- **Timeout**: una transazione in attesa su una risorsa trascorso il timeout viene abortita.
- **Prevenzione**:
- – Una transazione blocca tutte le risorse a cui deve accedere in lettura o scrittura in una sola volta (*o blocca tutto o non blocca nulla*).
  - Ogni transazione t_i acquisisce un timestamp $\texttt{TS}_i$ all'inizio della sua esecuzione. La transazione t_i può attendere una risorsa bloccata da t_j solo se vale una certa condizione sui TS ($\texttt{TS}_i < \texttt{TS}_j$) altrimenti viene abortita e fatta ripartire con lo stesso TS.
- **Rilevamento**: si esegue un'analisi periodica della tabella di `LOCK` per rilevare la presenza di un blocco critico (*corrisponde ad un ciclo nel grafo delle relazioni di attesa*). Questa tecnica è quella più frequentemente adottata dai sistemi reali.

La scelta di gestire i lock in lettura come lock condivisi produce un ulteriore problema chiamato starvation che tuttavia nel contesto delle DBMS risulta poco probabile.

Se una risorsa x viene costantemente bloccata da una sequenza di transazioni che acquisiscono `r_lock` su x, un'eventuale transazione in attesa di scrivere su x viene bloccata per un lungo periodo fino alla fine della sequenza di letture.

Anche se poco probabile tale evenienza può essere scongiurata con tecniche simili a quanto visto per il blocco critico. In particolare è possibile analizzare la tabella delle relazioni di attesa e verificare da quanto tempo le transazioni stanno attendendo la risorsa e di conseguenza sospendere temporaneamente la concessione di lock in lettura condivisi per consentire la scrittura da parte della transazione in attesa.

Poiché risulta molto oneroso per il sistema gestire l'esecuzione concorrente di transazioni con conseguente diminuzione delle prestazioni del sistema, SQL prevede la possibilità di rinunciare in tutto o in parte al controllo di concorrenza per aumentare le prestazioni del sistema.

Ciò significa che è possibile a livello di singola transazione decidere di tollerare alcune anomalie di esecuzione concorrente.

3.4.6 Gestione della concorrenza in SQL

Tutti i livelli richiedono 2PL stretto per le scritture.

- `serializable`: lo richiede anche per le letture e applica il lock di predicato per evitare l'inserimento fantasma.
- `repeatable read`: applica 2PL stretto per tutti i lock in lettura applicati a livello di tupla e non di tabella. Consente inserimenti e quindi non evita l'anomalia di inserimento fantasma.

- `read committed`: richiede lock condivisi per le letture ma non il 2PL stretto.
- `read uncommitted`: non applica lock in lettura.

Capitolo 4

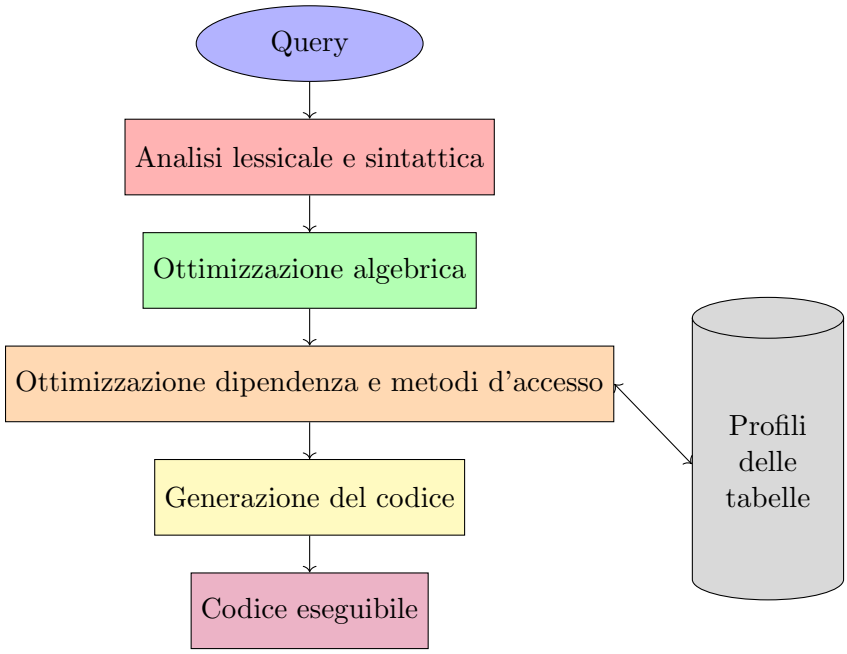
Ottimizzazione di interrogazioni

Ogni interrogazione sottomessa al DBMS viene espressa in un linguaggio dichiarativo, è quindi necessario trovare un equivalente espressione in linguaggio procedurale (*ad esempio in algebra relazionale*) per generare un piano di esecuzione.

L'espressione algebrica va ottimizzata rispetto alle caratteristiche del DBMS a livello fisico e della base di dati corrente.

4.1 Ottimizzazione

L'ottimizzazione avviene secondo i seguenti passi:



L'ottimizzazione algebrica di basa fondamentalmente sulle regole di ottimizzazione già note dell'algebra relazionale:

- Anticipo delle selezioni;
- Anticipo delle proiezioni.

4.2 Ottimizzazione dipendente dai metodi di accesso

Le operazioni tipiche di accesso supportate dai DBMS sono:

- Scansione (*scan*) delle tuple di una relazione.
- Ordinamento di un insieme di tuple.
- Accesso diretto alle tuple attraverso indice.
- Diverse implementazioni del join.

4.2.1 Scansione

Una operazione di `scan` opera contestualmente altre operazioni, tra cui:

- `scan` e `proiezione` senza eliminazione di duplicati.
- `scan` e `selezione` in base ad un predicato semplice.
- `scan` e inserimento, cancellazione o modifica.

Il costo di una scansione sulla relazione `R` è `NP(R)`, dove `NP(R)` è il numero di pagine date dalla relazione `R`.

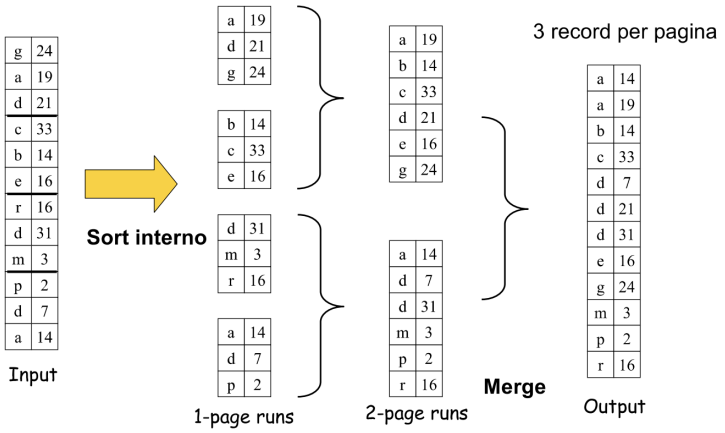
4.2.2 Ordinamento

L'ordinamento viene utilizzato per ordinare il risultato di un'interrogazione (*con la clausola* `order by`), eliminare duplicati (`select distinct`) e raggruppare tuple (`group by`).

L'ordinamento su memoria secondaria viene eseguito tramite l'algoritmo **Z-way Merge Sort** che si suddivide in due fasi:

- Sort interno: si leggono una alla volta le pagine della tabella; le tuple di ogni pagina vengono ordinate facendo uso di un algoritmo di sort interno. Ogni pagina così ordinata, viene detta anche "run", viene scritta su memoria secondaria in un file temporaneo.
- Merge: applicando uno o più passi di fusione, le run vengono unite a produrre un'unica run.

Z-way Sort-Merge



Il costo, considerando solo il numero di accessi a memoria secondaria è

$$2 \cdot \texttt{NP} \cdot (\lceil \log_2 \texttt{NP} \rceil + 1)$$

Dove la moltiplicazione per due considera lettura e scrittura.

4.2.3 Accesso diretto via indice

Le interrogazioni che fanno uso dell'indice:

- Selezioni con condizione atomica di uguaglianza ($A = v$) richiedono indice hash o B+-Tree.
- Selezioni con condizioni di range ($A >= v_1$ `AND` $A <= v_2$) richiedono indice B+-Tree.
- Le selezioni con condizione costruita da una congiunzione di condizioni di uguaglianza ($A >= v_1$ `AND` $B <= v_2$): in questo caso si sceglie per quale delle condizioni di uguaglianza utilizzare l'indice; la scelta ricade sulla condizione più selettiva. L'altra si verifica direttamente sulle pagine dati.
- Le selezioni con condizione da disgiunzione di condizioni di uguaglianza, così composte ($A >= v_1 \quad B <= v_2$): in questo caso è possibile utilizzare più indici in parallelo, facendo un merge dei risultati eliminando i duplicati oppure, se manca anche solo uno degli indici, è necessario eseguire una scansione sequenziale.

4.3 Algoritmi per il join

Il join è l'operazione più gravoso per un DBMS, le implementazioni più diffuse si riconducono ai seguenti operatori fisici:

- Nested Loop Join.

- Merge Scan Join.
- Hash-based Join.

Si noti, benché logicamente il join sia commutativo, dal punto di vista fisico vi sia una chiara distinzione, che influenza anche le prestazioni, tra operando sinistro e operando destro. Per semplicità nel seguito parliamo di relazione esterna e relazione interna per riferirci agli input del join, ma va ricordato che in generale l'input può derivare all'applicazione di altri operatori.

4.3.1 Nested-Loop Join

Date due relazioni in input $\mathcal{R}$ e $\mathcal{S}$ tra cui sono definiti i predicati di join PJ, e supponendo che $\mathcal{R}$ sia la relazione esterna, l'algoritmo opera come segue.

```
per ogni tupla t_R ∈ R{
  per ogni tupla t_S ∈ R{
    if ((t_R, T_S) soddisfa PJ)
      then aggiungi (t_R, T_S) al risultato
  }
}
```

Il costo di esecuzione dipende dallo spazio a disposizione nei buffer. Nel caso base in cui vi sia un buffer per $\mathcal{R}$ e un buffer per $\mathcal{S}$ allora bisogna leggere una volta $\mathcal{R}$ e $\texttt{NR}(\mathcal{R})$ volte $\mathcal{S}$, ovvero tante vole quante sono le tuple della relazione esterna. Per un totale di

$$\texttt{NP}(\mathcal{R}) + \texttt{NR}(\mathcal{R}) \cdot \texttt{NP}(\mathcal{S})$$

Se è possibile allocare $\texttt{NP}(\mathcal{S})$ buffer per la relazione interna il costo si riduce a $\texttt{NP}(\mathcal{R}) + \texttt{NP}(\mathcal{S})$.

Nested-Loop Join con indice B+-Tree

Data una tupla della relazione esterna $\mathcal{R}$, la scansione completa della relazione interna $\mathcal{S}$ può essere sostituita da una scansione basata su un indice costruito sugli attributi di join di $\mathcal{S}$.

Nel caso in cui vi sia un buffer per $\mathcal{R}$ e un buffer per $\mathcal{S}$, bisogna leggere una volta $\mathcal{R}$ e accedere $\texttt{NR}(\mathcal{R})$ volte a $\mathcal{S}$, ovvero tante volte quante sono le tuple della relazione esterna, per un totale di

$$\texttt{NP}(\mathcal{R}) + \texttt{NR}(\mathcal{R}) \cdot \left((\texttt{Prof indice} + \frac{\texttt{NR}(\mathcal{S})}{\text{VAL}(\mathcal{A}, \mathcal{S})} \right)$$

dove: $\texttt{VAL}(\mathcal{A}, \mathcal{S})$ rappresenta il numero di valori distinti dall'attributo di $\mathcal{A}$ che compaiono nella relazione $\mathcal{S}$.

Block Nested-Loop Join

Si tratta di una variante che bufferizza le righe lette della tabella esterna per ridurre il numero di volte che deve essere letta nel ciclo interno.

Il costo di esecuzione dipende dallo spazio a disposizione nei buffer. Nel caso base vi sia un buffer per $\mathcal{R}$ e un buffer per $\mathcal{S}$ allora bisogna leggere una volta $\mathcal{R}$ e $\texttt{NP}(\mathcal{R})$ volte $\mathcal{S}$, ovvero

tante volte quanti sono i blocchi della relazione esterna, per un totale di $\texttt{NP}(\mathcal{R}) + \texttt{NP}(\mathcal{R}) \cdot \texttt{NP}(\mathcal{S})$ accessi a memoria secondaria.

Se è possibile allocare $\texttt{NP}(\mathcal{S})$ buffer per la relazione interna il costo si riduce a $\texttt{NP}(\mathcal{R}) + \texttt{NP}(\mathcal{S})$.

4.3.2 Merge-Scan Join

Il Merge-Scan Join è applicabile quando entrambi gli insiemi di tuple in input sono ordinati sugli attributi di join e il join e un equi-join.

Ciò accade se per entrambe le relazioni di input $\mathcal{R}$ e $\mathcal{S}$ vale almeno una delle seguenti condizioni:

- La relazione è fisicamente ordinata sugli attributi di join (*file sequenziale ordinato come struttura fisica*).
- Esiste un indice sugli attributi di join della tabella che consente una scansione ordinata delle tuple.

La logica dell'algoritmo sfrutta il fatto che entrambi gli input siano ordinati per evitare di fare inutili confronti, il che fa sì che il numero di letture totali sia dell'ordine di $\texttt{NP}(\mathcal{R}) + \texttt{NP}(\mathcal{S})$ se si accede sequenzialmente alle due relazioni ordinate fisicamente.

Con indici B+-Tree su entrambe le relazioni il costo può arrivare al massimo a $\texttt{NR}(\mathcal{R}) + \texttt{NR}(\mathcal{S})$. Con una relazione ordinata e un solo indice al massimo a $\texttt{NP}(\mathcal{R}) + \texttt{NR}(\mathcal{S})$ o viceversa.

4.3.3 Hash-based Join

L'algoritmo di Hash Join, applicabile solo in caso di equi-join, non richiede nè la presenza di indici nè input ordinati, e risulta particolarmente vantaggioso in caso di relazioni molto grandi. L'idea su cui si basa l'Hash Join è semplice: si suppone di avere a disposizione una funzione hash h, che viene applicata agli attributi di join delle due relazioni

- se $t_{\mathcal{R}}$ e $t_{\mathcal{S}}$ sono due tuple di $\mathcal{R}$ e $\mathcal{S}$, allora è possibile che sia $t_{\mathcal{R}}.J = t_{\mathcal{S}}.J$ solo se $h(t_{\mathcal{R}}.J) = h(t_{\mathcal{S}}.J)$.
- Se, viceversa, $t_{\mathcal{R}}.J \neq t_{\mathcal{S}}.J$ allora $h(t_{\mathcal{R}}.J) \neq h(t_{\mathcal{S}}.J)$.

A partire da questa idea si hanno diverse implementazioni. che hanno in comune il fatto che $\mathcal{R}$ e $\mathcal{S}$ vengono partizionate sulla base dei valori della funzione di hash h, e che la ricerca di matching tuples avviene solo tra partizioni relative allo stesso valore di h.

Il costo può essere così valutato:

- costruzione hash-map: $\texttt{NP}(\mathcal{R}) + \texttt{NP}(\mathcal{S})$.
- Accesso alle matching tuples che nel caso pessimo può costare: $\texttt{NP}(\mathcal{R}) \cdot \texttt{NP}(\mathcal{S})$.

Dipende però fortemente dal numero di buffer a disposizione e dalla distribuzione dei valori degli attributi di join.

4.3.4 Scelta finale del piano di esecuzione

Data una espressione ottimizzata in algebra, si generano tutti i possibili piani di esecuzione (*alberi*) alternativi (*o sottoinsieme di questi*) ottenuti considerando le seguenti dimensioni:

- Operatori alternativi applicabili per l'accesso ai dati: ad esempio, `scan` sequenziale o via indice.
- Operatori alternativi applicabili nei nodi: ad esempio, `nested-loop join` o `hash-based join`.
- L'ordine delle operazioni da compiere (*associatività.*)

Si valuta con formule approssimate il costo di ogni alternativa in termini di accessi a memoria secondaria richiesti (*stima*) e si sceglia l'albero con costo approssimato minimo.

Nella valutazione delle stime di costo si tiene conto del profilo delle relazioni, solitamente memorizzato nella Data Directory e contenente per ogni relazione $\mathcal{T}$:

- Stima della cardinalità: $\texttt{CARD}(\mathcal{T})$;
- Stima della dimensione di una tupla: $\texttt{SIZE}(\mathcal{T})$;
- Stima della dimensione di ciascun attributo ($\mathcal{A}$): $\texttt{SIZE}(\mathcal{A}, \mathcal{T})$;
- Stima del numero di valori distinti per ogni attributo ($\mathcal{A}$): $\texttt{VAL}(\mathcal{A}, \mathcal{T})$;
- Stima del valore massimo e minimo per ogni attributo $\mathcal{A}$: $\texttt{MAX}(\mathcal{A}, \mathcal{T})$ e $\texttt{MIN}(\mathcal{A}, \mathcal{T})$;