Lezione 2 11/03/2024

Richiamo sui DBMS centralizzati

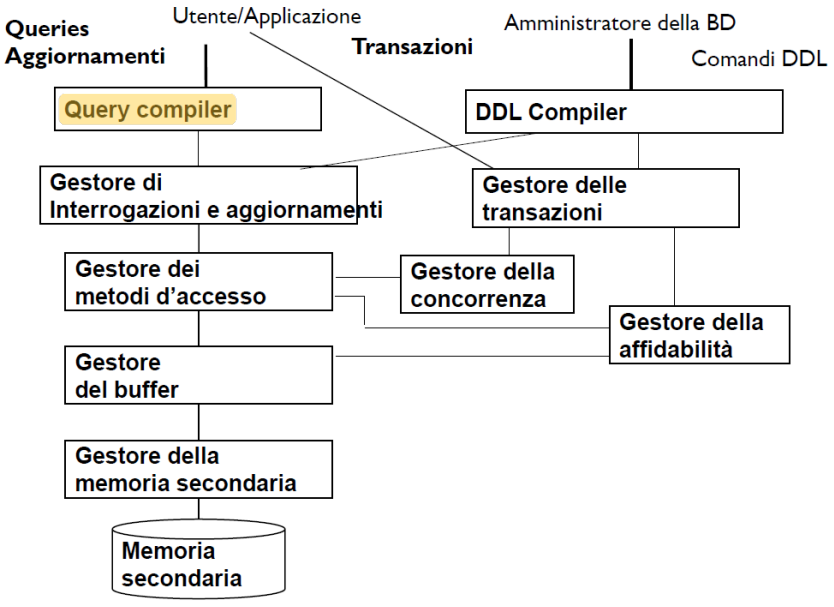
Il DBMS è il software che gestisce i dati. Deve gestire dati:

- **grandi**: molto più grandi di quelli che stanno nella memoria di un pc
- **persistenti**: con un ciclo di vita indipendente dalle singole esecuzioni dei programmi che le utilizzano
- **condivisi**: utilizzate da applicazioni diverse
- **affidabili**: resistenti ai guasti

Hanno un unico schema logico, un'unica base di dati, un unico schema fisico, un unico linguaggio di accesso, un unico sistema di gestione.

I DBMS gestiscono anche il trasferimento delle parti interessate del database in memoria centrale (RAM)

Architettura di un DBMS



Il **compilatore** è quello che deve tradurre un linguaggio semi strutturato in qualcosa di più formale per il pc (compila le query), poi ho bisogno un **gestore** che capisca e ottimizzi l'interrogazione, trasformandola in comandi elementari

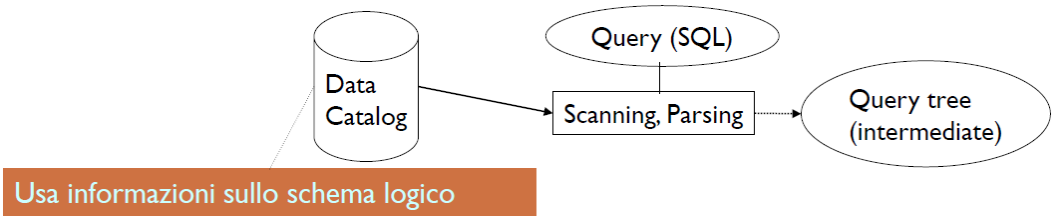
di accesso, passandoli al **gestore dei metodi di accesso** che li trasforma in comandi di accesso alle pagine (deve essere il sistema operativo ad accedere alla memoria), che li invia al **gestore del buffer** che li ottimizza per gestire i buffer, che li invia al **gestore della memoria secondaria** che le traduce in accessi a pagine su disco.

Il **gestore delle transazioni** esegue le transazioni e garantisce, interagendo con il **gestore della affidabilità** e il **gestore della concorrenza,** il rispetto delle proprietà transazionali **ACID (Atomicity, Consistency, Isolation e Durability)**

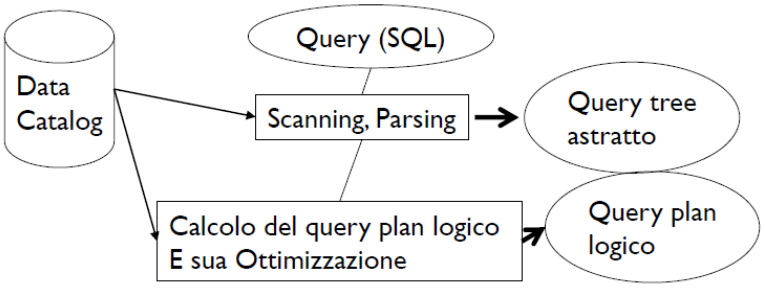
Accesso in lettura di un dato: ottimizzazione delle interrogazioni

Parsing

Il **parser** effettua un'analisi lessicale, sintattica e semantica, usando il dizionario ed effettua la traduzione in algebra relazionale, corrispondente a un query tree.



Una volta che ho una query, la devo ottimizzare, ovvero devo trovare una query che abbia lo stesso risultato ma che prenda in considerazione meno dati.



Quindi il parser trasforma il query tree in un query plan logico contenente operazioni di algebra relazionale sulle tabelle logiche dello schema, e lo ottimizza trasformando le espressioni di algebra relazionale in altre equivalenti ma più efficienti.

Quindi SELECT diminuiscono i dati considerati, la JOIN e il prodotto cartesiano li aumenta.

La query è rappresentata come un albero nel quale le foglie corrispondono alle tabelle e i nodi intermedi rappresentano operazioni algebriche di selezione, proiezione, join, prodotto cartesiano e operazioni insiemistiche.

La JOIN può creare delle tabelle intermedie molto grandi, per poi utilizzare pochi dati. Vediamo un'esempio.

Esempio

- Employee [Ename, lname, ssn (social security number), bdate (birthdate), address]
- Department [Dname, dnumber, mgrssn (manager ssn)]
- Project [Pname, pnumber, plocation, dnum]
- Works-on [Essn (Employee ssn), pno (pnumber), hours]

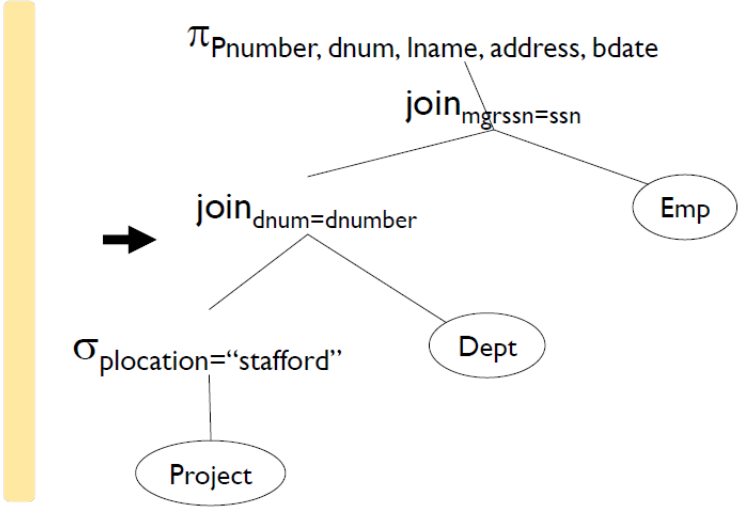
- Trovare Progetti localizzati a Stafford e loro Manager, con mome, indirizzo, e data nascita

```
SELECT  Pnumber, Dnum, Lname, Address, Bdate
 FROM Project P, Dept D, Emp E
Where P.dnum=D.dnumber and E.ssn = D.mgrssn  and P.location = 'Stafford'
```

$$R1 = (\sigma_{plocation="stafford"}(project))\ join_{dnum=dnumber}\ dept$$

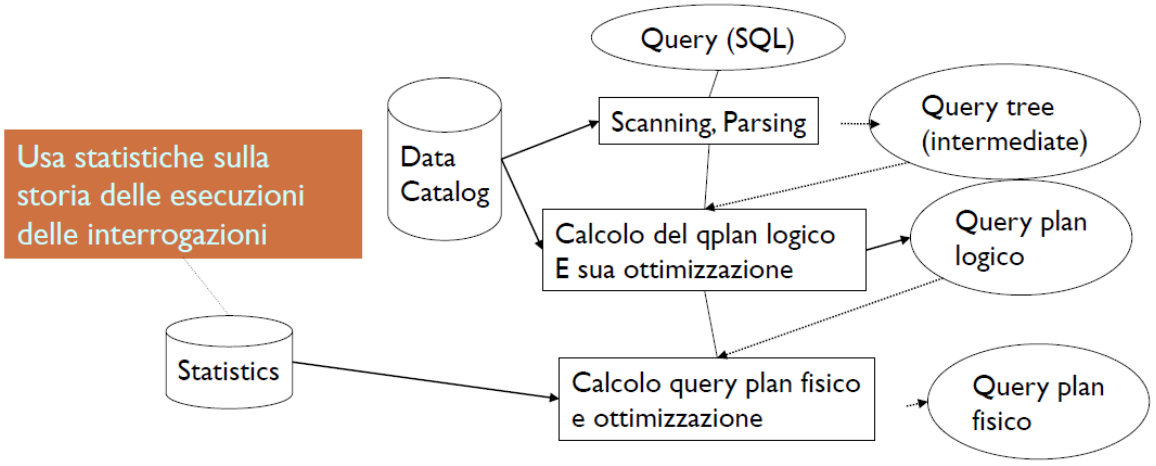
$$R = (\pi_{Pnumber,\ dnum,\ lname,\ address,\ bdate}\ (R1\ join_{mgrssn=ssn}\ emp))$$

Il DBMS fa questa trasformazione:



Parto dalla tabella project, applico una restrizione, poi faccio join con la tabella Dept, etc.

Un buon modo per misurare il tempo, è misurare lo spazio occupato. Quindi la query viene ottimizzata seguendo delle regole, per esempio cambiando l'ordine delle operazioni. In generale cerchiamo di fare i JOIN il più tardi possibile, perché accrescono il numero dai dati.



Infine viene calcolato e ottimizzato il query plan fisico, con i metodi di accesso alla memoria.

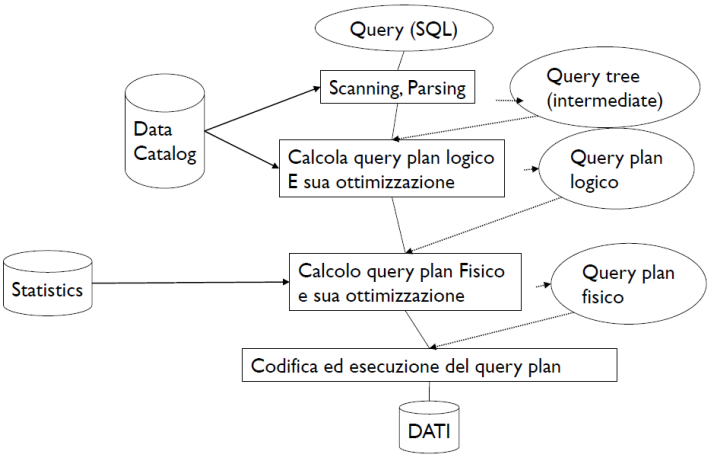
C'è anche un database che tiene traccia delle statistiche, per esempio quanto è grande una tabella.

Queste statistiche sono utili perché ora sappiamo quali tabelle sono più grandi, e possiamo fare la JOIN il più tardi possibile.

La trasformazione e' effettuata mediante una strategia che prevede di utilizzare:

1. Proprietà algebriche
2. Una stima dei costi delle operazioni fondamentali per diversi metodi di accesso

Query plan completo:



In generale, il problema di ottimizzare la query ha complessità esponenziale. In pratica, si introducono delle approssimazioni ragionevoli in base a euristiche.

Accesso in scrittura di un dato: transazioni

La **transazione** è un insieme di operazioni di scrittura e di lettura, che gode di alcune caratteristiche che garantiscono la corretta esecuzione in un ambiente concorrente e non affidabile.

Transazione: parte di programma caratterizzata da un inizio (**begin-transaction**, `start transaction` in SQL), una fine (**end-transaction**, non esplicitata in SQL) e al cui interno deve essere eseguito una e una sola volta uno dei seguenti comandi

- **commit work** per terminare correttamente
- **rollback work** per abortire la transazione

```
start transaction;
update ContoCorrente
   set Saldo = Saldo + 10 where NumConto =
   12202;
update ContoCorrente
  set Saldo = Saldo - 10 where NumConto =
   42177;
select Saldo into A
   from ContoCorrente
   where NumConto = 42177;
if (A>=0)  then commit work
         else rollback work;
```

Esempio di due update, i risultati vengono controllati in una select, se il risultato non è quello atteso allora viene effettuato un rollback. Ci pensa il DBMB a fare queste operazioni.

Commit work se la transazione è avvenuta correttamente

Rollback (abort) e la transazione non deve essere completata

- Il sistema deve ripristinare lo STATO precedente

Proprietà ACID

Atomicità: una transazione è una unità atomica di elaborazione, e non può lasciare la base di dati in uno stato intermedio.

Consistenza: se il db nello stato iniziale è in uno stato corretto, allora dopo la commit il dbms sarà ancora in uno stato consistente.

Isolamento: la transazione viene eseguita come se fosse l'unica in esecuzione in quel momento (es di due prenotazioni contemporanee, la seconda aspetterà il completamento della prima).

La **persistenza** è data dal fatto che i dati non vanno perduti, anche in presenza di guasti.

Anomalie

```
Void chooseSeat() {
 /* codice C che verifica la disponibilita' di posti su un volo
    e prenota un posto */
(1) EXEC SQL SELECT occupied INTO :occ
          FROM flights
          WHERE fltNum = :flight AND fltDate = :date
                                 AND fltSeat = :seat;
(2) if (!occ) {
       EXEC SQL UPDATE Flights
          SET occupied = TRUE
          WHERE fltNum = :flight AND fltDate = :date
                                 AND fltSeat = :seat;
        /* notifica posto prenotato */
 }
 else /* notifica volo completo */
}
```

- Supponiamo che la funzione chooseSeat() venga eseguita simultaneamente da due o piu' applicazioni (es due agenzie viaggi), che cercano di prenotare lo stesso posto sullo stesso volo
- E' possibile un'evoluzione temporale come la seguente:

Appl. 1	Appl. 2	tempo
Esegue (1), trova posto		↓
	Esegue (1), trova posto	
Esegue (2), alloca il posto		
	Esegue (2), alloca il posto	

- A questo punto, abbiamo una doppia prenotazione (LOST UPDATE)
- Il DBMS transazionale gestisce questo problema garantendo la proprieta' di isolamento

Altro esempio:

- Supponiamo che l'app1 termini l'esecuzione della prenotazione, ma non termina con successo la transazione
- Un'altra applicazione accede alla base di dati delle prenotazioni prima che il DBMS riporti lo stato del DB prima dell'esecuzione della app1
- Risultato il posto che è nella realtà libero per il sistema è occupato
  - (dirty read)

Altro esempio:

- Una transazione deve leggere due volte lo stesso dato. Tra una lettura e l'altra una seconda transazione modifica i valori del dato. La prima transazione leggere due dati diversi
  - Letture non ripetibili

Scriviamo la funzione precedente come una transazione:

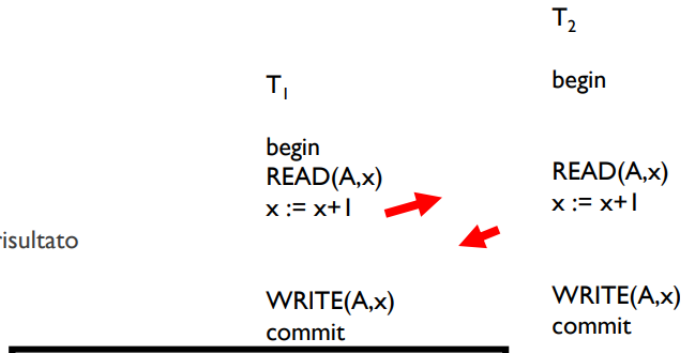
```
Void chooseSeat()  {
Begin transaction
(1)  EXEC SQL SELECT occupied INTO :occ
            FROM flights
            WHERE fltNum = :flight AND fltDate = :date
                                   AND fltSeat = :seat;
(2)  if (!occ)  {
        EXEC SQL UPDATE Flights
            SET occupied = TRUE
            WHERE fltNum = :flight AND fltDate = :date
                                   AND fltSeat = :seat;
         /* notifica posto prenotato */
}
 else { /* notifica volo completo */ }
 commit work;
}
```

La proprietà di isolamento di una transazione garantisce che essa è eseguita come se non ci fosse concorrenza.

Esempio **update loss**:

- Date due transazioni T_1, T_2 identiche:

  T: READ(A, x), x = x + 1, WRITE(A, x)
- L'esecuzione seriale per A=2 produce A=4, risultato di due aggiornamenti successivi



E' l'esempio della prenotazione voli

Il risultato finale e' A=3
Il primo dei due aggiornamenti si e' perso: T2 legge il valore iniziale di A, e scrive il valore finale. In questo caso, e' come se T1 non fosse stato eseguito!

Esempio **lettura sporca**:

- Sono date T1 = T2 come prima:

  T: READ(A, x), x = x + 1, WRITE(A, x)

- Si consideri la seguente schedule (si noti che T1 fallisce):

T_1	T_2
begin	
READ(A,x)	
x := x+1	
WRITE(A,x)	
	begin
	READ(A,x)
	x := x+1
rollback	
	WRITE(A,x)
	commit

Il problema e' che T2 legge un valore dal DB prima che T1 decida per il commit/rollback
Di conseguenza, T2 legge un valore temporaneo "sporco", che viene poi rimosso dall'azione di rollback
Il calcolo di A da parte di T2 si basa quindi su un input errato

Esempio **lettura non ripetibile**:

- T_1 esegue due letture consecutive dello stesso dato:

T_1	T_2
begin	begin
READ(A,x)	
	READ(A,x)
	x := x+1
	WRITE(A,x)
	commit
READ(A,x)	
commit	

- Tuttavia, a causa dell'aggiornamento concorrente da parte di T2, T1 legge due valori consecutivi diversi nel contesto della stessa transazione

Esempio **aggiornamento fantasma**:

- Si assuma il vincolo di integrita' A + B = 1000;

T_1	T_2	
begin		
READ(A,y)	begin	Qui T1 legge un valore di B che e' stato modificato da T2, e che quindi porta ad una violazione del vincolo. Si noti ancora una volta che sia T1 che T2 in situazione di isolamento portano ad un risultato corretto
	READ(A,y)	
	y := y-100	
	READ(B,z)	
	z = z+100	
	WRITE(A,y)	
	WRITE(B,z)	
READ(B,z)	commit	
// qui y+z = 1100		
commit		

Gestore della concorrenza

Schedule

Una sequenza di esecuzione di un insieme di transazioni è detta **schedule**. Uno schedule registra sequenzialmente le operazioni che le varie transazioni devono fare.

> T1: trova posto, T1: alloca posto; T2: trova posto; T2 alloca posto

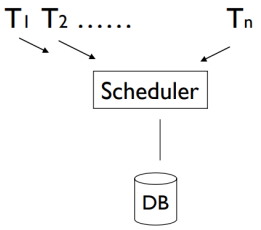
Una schedule è **seriale** se una transazione termina prima che la successiva inizi (quello sopra è seriale). Imporre uno schedule seriale incide però sulle prestazioni. Esempio di schedule non seriale:

> T1: trova posto, T2: trova posto; T1: alloca posto; T2 alloca posto

Serializzabilità

Uno schedule è **serializzabile** se l'esito della sua esecuzione è lo stesso che si avrebbe se le transazioni in esso contenute fossero eseguite in **una** (qualsiasi) sequenza seriale. Questo coincide con la proprietà di isolamento (ogni transazione esegue come se non ci fosse concorrenza).

Si può pensare di avere uno schedule non seriale che ha lo stesso comportamento di uno schedule seriale, ma non è facile. Quindi viene usato un protocollo che costruisce gli schedule di modo tale che siano serializzabili.



Idea: definizione di un protocollo che garantisca a priori la conflict serializability. Questa richiede la presenza di uno scheduler nell'architettura del DBMS.

Tramite il meccanismo di **lock**, prendo in maniera esclusiva (per come lo vediamo ora) una risorsa e poi applico la mia transazione. Ci sarà quindi un lock manager che ha una tabella delle tabelle lockate. Dopo aver terminato la transazione la risorsa viene rilasciata (unlock).

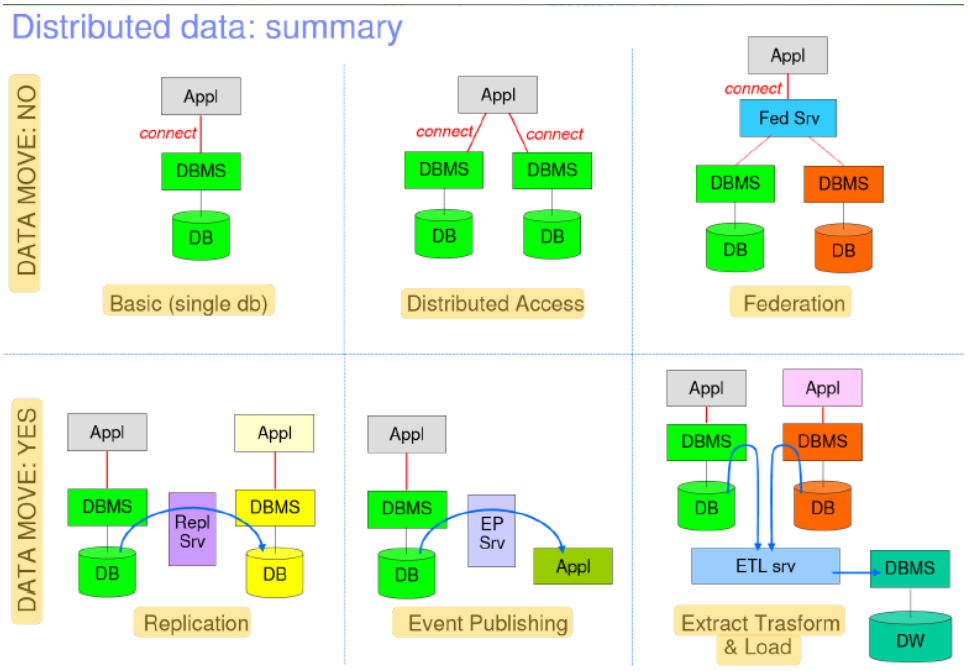
- Lock (esclusivo): $l_i(A)$ → Chiedo la risorsa in modo esclusivo
- Unlock: $u_i(A)$ → Rilascio la risorsa

Quindi lo **scheduler** trasforma le transazioni aggiungendo i comandi di lock, sulla base della conoscenza delle assegnazioni precedenti, in modo da garantire la serializzabilità.

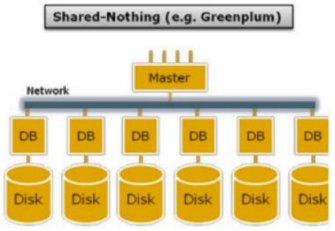
L'algoritmo si chiama **two-phase locking** (lock, transazione, unlock), con la regola: in ogni transazione tutte le richieste di lock precedono tutti gli unlock.

DDBMS

Tipi di DDBMS



Il primo caso è il caso classico che abbiamo visto fin ora.
Nel secondo caso è il livello applicativo che deve sapere dell'esistenza di due DBMS.
Noi ci concentreremo sul terzo caso, dove l'applicazione si collega ad un federation server che gestisce i DBMS.



Noi lavoriamo con l'architettura share nothing dove ciascuna macchina ha il suo DBMS.

Strategie di query processing nel DDBMS

Chi scrive la query non sa che la tabella è distribuita. La query viene tradotta in algebra relazionale. Dopo aver costruito l'albero di accesso ai dati mi devo porre il problema di dove sono i dati. In un sistema distribuito non so dov'è la tabella.

Fasi del query processing

- **Query decomposition**: opera sullo schema logico globale, non considera la distribuzione, usa tecniche di ottimizzazione algebrica come quelle centralizzate, ha come output un query tree non ottimizzato rispetto ai costi di comunicazione.

- **Data localisation**: considera la distribuzione dei frammenti e ottimizza l'operazione in base a questi, ha come output una query che opera in modo efficiente sui frammenti, non ottimizzata.

  - Es. $\sigma_{eno = 'E20'}$ EMP →
  - Supponiamo che EMP sia rappresentata nei nodi 1, 2, e 3.
  - La query puo' essere inizialmente frammentata nella
  - $\sigma_{eno = 'E20'}$ EMP1 $\cup$ $\sigma_{eno = 'E20'}$ EMP2 $\cup$ $\sigma_{eno = 'E20'}$ EMP3
  - Applichiamo una tecnica di riduzione (supponiamo che E20 sia solo in EMP2) → $\sigma_{eno = 'E20'}$ EMP2

- **Ottimizzazione globale**: nel query tree vengono aggiunti agli operatori di algebra relazionale, quelli di comunicazione (send/receive tra nodi). L'obiettivo è quello di trovare l'ordinamento "migliore" delle operazioni definite dalla fragment query utilizzando modelli di costo che tengono conto dei costi di comunicazione.

Esempio

- Schema:
- Employee (eno, ename, title)
- AssiGN(eno, projectno, resp, dur)
- dove resp indica il tipo di responsabilita'
- |EMP| = 400, |ASG| = 1000

- Query: "trovare nomi dei dipendenti che sono manager di progetti"

SQL (trasparenza di frammentazione):

SELECT ename

FROM Employee E JOIN AssiGN A on E.eno=A.eno

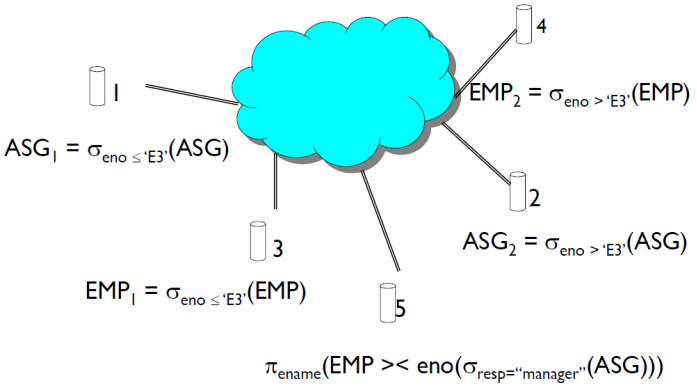
WHERE resp="manager"

AR: π_{ename}(EMP ><$_{eno}$ ($\sigma_{resp="manager"}$(ASG)))

L'ultima riga è la traduzione in algebra relazionale, che parte da ASG e va all'esterno.

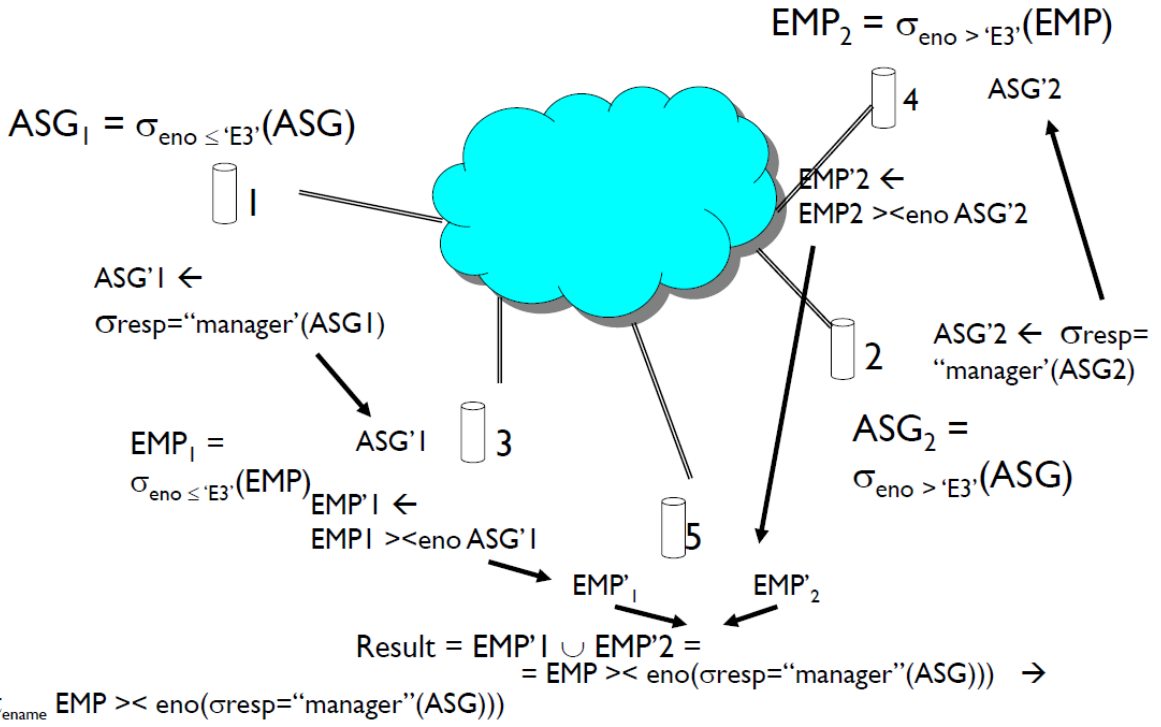
Supponiamo una frammentazione orizzontale di questo tipo:

- ASG1 = σeno ≤ 'E3'(ASG) nodo 1
- ASG2 = σeno > 'E3'(ASG) nodo 2
- EMP1 = σeno ≤ 'E3'(EMP) nodo 3
- EMP2 = σeno > 'E3'(EMP) nodo 4
- Risultato nodo 5

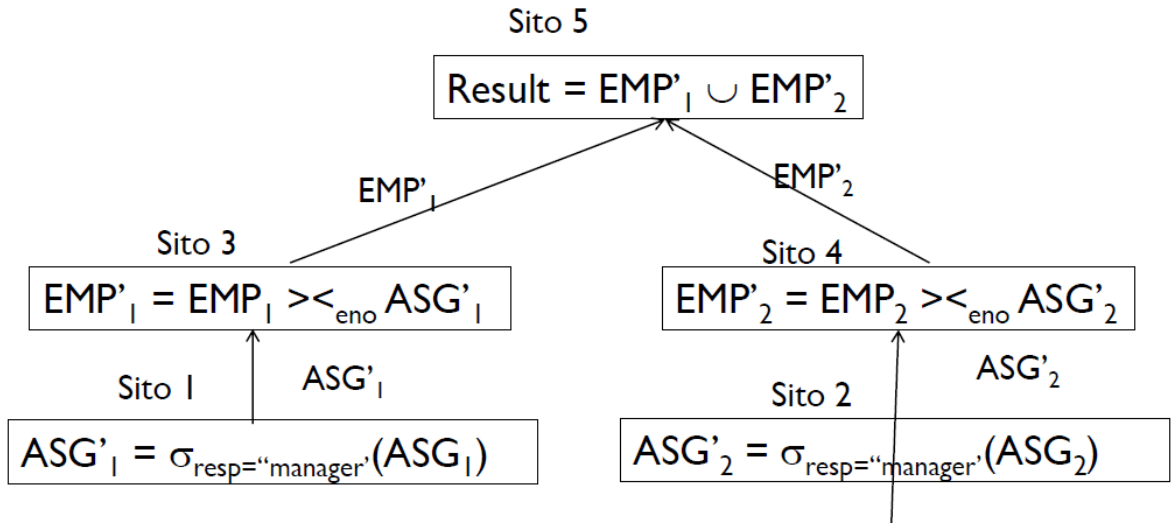


$\pi_{ename}(EMP >< eno(\sigma_{resp="manager"}(ASG)))$

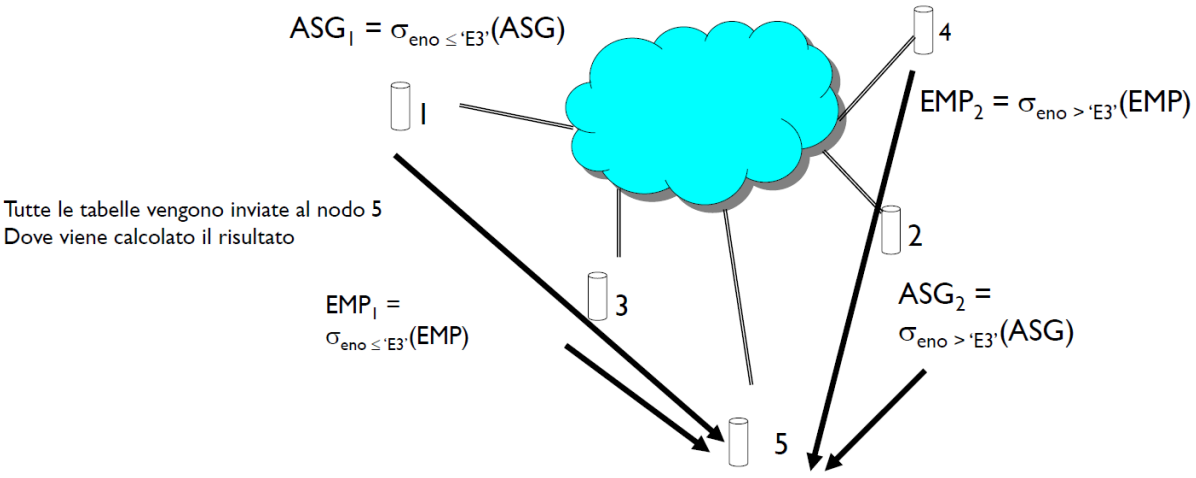
Una possibile soluzione è quella di spostare i dati da 1 a 3 a 5, e in parallelo da 2 a 4 a 5. Il più lento (a livello di rete) determina il tempo di questa query.

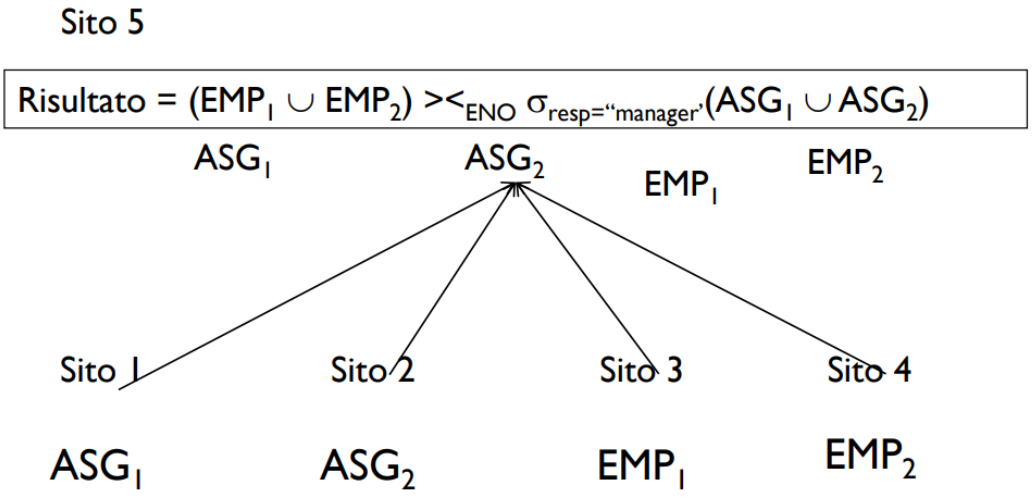


Rappresentazione della strategia come operazioni + trasmissioni:



Oppure si può anche fare:





Sono due strategie accettabili ma diverse. Nel secondo caso sto sperando che i dati siano più piccole rispetto alle operazioni che poi devo fare.

Confronto di costi tra le due strategie:

- Modelli di costo (semplificato):
  - Costo accesso a un record: 1
  - Costo trasferimento di un record: 10
  - $|EMP| = 400, |ASG| = 1000, |\sigma_{resp="manager'}(ASG_1)| = |\sigma_{resp="manager'}(ASG_2)| = 10$
  - Accesso diretto a EMP via ENO, ASG via RESP

Costo strategia A:

1. Calcolo ASG'_1 e ASG'_2 → 10+10 per accesso diretto
2. Trasferimento ASG'_1 e ASG'_2 → 20 x 10 = 200
3. Calcolo EMP'_1 ed EMP'_2: join (single-loop) → (10 + 10) x 2
4. Trasferimento EMP'_1, EMP'_2: 20 x 10 = 200

- Totale: 460 (400 trasferimento e 60 calcolo)

- Modelli di costo (semplificato):
  - Accesso a un record: 1
  - Trasferimento di un record: 10
  - $|EMP| = 400, |ASG| = 1000, |\sigma_{resp="manager'}(ASG_1)| = |\sigma_{resp="manager'}(ASG_2)| = 10$
  - Accesso diretto a EMP via ENO, ASG via RESP

Costo strategia B (non si trasferiscono gli indici!):

1. Trasferimento EMP_1, EMP_2 sul nodo 5: 400 x 10 = 4.000
2. Trasferimento ASG_1, ASG_2 sul nodo 5: 1000 x 10 = 10.000
3. Calcolo ASG' (selezione): 1.000 (no indice)
4. Join ASG', EMP: 20 x 400 = 8.000 (non ricostruisce gli indici → nested loop)

- Totale: 23.000 (14.000 trasferimento e 9.000 calcolo)

Costo

- Costo totale =
  somma dei costi delle operazioni (I/O e CPU) +
  costi di comunicazione (trasmissione)

- Response time = somma dei costi tenendo conto del parallelismo

Rispetto al caso centralizzato, in cui i costi più rilevanti sono quelli di trasferimento dei blocchi da memoria secondaria a principale, consideriamo qui i soli costi di comunicazione e trascuriamo i costi di I/O.

Costo comunicazione = C_{MSG} * #msgs + C_{TR} * #bytes

C_{MSG} = costo fisso di spedizione/ricezione messaggio (setup)

C_{TR} = costo (fisso rispetto alla topologia!) di trasmissione dati

Nel **tempo di risposta**, a differenza del costo di trasmissione, i costi delle operazioni in parallelo non si sommano.

Tempo di risposta (solo comunicazione) =

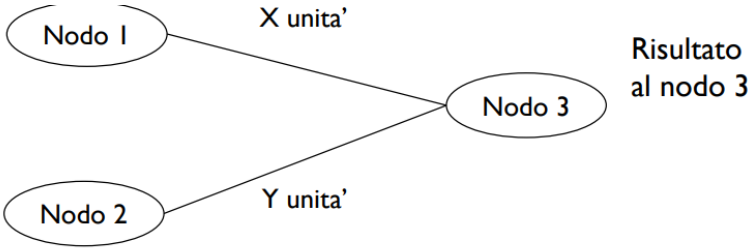
C_{MSG} * seq_#msgs + C_{TR} * seq_#bytes

dove seq_#msgs e' il massimo numero di messaggi che devono essere comunicati in modo sequenziale.

Nelle grandi reti geografiche il **costo di comunicazione** è ben maggiore del costo di I/O (fattore da 1 a 10). Nelle reti locali i due costi sono paragonabili.

Il costo di comunicazione e' ancora il fattore critico, ma si stanno avvicinando. Possiamo, in alternativa a trascurare i costi di I/O, utilizzare pesi nelle formule di costo.

Esempio:



- Costo di trasferimento di x unita' da 1 a 3 e di y unita' da 2 a 3:
- Costo comunicazione = $2\ C_{MSG} + C_{TR} * (x + y)$
- Tempo di risposta = $\max(C_{MSG} + C_{TR} * x, C_{MSG} + C_{TR} * y)$

dato che x e y vengono trasferiti in parallelo

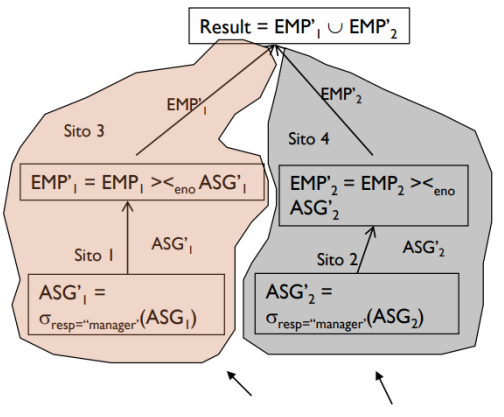
Possiamo essere interessati a:

- **Minimizzazione tempo di risposta**: più parallelismo → può portare ad aumento del costo totale (maggiore numero di trasmissioni e processing locale)
- **Minimizzazione costo totale** = somma dei costi senza tener conto del parallelismo: utilizza meglio le risorse → aumento del throughput (con peggioramento del response time in generale)

NEL NOSTRO ESEMPIO

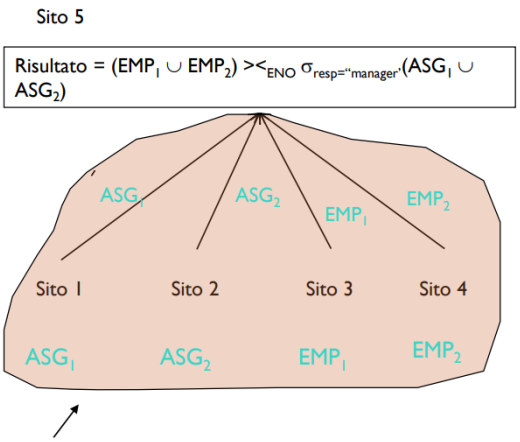
Strategia A

Ottimizza il costo



Strategia B

Parallelizza di piu' ma, in questo caso, non minimizza il tempo di riposta



Operazioni e trasmissioni eseguibili in parallelo

Join e semijoin

In un DBMS distribuito l'operazione di **semijoin** può essere in alcune circostanze un'alternativa più efficiente alla operazione di join.

Definizione: $R \text{ semijoin}_A S \equiv \pi_{R*}(R \text{ join}_A S)$

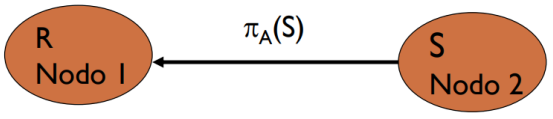
Dove R* e' l'insieme degli gli attributi di R

Il semijoin $R \text{ semijoin}_A S$ percio' e' la proiezione sugli attributi di R della operazione di join

Attenzione: il semijoin e' non comutativo!

Prima osservazione, dalla definizione di semijoin:

il calcolo di ($R \text{ semijoin}_A S$) richiede la presenza del solo attributo S.A da parte di S, cioe' solo di $\pi_A(S)$



Osservazione 1: se R, S sono allocate su nodi diversi, allora R $join_A$ S puo' essere calcolato tramite operazioni di semijoin piuttosto che di join. Valgono infatti le seguenti equivalenze (vedi ad es. [Ozsu-Vald. sec.9.3.2]) tra join e semijoin:

1. R $join_\theta$ S $\Leftrightarrow$ (R $semijoin_\theta$ S) $join_\theta$ S
2. R $join_\theta$ S $\Leftrightarrow$ R $join_\theta$ (S $semijoin_\theta$ R)
3. R $join_\theta$ S $\Leftrightarrow$ (R $semijoin_\theta$ S) $join_\theta$ (S $semijoin_\theta$ R)

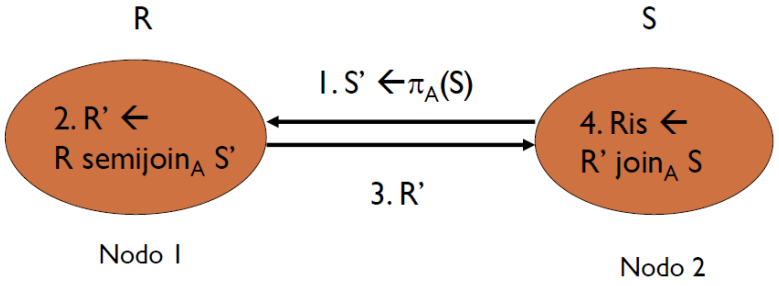
Ognuna da' luogo ad una diversa strategia. La scelta tra le strategie richiede la stima dei loro costi

Esempio semijoin:

R $join_A$ S

$\Leftrightarrow$

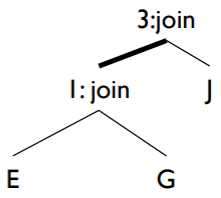
(R $semijoin_A$ S) $join_A$ S

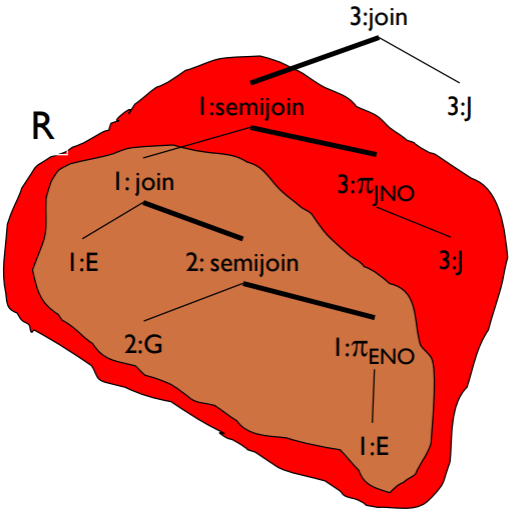


In generale l'uso del semijoin e' conveniente se il costo del suo calcolo e del trasferimento del risultato e' inferiore al costo del trasferimento dell'intera relazione e del costo del join intero.

ESEMPIO CON TRE RELAZIONI

E join G join J



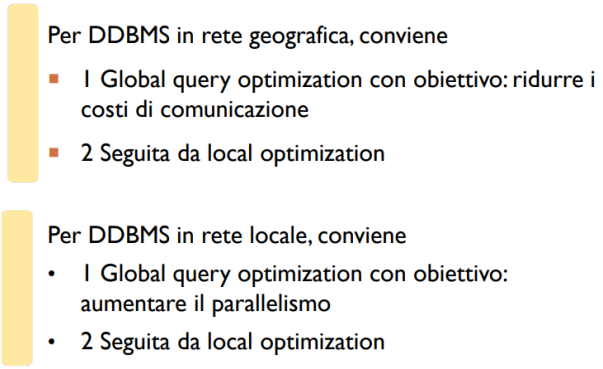


R = G $join_{eno}$ E = (G $semijoin_{eno}$ E) $join_{eno}$ E = $\pi_{G.*}$(G $join_{eno}$ (π_{eno}(E))) $join_{eno}$ E

R $join_{jno}$ J = (R $semijoin_{jno}$ J) $join_{jno}$ J = πR.*(R $join_{jno}$ (π_{jno}(E))) $join_{jno}$ J

Quarta fase del query processor:

- Local query optimization → local schemas: Ogni nodo riceve una fragment query e la ottimizza in modo indipendente. Vengono utilizzate tecniche analoghe a quelli dei sistemi centralizzati



Nella progettazione delle basi di dati distribuite si dovrebbe anche tenere conto di:

- Topologia della rete
- Tipologie di query distribuite
- Stime o statistiche su query distribuite

Controllo di concorrenza

Classificazione delle transizioni

Non possiamo imporre uno schedule seriale perché perderemmo efficienza. Le operazioni importanti sono quelle read/write perché le write causano problemi, le read singole no.

Le transizioni possono essere:

- Dirette ad un **unico** server remoto:
  - **Remote requests**: transazioni read-only con un numero arbitrario di query SQL
  - **Remote transactions**: transazioni read-write con un numero arbitrario di operazioni SQL (select, insert, delete, update)
- Dirette ad un **numero arbitrario** di server:
  - **Distributed requests**: transazioni read only arbitrarie, nelle quali ogni singola operazione SQL si può riferire a qualunque insieme dei server. Questo tipo di transazione richiede un ottimizzatore distribuito.

- **Distributed transactions**: hanno un numero arbitrario di operazioni SQL (select, insert, delete, update), ogni operazione e' diretta ad un unico server. Le transazioni possono modificare più di un DB. Richiede un protocollo transazionale di coordinamento distribuito → **two-phase commit**.

Esempio di transazione distribuita: trasferimento di un importo tra conti bancari

Questo caso funziona tramite two phase locking, usando quindi l'accesso esclusivo (locking) delle risorse. Una volta che la transazione singola ha acquisito tutti lock necessari, fa le operazioni, fa la commit e poi rilascia i lock. Solo dopo il rilascio dei lock le altre transazioni possono partire. Le transazioni che lavorano su altre risorse possono andare avanti. Questo meccanismo è un compromesso.

Base dati: ACCOUNT (AccNum,Name,Total):

Frammentazione

- AccNum < 10000 → frammento ACCOUNT1 sul nodo 1
- AccNum >= 10000 → frammento ACCOUNT2 sul nodo 2

```
begin transaction
update Account1
  set Total = Total - 100000 where AccNum = 3154;
update Account2
  set Total = Total + 100000 where AccNum = 14878;
 comamit work;
end transaction
```

- Nota: deve comunque valere la proprieta' di atomicita' rispetto alle due updates

---

DDBMS e proprietà ACID

La distribuzione non ha conseguenze su consistenza e durabilità:

- Consistenza: non dipende dalla distribuzione, perché i vincoli descrivono solo proprietà logiche dello schema (indipendenti dall'allocazione)
- Durabilità (persistenza): garantita localmente da ogni sistema

Invece, è necessario rivedere alcuni componenti dell'architettura:

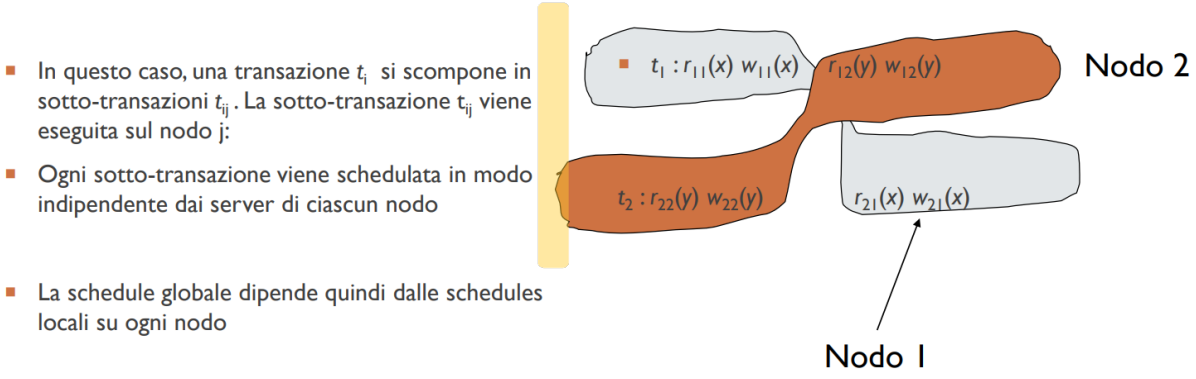
- Concurrency control (Isolamento), bisogna evitare che due scritture quasi concorrenti causino problemi.
- Reliability control, recovery manager (Atomicità), bisogna garantire che una transazione o va a buon fino o non esiste.

Concurrency control

Una transazione non lavora più su un singolo nodo, può capitare che un pezzo sia eseguito sul nodo 1, e un pezzo sul nodo 2. Localmente ciascun nodo non saprà cosa accade sugli altri nodi.

In questo caso, ogni singola sotto-transazione (eseguite localmente sul nodo) viene schedulata secondo le logiche del DBMS centralizzato, quindi quando sul nodo 2 arriva un pezzo della transazione, questa va a vedere se la risorsa y è disponibile, ne ottiene il lock e effettua l'operazione.

Sul nodo 1 la transazione inizia, chiede l'accesso alla risorsa x, viene lockata, e non può essere rilasciata finchè la transazione 1 (anche sul nodo 2) finisce. Questo può provocare situazioni complicate, per esempio quando la transazione 1 prende il lock della risorsa x sul nodo 1, la transazione 2 parte dal nodo 2 e cerca di leggere la risorsa y e viene lockata, la transazione poi va sul nodo 2 e chiede la risorsa y ma è lockata. ... (mi sono perso il resto, è quello nell'immagine sotto, con la transazione 1 e 2)

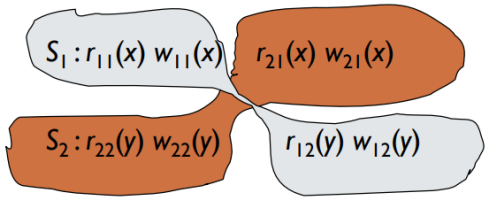


Questo accade perché il nodo 1 non sa cosa succede sul nodo 2.

Serializzabilità globale e repliche

Devo in qualche modo avere una visione globale.

- Osservazione: la serializzabilita' locale di ogni schedule non garantisce la sua serializzabilita' globale. Esempio S1 al nodo 1, S2 al 2:



TRANSAZIONE T2

TRANSAZIONE T1

$t_1 \rightleftharpoons t_2$

- X allocato al nodo 1, Y allocato al nodo 2
- S_1 e S_2 sono schedules locali, ciascuna e' seriale
- Il grafo globale dei conflitti ha un ciclo:
  - Sul nodo 1, t_1 precede t_2 ed e' in conflitto con t_2
  - Sul nodo 2, t_2 precede t_1 ed e' in conflitto con t_1

Secondo problema: quando ho delle repliche, dove leggo? dove scrivo?

Devo essere certo che ogni scrittura fatta su un nodo viene fatta anche sugli altri. Qui il fattore tempo diventa fondamentale, perché se io andassi a leggere nel nodo 2 dopo che il nodo 1 ha fatto scrittura, il dato nel nodo 2 è vecchio.

Senza Repliche

- Se il DB non e' replicato e ogni schedule locale e' serializzabile, allora
- la schedule globale e' serializzabile se gli ordini di serializzazione sono gli stessi per tutti i nodi coinvolti

Con repliche

- Semplice esempio di problema di serializzazione:
- DB su due nodi, X replicata nei due nodi, chiamiamo x1 e x2 le due copie.
- Due transazioni:
- T1: Read (x) x <- x + 5 Write (x) Commit
  T2: Read (x) x <- x * 10 Write (x) Commit
- Che possiamo trascrivere in
  T1: r1(x) w1(x)
  T2: r2(x) w2(x)

Consideriamo le seguenti due schedule che possono essere generate ai due nodi 1 e 2:

S1 al nodo 1 : r11(x) w11(x), r21(x) w21(x)

S2 al nodo 2 : r22(x) w22(x) r12(x) w12(x)

Sia S1 che S2 sono seriali, ma serializzano T1 e T2 in ordine opposto, come nell'esempio precedente

In questo caso, viene violata la *mutua consistenza* dei due DB locali, per la quale tutte le copie devono avere lo stesso valore al termine della transazione

Se inizialmente x1 = x2 = 1, alla fine: x1 = 60, x2 = 15

Abbiamo bisogno di un protocollo di controllo delle repliche

Protocollo ROWA di controllo delle repliche

Per essere sicuro che le repliche siano sempre aggiornate si usa ROWA. Solo quando tutte le repliche comunicano di aver scritto in maniera persistente il dato sul sistema si può andare avanti. Questo garantisce che posso leggere

qualunque nodo senza preoccuparmi che la versione di un dato su un particolare nodo sia l'ultima, perché siamo sicuri che lo è.

In alcuni contesti questa scelta (ROWA) è fondamentale, perché voglio privilegiare il fatto che le repliche siano sempre corrette.

- Read Once Write All (ROWA)
- Dato un item X con copie $X_1,\ldots X_n$:
- X e' detto item logico, $X_1,\ldots X_n$ sono items fisici
- Le transazioni "vedono" solo X
- Il protocollo di controllo mappa:
  - Read(X) su una (qualunque) delle copie
  - Write(X) su tutte le copie
- Questa condizione puo' essere rilassata con protocolli asincroni, piu' efficienti, ma non ce ne occupiamo
- E' implementato nelle estensioni a 2PL – prossimo argomento

2 PHASE LOCKING

Come posso estendere l'algoritmo 2PL al caso distribuito?

- L'algoritmo 2PL si estende al caso distribuito
- Due strategie:
  - Centralized (or Primary Site) 2PL [Ozsu-Vald 11.3.1] basato sui siti
  - Primary copy 2PL [Ozsu-Vald 11.3.2] basato sulle copie

two phase locking: