

Transazioni

Transazione : parte di un programma caratterizzata da un inizio (**begin**) e da una fine (**end**), al cui interno devo eseguire una sola volta una delle due azioni (**commit,abort**).

ACID : Atomicità, Consistenza, Isolamento, Durata

Atomicità: la transazione è un unità atomica di operazioni. Se avvengono errori, devo garantire che le azioni vengano effettivamente svolte (REDO) oppure rimosse (UNDO) a seconda che si sia giunti o meno al commit. (Abort-Rollback-Restart, protocolli di commit)

Consistenza: la transazione rispetta i vincoli di integrità. Se stato iniziale è corretto, anche lo stato finale lo deve essere. (Verifiche di integrità DBMS)

Isolamento: in un contesto di più transazioni concorrenti in esecuzione nello stesso istante, l'azione di una transazione non deve interferire con quelle delle altre e viceversa (NO esposizione stati intermedi) (Concurrent control)

Durata: gli effetti della transazione giunta a commit sono "eterni" (Recovery Manager)

Controllo di concorrenza

Transazioni concorrenti necessarie per garantire il **throughput** di dati richiesto dalle moderne applicazioni. Impensabile usare esecuzione seriale. Bisogna gestire le problematiche dovute all'esecuzione concorrente (transazioni interleaved e nested).

1. **Lost update**: gli effetti di t2 sono sovrascritti da quelli di t1 (t1 scrive su un dato x su cui aveva scritto in precedenza t2);
2. **Dirty read**: la "read" di t2 legge uno stato intermedio del dato x generato dalle azioni di t1 (che poi va in abort);
3. **Read inconsistente**: successive letture di x da parte di t1 leggono valori differenti nell'ambito della stessa transazione (perchè t2 scrive su x);
4. **Ghost update**: t1 osserva solo una parte degli effetti di t2, ovvero uno stato che non soddisfa i vincoli di integrità
5. **Ghost insert**: t2 inserisce una tupla che rientra in un insieme di tuple valutate più volte da t1 con una condizione.

Si introduce il concetto di **Schedulazione**: uno schedule S1 rappresenta la sequenza di esecuzione delle operazioni di lettura/scrittura effettuate dalle transazioni concorrenti.

Es. S1 = r1(x)r2(z)w1(x)w2(z)

Lo scheduler è il componente che effettua il controllo di concorrenza. Il suo obiettivo è rifiutare le schedulazioni che generano anomalie. Nella schedulazione seriale le azioni delle transazioni appaiono in sequenze contigue.

Es. S2 = r1(x)r1(y)w1(x)r2(y)r2(z)w2(z)r3(x)w3(y)w3(z)

Una schedulazione serializzabile produce lo stesso output che produrrebbe una qualunque schedulazione seriale delle operazioni. Si introduce la nozione di equivalenza degli schedule

e si assume in prima approssimazione che nello schedule compaiano solo le azioni dovute a transazioni giunte a commit (e non a quelle abortite) (**commit-proiezione**). In realtà lo scheduler reale deve operare non conoscendo a priori l'esito di ogni transazione.

Relazioni lettura-scrittura: in uno schedule S:

1. **ri(x) legge da wj(x)** quando wj(x) precede ri(x) in S e non ho wk(x) in S tra di loro;
2. **wi(x) è una scrittura finale** se è l'ultima scrittura di x che compare in S;

=> Due schedule S1 ed S2 sono **equivalenti** ($S1 \approx S2$) se possiedono le medesime relazioni "legge da" e "scrittura finale"

=> Uno schedule S è **view-serializzabile** se è equivalente ad un qualunque schedule seriale (in tal caso appartiene alla classe **VSR**)

La decidibilità della view-serializzabilità di uno schedule generico è un problema NP-Completo. Si pone una nuova definizione:

1. L'azione ai è **in conflitto** con l'azione aj ($i \neq j$) se entrambe operano sullo stesso oggetto e almeno una delle due è una scrittura (wr,rw,ww)

=> Due schedule S1 ed S2 sono **conflict-equivalenti** se contengono le stesse operazioni e tutte le coppie in conflitto sono nello stesso ordine.

=> Uno schedule S è **conflict-serializzabile** se esiste uno schedule seriale ad esso conflict equivalente (in tal caso appartiene alla classe **CSR**)

SR incluso in CSR incluso in VSR

Per determinare se lo schedule S appartiene a CSR: analisi del **grafo dei conflitti**. Uno schedule è in CSR sse il suo grafo dei conflitti è aciclico. Infatti, se il grafo è aciclico, allora sarà ordinato topologicamente. Per ogni conflitto i , j avrò sempre che $i < j$

In realtà la tecnica non è efficiente perchè lo scheduler agisce "incrementalmente" ad ogni richiesta, e sarebbe molto oneroso valutare l'aciclicità del grafo ad ogni operazione. Si utilizza allora il **locking**.

Lock Manager

Definiamo una transazione **ben formata rispetto al locking** se:

1. Le operazioni di lettura sono precedute da una richiesta (**r_lock**) e seguite da **unlock**
2. Le operazioni di scrittura sono precedute da una richiesta (**w_lock**) e seguite da **unlock**
3. Una transazione che prima legge e poi scrive in un oggetto può innalzare il lock;

Il **lock manager** ha una tabella in cui mantiene lo stato delle risorse del DBMS. Ad ogni operazione, esamina la richiesta di lock e lo concede/rifiuta a seconda delle condizioni e della **Tabella dei conflitti**:

	<i>FREE</i>	<i>R_LOCKED</i>	<i>W_LOCKED</i>
R_LOCK	OK / R_LOCKED	OK / R_LOCKED	NO / W_LOCKED
W_LOCK	OK / W_LOCKED	NO / R_LOCKED *	NO / W_LOCKED
UNLOCK	ERROR	OK / dipende **	OK / FREE

(*) se la richiesta è un innalzamento di lock da parte della transazione che detiene l'R_LOCK, allora viene accettata e la risorsa va in stato W_LOCKED

(**) se ci sono più di una transazione a detenere l' R_LOCK , allora lo stato è R_LOCK altrimenti è FREE

Per la serializzabilità bisogna garantire che il locking sia a 2 fasi (2PL)

1. Una transazione non può acquisire ulteriori lock una volta che ne ha rilasciato uno
2. I lock possono essere rilasciati solamente dopo le operazioni di COMMIT/ABORT (*)

Uno schedatore che usa transazioni ben formate rispetto al lock, che concede le risorse in base alla tabella dei conflitti e che usa la strategia a 2 fasi, produce schedule appartenenti alla **classe 2PL**, che sono serializzabili (infatti 2PL implica CSR).

(*) Per rimuovere l'ipotesi di commit-proiezione, si passa al **2PL strict**, ovvero si garantisce che i lock vengano rilasciati solo dopo le operazioni di commit e abort.

Metodo del Timestamp

E'una tecnica alternativa al 2PL. Si definisce un timestamp che impone un ordinamento temporale per tutti gli eventi del sistema.

1. Ogni transazione ha associato un **timestamp** generato al momento della sua creazione
2. Uno schedule è accettato solo se riflette l'ordinamento seriale indotto dai timestamp
3. Ogni risorsa x ha 2 contatori **RTM(x)** e **WTM(x)**, che indicano rispettivamente il tempo dell'ultimo accesso in lettura e di quello in scrittura

=> Quando un'azione associata ad una transazione vuole accedere ad un dato, si confronta il timestamp della transazione con il valore dei contatori del dato. Se l'accesso è inconsistente, la transazione viene abortita (e fatta ripartire), altrimenti si svolge l'operazione.

Es. **read(x,i_ts)** la transazione i richiede l'accesso in lettura al dato x.
 if(WTM(x) > i_ts) then kill(i)
 else RTM(x) = max (i-ts,RTM(x))

write(x,i_ts) la transazione i richiede l'accesso in scrittura al dato x
if ((WTM(x) > i_ts) or (RTM(x) > i_ts)) then kill(i)
else WTM(x) = i_ts

Questo approccio genera l'uccisione di molte transazioni, inoltre bisogna bufferizzare le scritture fino al commit (si genera attesa).

2PL vs TIMESTAMP

- Gli schedule **2PL** non sono equivalenti agli schedule **TS** (anche se l'intersezione tra le due classi non è vuota).
- **2PL** pone transazioni in attesa, **TS** killa le transazioni
- **2PL**: ordine imposto dai **conflitti**, **TS**: dalla **marcatore temporale**
- Attesa del commit: **2PL strict** , attesa in **TS**
- **2PL** può dar luogo a deadlock, **TS** a starvation
- restart di una transazione (**TS**) è più oneroso che farla attendere (**2PL**)

Si può modificare TS affinché generi più copie dello stesso dato ad ogni scrittura: in questo modo le transazioni in lettura possono accedere alla versione del dato "che gli serve". La copia è mantenuta finché esistono transazioni che possono volervi accedere.

`read(x,i_ts)` è sempre accettata: si legge la versione utile.

`write(x,i_ts)` se $i_ts < RTM(X)$ la richiesta è respinta, altrimenti si crea una nuova versione del dato con $WTMn(x) = i_ts$

Lock gerarchico

E'possibile stabilire un **locking gerarchico**, in modo da avere un controllo a granularità più o meno fine a seconda dell'applicazione da realizzare. In questo modello, si ha un albero gerarchico che identifica l'intero DBMS e in cui ogni nodo rappresenta una risorsa del database. Ad ogni livello, la tipologia di risorsa è via via più definita (file, pagina, relazione, frammento, tupla, valore). Questo permette di aumentare la concorrenza andando a gestire lock più mirati (senza esagerare per non generare un carico eccessivo sul lock manager).

Si hanno 5 modalità di lock:

1. **XL**: exclusive lock (come il WL)
2. **SL**: shared lock (come il RL)
3. **ISL**: intention shared lock (intenzione di SL su un discendente del nodo corrente)
4. **IXL**: intention exclusive lock (intenzione di WL su un discendente del nodo corrente)
5. **SIXL**: shared intentional exclusive lock (blocco il nodo corrente in modo condiviso, intenzione di lock esclusivo su un discendente)

- Le richieste di lock sono esercitate partendo dal nodo radice e scendendo nella gerarchia, gli unlock vengono rilasciati dai livelli più bassi e risalendo lungo la gerarchia;
- Per richiedere **SL** o **ISL** su un nodo non radice una transazione deve avere **ISL** o **IXL** sul genitore;
- Per richiedere **XL**, **IXL** o **SIXL** su un nodo non radice, una transazione deve avere **SIXL** o **IXL** sul genitore;

<i>Req \ State</i>	<i>ISL</i>	<i>IXL</i>	<i>SL</i>	<i>SIXL</i>	<i>XL</i>
ISL	OK	OK	OK	OK	NO
IXL	OK	OK	NO	NO	NO
SL	OK	NO	OK	NO	NO
SIXL	OK	NO	NO	NO	NO
XL	NO	NO	NO	NO	NO

Livello di isolamento SQL 1999

Le write sono sempre effettuate in 2PL strict. Per le read ho vari livelli di isolamento associabili ad ogni transazione, a seconda della criticità della transazione:

1. **read uncommitted**: no lock in lettura, non rispetta lock altrui
 2. **read committed**: lock in lettura e rispetto di lock altrui, ma non è 2PL (evita dirty)
 3. **repeatable read**: 2PL in lettura e lock sui dati (evita dirty, inconsistent, ghost)
 4. **serializable read**: 2PL e lock di predicato (evita tutte le anomalie)
- Il lock di predicato avviene sull'intera relazione (worst case) o sull'indice (best case)

Deadlock

Si verifica quando transazioni concorrenti trattengono risorse e a loro volta richiedono risorse bloccate da altre transazioni. La probabilità di incorrere in un deadlock crescono linearmente con l'aumentare delle transazioni e con il quadrato del numero di lock richiesti da ciascuna transazione.

Un **deadlock** è rappresentato come un **ciclo** nel grafo di attesa delle risorse.

Ci sono diverse tecniche atte a risolvere le condizioni di deadlock:

1. **Timeout**: una transazione che attende oltre un certo tempo viene abortita e fatta ripartire (criticità: scelta del valore di timeout)
2. **Individuazione**: algoritmi che cercano cicli nel grafo delle attese, controllando periodicamente la tabella dei lock.
3. **Prevenzione**: uccisione delle transazioni che potrebbero generare cicli (scelta preemptive e non-preemptive). Attenzione alla starvation delle singole transazioni (politica di uccisione delle transazioni che hanno lavorato meno, si impedisce di uccidere ripetutamente la stessa transazione)

Controllo di affidabilità

La memoria centrale non è persistente, la memoria di massa è persistente ma può danneggiarsi. Si ricorre ad un'astrazione, **memoria stabile** (che non si danneggia mai). Questa astrazione può essere realizzata introducendo elementi di ridondanza che riducano la probabilità di guasto ad un livello arbitrariamente prossimo allo zero.

Per garantire l'efficienza, non si fa operare la base di dati direttamente su memoria di massa, ma si usa un **buffer in memoria centrale** sul quale vengono eseguite le operazioni: la scrittura effettiva in memoria di massa avviene in modalità differita (sistema di caching)

Buffer:

Il buffer è una zona di memoria centrale suddivisa in **pagine** contenenti i dati in uso alle transazioni del DBMS. Ogni pagina ha associato un contatore che indica il numero di transazioni che la stanno utilizzando, e un bit dirty che indica se i dati della pagina sono stati modificati da qualche transazione rispetto a quando sono stati caricati nel buffer:

Primitive del buffer

1. **fix**: richiesta di utilizzo di una determinata pagina dati;
2. **setdirty**: set del bit dirty della pagina;
3. **unfix**: notifica di fine utilizzo di una determinata pagina dati;
4. **force**: scrittura sincrona in memoria di massa di una pagina del buffer;
5. **flush**: scrittura asincrona in memoria di massa di una pagina del buffer;

La primitiva **fix** cerca la pagina richiesta tra quelle già presenti. Se non è presente, cerca di allocare una pagina libera (cercando quelle con contatore pari a 0, e flushando la pagina in memoria di massa se il bit Dirty è settato). Se non ci sono pagine libere ho un'alternativa in base alla politica in esecuzione nel buffer:

- **Steal**: si sottrae una pagina ad un'altra transazione. La pagina **victim** viene flushata in memoria di massa
- **No steal**: la transazione viene sospesa e messa in una coda di attesa.

Quando si verifica un **fix** con successo, si incrementa il contatore della pagina di un'unità. Il contatore viene decrementato ogni volta che viene eseguito un **unfix** su quella stessa pagina.

Vi sono ulteriori politiche di gestione:

- **Politica Force**: le pagine vengono scritte in memoria di massa all'esecuzione del commit (in opposizione alla politica **No Force**)
- **Pre Fetching**: si anticipa la lettura di pagine sequenzialmente (località spaziale dei dati)
- **Pre Flushing**: si anticipa la scrittura delle pagine deallocate (velocizza la **fix**)

Log

Si sviluppa un metodo per garantire le proprietà ACID delle transazioni, in particolare **atomicità** e **durata**. Infatti in caso di errore prima del commit o di abort, bisogna effettuare l'**undo** delle operazioni, in caso di errore dopo il commit bisogna fare il **redo** delle azioni.

Il **file di log** registra in memoria stabile le azioni di una transazione sotto forma di transizioni di stato delle risorse.

In un file di log vengono inserite le informazioni relative ad una risorsa, ricordando lo stato prima e dopo l'azione (before state e after state).

Se avviene un **rollback** oppure un **guasto prima del commit**: UNDO T1 : 0 = BF(0)

Se avviene un **guasto dopo il commit**: REDO T1 : 0 = AF(0)

Vale il principio di **idempotenza** delle azioni undo/redo: ripetere più volte un'operazione di undo/redo è come compierla una volta sola.

Entries nel LOG:

- **begin,commit,abort**: record relativi ai comandi transazionali
- **insert,delete,update**: record relativi alle operazioni di modifica delle transazioni
- **dump, checkpoint**: record relativi alle azioni di recovery

L'operazione di **checkpoint** sospende l'accettazione di operazioni di scrittura,commit e abort, scrive in memoria di massa (**force**) le pagine dirty del buffer relative a transazioni giunte a commit, si scrive in log un record di checkpoint contenente gli identificatori delle transazioni attive in quell'istante, poi si riprendono le operazioni normalmente.

L'operazione di **dump** rappresenta una copia completa del database in backup, nel log viene segnata la presenza di questa copia con un record di dump.

Regole di gestione del log:

- **Write-ahead-log**: scrivo il log (BF) prima di modificare il db (consente **undo**)
- **Commit-rule**: si scrive il log (AF) prima del commit (consente **redo**)

Se scrivo nel DB solo dopo il commit, non dovrò effettuare scritture per gestire l'abort.

Il database viene gestito secondo il **modello Fail-Stop**: quando il sistema rileva un guasto, forza l'arresto delle transazioni ed il ripristino del sistema operativo (**boot**), quindi viene attivata una procedura di **ripresa** (**warm** o **cold** a seconda del tipo di guasto) che riporta il database in uno stato di funzionamento normale, oppure nello stato di stop se il ripristino fallisce.

Warm restart: tipicamente guasto software, abbiamo perso il contenuto del buffer

1. Si legge il log a partire dall'ultimo checkpoint;
2. Si separano le transazioni attive in 2 insiemi, **undo** e **redo**;
3. Si eseguono le relative azioni;

Cold restart: tipicamente guasto hardware, abbiamo perso i dati del dbms.

1. Si ripristinano i dati a partire dall'ultimo dump;

2. Si rifanno (**redo**) le azioni fino all'istante del guasto;
3. Si attiva una procedura di warm restart;

Basi di dati distribuite

C'è necessità di rendere interoperabili le applicazioni, di riflettere la natura distribuita delle organizzazioni e di sfruttare le maggiori disponibilità di capacità computazionale a prezzi sempre più ridotti.

Tipicamente si classificano sistemi distribuiti rispetto all'ambito di distribuzione (LANo WAN) e alla tipologia dei dbms coinvolti (**omogenei** o **eterogenei**).

Problemi:

1. **Autonomia:** localizzare le risorse laddove vengono effettivamente usate (no centri di calcolo centralizzati). Esigenza di cooperazione: alcune applicazioni sono intrinsecamente distribuite.
2. **Trasparenza:** si esprimono vari livelli di trasparenza per mascherare il più possibile la natura distribuita della base di dati, e permetterne un utilizzo “come se fosse” centralizzata.
3. **Efficienza:** si cerca di ottimizzare la modalità d'esecuzione delle query (seriali o parallele)
4. **Affidabilità:** si introducono protocolli in grado di garantire atomicità e isolamento a livello distribuito (durata e consistenza sono infatti proprietà locali da garantire indipendentemente a livello di singolo dbms)

Frammentazione dei dati: si scompongono le tabelle per garantirne la distribuzione. Deve avvenire garantendo completezza e ricostruibilità. Si possono distinguere **frammentazione orizzontale** (insemi di tuple) e **verticale** (insiemi di attributi).

Frammentando la base di dati posso localizzare i frammenti laddove vengono effettivamente utilizzati per migliorare l'efficienza nell'accesso. Tuttavia posso sempre ricostruire la base di dati nella sua interezza con operazioni di **unione** (per i frammenti orizzontali) e **join su chiave** (per i frammenti verticali).

Riguardo alla trasparenza ci sono vari livelli su cui operare (nel senso di modi di scrivere l'applicazione che accede alla base di dati):

1. **Frammentazione:** posso scrivere le mie query come se la base di dati fosse non-frammentata.
2. **Allocazione:** chi scrive la query conosce quali frammenti sono presenti, ma non si preoccupa della loro effettiva allocazione.
3. **Linguaggio:** chi scrive la query deve indicare i frammenti e le allocazioni cui vuole accedere, però il linguaggio utilizzato è unico e maschera la presenza di dbms eterogenei.
4. **Assenza di trasparenza:** non vi è neanche uno standard di inter-operabilità comune. Si usano “dialetti” per ogni DBMS.

Classificazione delle transazioni distribuite

1. **remote request:** transazioni read-only indirizzate a un solo DBMS remoto
2. **remote transaction:** transazioni indirizzate ad un solo DBMS remoto
3. **distributed transaction:** transazioni rivolte ad un numero arbitrario di DBMS ma in cui ogni comando SQL si riferisce a dati memorizzati in un unico DBMS
4. **distributed request:** transazioni distribuite generiche operanti su un numero arbitrario di nodi, ogni comando SQL può accedere anche a vari nodi.

Proprietà ACID per transazioni distribuite

Le transazioni distribuite vengono scomposte in **sotto-transazioni** agenti ognuna su un particolare dbms facente parte dell'architettura distribuita. Bisogna garantire le proprietà ACID anche per le transazioni distribuite.

1. **Atomicità:** principale problema delle transazioni distribuite;
2. **Consistenza:** se si preserva l'integrità locale di ogni sottotransazione, i dati saranno globalmente consistenti;
3. **Isolamento:** se ogni sottotransazione è 2PL, la transazione è globalmente serializzabile. Per gli scheduler TS si usa il metodo di Lamport.
4. **Durata:** se ogni sottotransazione è gestita correttamente a livello di log, avrà la persistenza globale dei dati

In un sistema distribuito ho varie tipologie di guasti: **caduta di nodi, perdita di messaggi, partizionamento della rete**

Metodo di Lamport

Utilizzato per garantire l'isolamento con il controllo di concorrenza TS: ciascuna transazione acquisisce un timestamp corrispondente all'istante di tempo in cui la transazione distribuita deve sincronizzarsi con le altre transazioni.

Timestamp caratterizzato da 2 gruppi di cifre: le meno significative identificano un nodo, le più significative identificano **eventi locali** (numerati con un contatore incrementale locale). Quando 2 nodi si scambiano un messaggio, i timestamp vengono sincronizzati: l'evento "ricezione" deve avere un timestamp successivo all'evento "invio" (viene incrementato il contatore locale del ricevente).

In questo modo i timestamp riflettono l'ordinamento degli eventi indotto dagli scambi di messaggi.

Rilevazione distribuita dei deadlock (protocollo NTP)

Per rilevare deadlock distribuiti basta la soluzione semplice del **timeout**.

Esiste però un algoritmo efficiente di rilevazione dei deadlock distribuiti.

Posso avere sotto-transazioni in attesa locale (rispetto ad una risorsa bloccata da un'altra sotto-transazione della stessa transazione in un DBMS) oppure in attesa del ritorno di una chiamata a procedura remota su un DBMS remoto.

1. Si caratterizzano le condizioni di attesa visibili su ogni DBMS mediante **condizioni di**

precedenza: una sottotransazione T_i , attivata da un DBMS remoto, attende un'altra transazione T_j (locale), che a sua volta sta attendendo una sottotransazione remota (**sequenza di attesa**);

es. Ein --> T_i --> T_j --> Eout

2. L'algoritmo è di tipo distribuito, e viene attivato periodicamente ai vari DBMS del sistema. Analizza la situazione locale e comunica le sequenze di attesa alle altre istanze dell'algoritmo sugli altri DBMS.
3. Per evitare il riconoscimento multiplo dello stesso deadlock, l'algoritmo scambia messaggi solamente "in avanti" (verso Eout) e solo se $i > j$

Ad ogni passo, l'algoritmo si comporta come segue:

- I messaggi con sequenze d'attesa di altri DBMS vengono letti, si compone l'informazione nel grafo di attesa locale. Ogni transazione ha un solo nodo, indipendentemente dal numero di sottotransazioni di cui è composta
- Si fa una ricerca locale di deadlock. Se sono rilevati, vengono risolti forzando l'abort di una transazione coinvolta;
- Si ricomputano le sequenze di attesa e si trasmettono "in avanti" le sequenze. L'algoritmo si disattiva.

Protocollo di commit a 2 fasi

Il protocollo garantisce l'atomicità delle sotto transazioni distribuite.

In una transazione distribuita si individuano il **coordinatore (TM)** e più partecipanti (**RM**). Bisogna fare in modo che nel log vengano scritte le azioni da compiere riguardo alla transazione.

Nel log del **Transaction Manager**:

1. **Prepare:** contiene l'identità dei processi RM coinvolti dalla transazione
2. **Global Commit/Abort:** istante di tempo che esprime la decisione atomica e persistente riguardo alla transazione, decisione che si dovrà riflettere su tutti i partecipanti
3. **Complete:** indica la conclusione del protocollo

Nel log del **Resource Manager**:

1. **I record usuali (begin, insert, update, ...)**
2. **Ready:** indica l'irrevocabile disponibilità a partecipare al protocollo a 2 fasi contribuendo per una decisione di commit
3. **Local Commit/Abort:** indica la ricezione della decisione del TM

Il record di **ready** fa perdere al RM ogni autonomia decisionale (che viene comandata dal TM). Un RM può scrivere un **ready** solo se è in uno stato affidabile.

La finestra di incertezza è il periodo di tempo che intercorre tra la scrittura del record **ready** e la ricezione della decisione del TM (**Local Commit/Abort**). Infatti in questo lasso di tempo il RM si "mette in attesa" del TM (che potrebbe guastarsi, si potrebbero perdere messaggi ...). Durante la finestra di incertezza le risorse per cui è stato emesso il **ready** sono lockate: rischio di bloccare grandi porzioni del DBMS se la finestra di incertezza è

troppo ampia!

Protocollo in assenza di guasti:

Prima Fase

1. **TM** scrive **prepare** nel suo log e invia **prepare** agli **RM** coinvolti. Imposta un timeout sulla ricezione dell'**ack**.
2. **RM** in stato affidabile scrivono **ready** nel loro log e trasmettono **ready** al **TM**. Se **RM** non è affidabile, invia **not-ready** (terminando il protocollo). Se **RM** abortisce una sotto-transazione in modo autonomo e ne disfa gli effetti, causa un **global abort**;
3. **TM** colleziona messaggi degli **RM**. Se sono tutti positivi, scrive nel suo log **global commit**. Se almeno un messaggio è negativo, oppure scatta il timeout su qualche ricezione, **TM** scrive **global abort**.

Seconda fase

1. **TM** trasmette la sua decisione globale agli **RM**. Imposta un timeout.
2. **RM** in stato di **ready** attendono decisione del **TM**: appena arriva, scrivono **commit** nel log (o **abort**). Rispondono al **TM** con un **ack**. A questo punto le cose procedono localmente come si è già visto.
3. **TM** attende **ack** degli **RM**. Se arrivano tutti, scrive nel suo log **complete** e termina protocollo. Se scatta timeout, **TM** imposta un altro timeout e ripete trasmissione verso **RM** di cui non ha avuto notizia, loop finchè non arrivano tutti gli **ack**.

In contesti di transazioni eseguite periodicamente con gli stessi attori, spesso si instaurano sessioni di comunicazione semi-permanenti tra **TM** e i vari **RM** coinvolti, in modo da rendere efficiente e robusta l'esecuzione delle transazioni.

Protocolli di ripristino

A) Caduta di un partecipante:

la caduta di un partecipante comporta la perdita del buffer e può lasciare la base di dati in uno stato inconsistente. Lo stato incerto si deduce dai log, durante il loro **warm restart**, andando a leggere l'ultimo record del log:

- **abort, azione:** devo effettuare **undo**
- **commit:** devo effettuare **redo**
- **ready:** Il partecipante è in dubbio circa l'esito della transazione. Si costituisce un insieme di transazioni in dubbio (insieme ready), e si richiede info sul loro stato al TM (remote recovery request).

B) Caduta del coordinatore:

Avviene durante la trasmissione dei messaggi, comporta eventuale perdita dei messaggi. Si controlla ultimo record del log:

- **prepare:** causa blocco nei RM! Si opta per inviare un **global abort** e passare alla seconda fase, oppure si ripete la prima fase sperando che siano ancora **ready**.
- **global decision:** parte degli RM può essere in blocco. **TM** ripete la seconda fase.
- **Complete:** nessun effetto sulla transazione.

I partecipanti possono ricevere più volte lo stesso messaggio, in particolar modo riguardo alla decisione. In questo caso non fanno nulla ma devono cmq rispondere con **ack**.

C) Perdita di messaggi / partizionamenti di rete:

- Perdita **prepare** o successivo **ready**: non distinguibili dal **TM**. Scatta comunque timeout della prima fase quindi ho **global abort**;
- Perdita **global decision** o **ack**: non distinguibili. Scatta timeout seconda fase, ripetizione del messaggio.
- **Partizionamento rete**: non dà problemi, perchè la transazione avrà successo solo se **TM** ed **RM coinvolti** si trovano nella stessa partizione.

Ottimizzazioni protocollo:

1. **Protocollo di abort presunto**: se un **TM** riceve **remote recovery** da una transazione in dubbio che non gli è nota, risponde di default con **global abort**

- Si evita scrittura sincrona (**force**) di prepare e global abort. Complete può essere omesso
- Solo ready, local commit e global commit devono essere scritti con **force**

2. **Read-Only**: se **RM** ha svolto solo operazioni di lettura, risponde **read-only** al messaggio di prepare ed esce dal protocollo. **TM** ignora tutti **RM read-only** nella seconda fase.

Versioni differenti del protocollo:

Protocollo a 4 fasi: si replica il **TM** con un processo backup su un nodo differente, in modo che possa subentrare all'originale in caso di guasto.

Protocollo a 3 fasi: usando una terza fase, ogni **RM** può diventare **TM**.

Standardizzazione: X-OPEN-DTP

Standard X-Open Distributed Transaction Processing: garantisce l'interoperabilità tra diversi DBMS distribuiti.

Interfaccia client-TM : **TM-Interface** (definisce i servizi del coordinatore offerti ad un client per eseguire il commit di partecipanti eterogenei)

Interfaccia TM-RM : **XA-Interface** (definisce i servizi di partecipanti passivi che rispondono a chiamate del coordinatore)

1. **RM**: sono passivi, rispondono a **remote procedure calls** dei **TM**

2. **Protocollo**: commit a 2 fasi (abort-presunto e read-only)

3. **Decisioni euristiche**: dopo un guasto, gli operatori forzano commit o abort. Se le decisioni forzate sono inconsistenti, queste vengono riportate al client.

Interfaccia TM:

1. **tm_init/exit**: inizio/fine dialogo con il client

2. **tm_open/term**: apertura/chiusura sessione con **TM**

3. **tm_begin/commit/abort**: richiesta di esecuzione transazione e decisione globale

Interfaccia XA:

1. **xa_open/close**: inizio/termine dialogo con RM
2. **xa_start/end**: attivano/terminano una transazione RM
3. **xa_precomm**: richiesta al RM di svolgere la prima fase del protocollo
4. **xa_commit/abort**: decisione globale comunicata agli RM (seconda fase dl protocollo)
5. **xa_recover**: richiesta al RM di inizio procedura di ripristino. RM risponde con 3 insiemi di transazioni: in dubbio, abort/commit euristico;
6. **xa_forget**: consente al RM di dimenticare transazioni decise in modo euristico: TM termina transazioni in dubbio e esegue **xa_forget** per le speculazioni errate

Parallelismo delle basi di dati

Le applicazioni DBMS sono tipicamente scalabili (si conservano prestazioni elevate all'aumentare del carico). Il carico può aumentare sia come dimensione sia come complessità delle query eseguite. Il carico può essere **transazionale** (tipico dei sistemi OLTP) o di **analisi dati** (tipico dei sistemi OLAP)

Il parallelismo viene ottenuto aumentando il numero di processori che lavorano su una base di dati.

Tipologie di parallelismo:

- **inter-query**: a ciascun processore è indirizzata una parte del carico (OLTP)
- **intra-query**: una query complessa è scomposta su più processori (OLAP)

Tipi di architetture (sottosistemi collegati in LAN):

1. **Shared nothing**: ogni sottosistema è composto da CPU, RAM, HD
2. **Shared memory**: i processori condividono RAM e HD;
3. **Shared disks**: ogn sottosistema è composto da CPU e RAM, HD condivisi

Le prestazioni di questi sistemi sono valutate in termini di **speed-up** e **scale-up**:

1. **Speed-up**: (inter-query) andamento **tps** su **#processori**. Asintoticamente ho un andamento lineare;
2. **Scale-up**: (inter/intra-query) andamento **costo per transazione** su **#processori**. Asintoticamente ho un andamento costante.

Per valutare le prestazioni si ricorre ad un insieme di **benchmark standard** (TPC - **Transaction Processing Performance Council**) che definiscono i criteri per gli "applicativi tipo", i metodi per la generazione casuale di dati, il codice delle transazioni, la dimensione della base dati, la distribuzione degli arrivi delle transazioni e le modalità di misurazione/certificazione.

Normalmente si associa il parallelismo con la **frammentazione dei dati**, distribuendo i vari frammenti su più processori e dischi. Questo consente l'esecuzione di **join distribuiti**, essenziale per il parallelismo intra-query.

Replicazione delle basi di dati

Servizio essenziale per la realizzazione di architetture distribuite. Effettuato da appositi prodotti (**data replicator**). La replicazione rende un sistema meno sensibile ai guasti e aumenta la garanzia di persistenza dei dati (forma sofisticata di backup)

La funzione principale di un data replicator è mantenere l'allineamento tra copia principale e copie secondarie.

Modalità di replicazione:

1. **Asimmetrica:** ho una gerarchia tra le copie, le modifiche vengono indirizzate alla copia principale che poi le propaga alle copie secondarie;
2. **Simmetrica:** non ho gerarchia tra le copie, le modifiche possono essere indirizzate a qualunque copia, la propagazione avviene in modalità p2p.

Modalità di trasmissione delle variazioni:

1. **Asincrona:** vi è una transazione master che aggiorna la base di dati principale, e poi in un secondo tempo viene emessa una transazione di allineamento verso le basi di dati secondarie che propaga la modifica;
2. **Sincrona:** l'unica transazione ad agire sul sistema è quella master che gestisce sia l'aggiornamento della copia principale, sia l'aggiornamento delle copie secondarie

Modalità di allineamento:

1. **Refresh:** ad intervalli regolari l'intero contenuto della base di dati principale viene copiato sulle basi di dati secondarie.
2. **Incrementale:** si generano insiemi di variazioni sui dati (**delta plus, delta minus**) che vengono inviate alle copie secondarie

L'allineamento può avvenire periodicamente, a comando, ad accumulo di variazione

*Gestione di replicazione asimmetrica, asincrona ed incrementale: il replication manager viene implementato tramite 2 moduli (**capture** e **apply**) e una base di dati contenente le variazioni (**delta plus/minus**). Tramite i **triggers** il modulo capture registra le variazioni sulla copia principale e le inserisce nel database delle variazioni. Questo viene letto con politiche definite dall'operatore dal modulo apply che si preoccupa di allineare le copie secondarie.*

*Molto importante nella gestione dei dispositivi mobili: rappresentano porzioni di dati secondarie frammentate che rimangono "scollegate" dal sistema principale anche per molto tempo. Quando ritornano in comunicazione, bisogna **riallineare** sia la base di dati principale (inserendo le transazioni eseguite sul frammento) sia la base di dati mobile (aggiornando il frammento distribuito): questo è fatto tramite procedure di replicazione simmetrica con riconciliazione dei dati.*

Analisi dei dati

Storicamente è stata posta più enfasi su prodotti tipicamente **OLTP**, ovvero su sistemi serventi un gran numero di transazioni relativamente semplici, richieste da un gran numero di utenti in contemporanea. In questo modo si accumulano un gran numero di informazioni che possono essere analizzate anche a livello di trend per fornire un ausilio alla pianificazione strategica aziendale.

Questo si traduce nell'esigenza di servire un numero relativamente ristretto di utenti (solitamente amministratori o analisti) i quali vogliono eseguire delle query particolarmente complesse ed articolare in modalità "offline". Queste query sono mirate ad analizzare i dati aggregati presenti nella base di dati, oppure a cercare regolarità/irregolarità nei trend di certi dati. Si introduce quindi l'esigenza di definire prodotti di tipo **OLAP**, che per i motivi sopra detti hanno requisiti e specifiche ortogonali ai sistemi **OLTP** (restando nell'ambito di databases relazionali) e vanno quindi realizzati a parte per non avere influssi negativi sulle prestazioni.

OLTP:

1. concorrenza elevata
2. operazioni "predefinite" e "semplici"
3. operazioni coinvolgono "pochi" dati ma aggiornati
4. enfasi sull'importanza delle proprietà **ACID**

OLAP:

1. bassa concorrenza
2. operazioni "custom" e "complesse"
3. le query devono accedere ad insiemi vasti di dati, anche storici e non aggiornati
4. le proprietà **ACID** non sono fondamentali dato che si accede in sola lettura

Per ragioni tecniche e organizzative, è quasi impossibile configurare un sistema dbms perchè sia in grado di soddisfare efficientemente entrambi i requisiti. In un sistema **OLTP+OLAP**:

- o si rallentano parecchio le transazioni OLTP (bloccate dai lock vasti delle OLAP) o non si riesce a completare le transazioni OLTP (risorse bloccate dalle transazioni OLTP)
- o le OLTP rallentano per l'aggiornamento degli indici, o le OLAP non hanno a disposizione gli indici necessari per eseguire efficientemente
- nelle OLTP raramente si pre-computano le query, cosa basilare per le OLAP
- OLTP richiede elevato numero di tabelle frammentate, OLAP vuole poche tabelle denormalizzate
- Gli algoritmi di join sono differenti
- Qualità dei dati: OLTP ha un gran numero di dati replicati e disomogenei, analisi OLAP richiede dati aggregati e normalizzati

Bisogna **separare i due sistemi** (parallelismo non aiuta, problema è intrinseco!): si introducono i **data warehouses**.

Data warehouse:

1. **Base di dati integrata:** dati provengono da vari **data source** (tipicamente **OLTP**), integrazione a livello aziendale.

2. **Contiene informazione storica:** di interesse, a differenza dell'enfasi sul valore attuale del dato nei sistemi OLTP
3. **Dati aggregati**
4. **Autonomo:** è fisicamente separato dai **data sources**
5. **Base Dati offline:** importazione dati asincrona e periodica. Dati contenuti non perfettamente aggiornati, entro un limite accettabile per operare l'analisi.

Componenti di un DW

a) DATA SOURCES

Sorgenti dei dati. Sono sistemi informativi già presenti in azienda oppure sistemi informativi esterni. Tipologie varie (anche legacy)

b) DATA WAREHOUSE SERVER

Sistema dedicato alla gestione del data warehouse. Memorizzazione dei dati strutturate per rendere efficienti operazioni complesse. Implementa speciali operazioni (roll up, drill down, data cube). Si possono costruire **data mart** specializzati (viste materializzate) che soddisfino esigenze specifiche di analisi.

c) SISTEMA DI ALIMENTAZIONE

- **Estrazione, pulizia, trasformazione** dei dati: i dati vanno raccolti dai vari data source, poi bisogna eliminare i dati inconsistenti e/o palesemente errati. A questo punto bisogna riconciliare i dati convertendoli in modo ottimale per prepararli ad un'analisi significativa
- **Caricamento dei dati:** i dati vengono allineati tipicamente eseguendo un refresh iniziale per popolare la base dati, e poi si aggiorna in maniera incrementale. Si utilizzano archivi variazionali e si marcano le cancellature come dati storici.

D) STRUMENTI DI ANALISI

Strumenti con interfacce user-friendly in grado di effettuare analisi dei dati: analisi multidimensionale e data mining.

Modello multidimensionale dei dati (a stella)

1. **Fatto:** concetto del sistema informativo aziendale sul quale ha senso svolgere un'analisi.
2. **Misura:** proprietà atomica di un fatto (da analizzare)
3. **Dimensione:** prospettiva lungo la quale effettuare l'analisi (tipicamente organizzate in gerarchie)

Estensione -> **modello snow-flake:** evita ridondanze eccessive (dalla tabella dei fatti è sempre possibile raggiungere le tabelle delle dimensioni muovendosi lungo associazioni n:1)

Una naturale rappresentazione dei fatti è data dai cubi n-dimensionali, composti da elementi atomici detti celle. Ogni dimensione fisica corrisponde ad una dimensione concettuale del fatto. Fissando le dimensioni, si possono determinare le coordinate di un cubo, ed eseguire operazioni significative per l'analisi dei dati.

Operazioni sui dati multidimensionali:

- **slice & dice:** seleziona e proietta un sottoinsieme di celle.
- **roll up:** operazioni aggregate sui dati (misure additive / non additive)
- **drill down:** disaggregazione dei dati (inverso di roll-up)
- **pivot:** re-orienta il cubo

Progettazione di data warehouse

Differente dalla progettazione di una base dati OLTP perchè i requisiti operativi sono differenti, così come i dati trattati. Inoltre è vincolata dai dbms già esistenti.

Si pone enfasi sulla generalizzazione e sulla chiarezza concettuale.

1. **Limitata frammentazione:** visione sintetica dei dati

2. **Evidenziare aspetti comuni**

3. **Significati intuitivi** per l'utente finale (rispetto alle entità proposte)

4. **Metadati fondamentali** per la comprensione del sistema finale

Analisi DB esistenti - integrazione - progettazione (concettuale, logica, fisica)

a) **Analisi data sources esistenti**

1. **Selezione** sorgenti utili (qualità sorgenti, correlazione con requisiti, priorità)

2. **Traduzione** in modello concettuale di riferimento

3. **Analisi data source:** identificare fatti, misure, dimensioni

b) **Integrazione data sources**

1. **Fusione** in un'unica base di dati globale (patrimonio informativo aziendale)

2. **Unificare** diverse rappresentazioni degli stessi aspetti di interesse

3. **Identificazione, analisi, risoluzione** dei conflitti (terminologici, strutturali, di codifica)

c) **Progettazione DW**

1. **Concettuale:** completare rappresentazione dei concetti dimensionali necessari per l'analisi (storico-geografici ad esempio)

2. **Logica:** effettuare trade-off tra aggregazione e normalizzazione dei dati

3. **Fisica:** individuare distribuzione dei dati e relative strutture di accesso

Per la realizzazione fisica esistono tecniche specifiche:

- **MOLAP (Multidimensional OLAP):** strutture interne non relazionali (migliori prestazioni)
- **ROLAP (Relational OLAP):** strutture interne relazionali, gestisce bene grandi quantità di dati

Indici bitmap: permettono implementazione efficiente di confronti logici e semplici confronti strutturali.

Indici di join: precomputano join tra dimensione e tabella dei fatti

Materializzazione di viste: le viste più usate vengono precalcolate in modo da velocizzare le

computazioni

Data Mining

Obiettivo: estrazione di informazione nascosta nei dati, informazione in grado di migliorare il processo decisionale

Utilizzi:

- **Analisi di mercato:** modalità di acquisizione dei beni da parte del consumatore
- **Analisi di comportamento:** evidenziare illeciti
- **Previsione:** previsione dei costi
- **Controllo:** errori di produzione

Definisco delle **regole di associazione** che tentano di evidenziare regolarità / irregolarità nei dati (**corpo:** premessa della regola , **testa:** conseguenza della regola)

Caratteristiche delle regole di associazione:

1. **Supporto:** probabilità di presenza contemporanea di corpo e testa in una transazione. Misura l'importanza di una regola
2. **Confidenza:** probabilità di presenza della testa essendo presente il corpo, in una transazione. Misura il grado di affidabilità della regola.
3. **Formulazione del problema:** estrarre tutte le regole con supporto e confidenza superiore a valori fissati.

Pattern Sequenziali: si individua un sottoinsieme dei dati del DW su cui effettuare il data mining, e codifica dei dati in un valore significativo per l'algoritmo di data mining. Applicando le tecniche di data mining, si evidenziano pattern ricorrenti nei dati. I pattern vengono valutati, cercando di trarre implicazioni applicative dai pattern.

In questa fase spesso si usa la **discretizzazione** dei dati per rendere compatta la loro rappresentazione astruendo dai dettagli inutili (valore numerico reale può essere mappato su valore "alto" "medio" "basso" rispetto a qualche cosa). Questo permette di determinare facilmente i valori critici e facilita le analisi successive.

Con la **classificazione** invece si cataloga un fenomeno particolare in una classe definita. Vengono presentati fatti elementari (tuple). Si costruisce automaticamente un algoritmo classificatore a partire da un insieme di dati di prova (training set). Tipicamente i classificatore sono alberi di decisione.