

DB2

Silviu Filote

August 3, 2021

Contents

1	Transaction	2
2	Concurrency control	4
3	Reliability	25
4	Commit Protocols	35
5	Distributed Databases	44
6	Parallel and Replicated Databases	54
7	Data Warehouses	59
8	Active Databases	65
9	Physical data structures and query optimization	73

1 Transaction

- **Transazione:** indica una qualunque sequenza di operazioni lecite che, se eseguita in modo corretto, produce una variazione nello stato di una base di dati.
- La transazione viene encapsulata all'interno di 2 comandi
 - begin transaction
 - end transaction
- All'interno della trasazione possono essere eseguiti solo una volta i seguenti comandi:
 - commit work (commit)
 - rollback work (abort)
- **Transaction system (OLTP):** è un sistema in grado di garantire l'esecuzione concorrente di transazioni da diverse applicazioni
- Normale esecuzione di una transazione
 - **begin transaction**
 - codice per la manipolazione dei dati (lettura, scrittura)
 - **commit work - rollback work**
 - no manipolazione dei dati (no accessi o scritture)
 - **end transaction**
- **Proprietà di una transazione → ACID properties**
 - **Atomicity**
 - * Il processo deve essere suddivisibile in un numero finito di unità indivisibili, chiamate transazioni.
 - * **L'esecuzione** di una transazione deve essere per definizione **o totale o nulla**, e non sono ammesse esecuzioni parziali;
 - * Possibili comportamenti di un'esecuzione possono essere:
 1. Commit work: **SUCCESS**
 2. Rollback work oppure errore prima del commit: **UNDO**
 - . *UNDO (disfare)*: in caso di fallimento della transazione deve essere possibile "disfare" l'azione svolta sui dati

3. Errore dopo il commit: **REDO**

- *REDO (rifare)*: se la transazione ha avuto successo ma le modifiche al DB non sono state rese permanenti, le modifiche vanno ripetute.

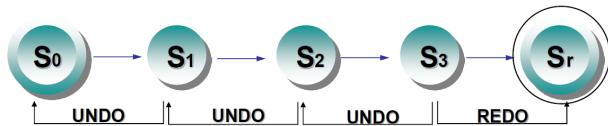


Figure 1: esecuzione transazione

– Consistency

- * il database rispetta i **vincoli di integrità**, sia a inizio che a fine transazione. Non devono verificarsi contraddizioni (incoerenza dei dati) tra i dati archiviati nel DB.

– Isolation

- * ogni transazione deve essere eseguita in modo isolato e indipendente dalle altre transazioni, l'eventuale fallimento di una transazione non deve interferire con le altre transazioni in esecuzione.

– Durability

- * detta anche persistenza, si riferisce al fatto che una volta che una transazione abbia richiesto un commit work, i cambiamenti apportati non dovranno essere più persi.
- * Per evitare che nel lasso di tempo fra il momento in cui la base di dati si impegna a scrivere le modifiche e quello in cui li scrive effettivamente si verifichino perdite di dati dovuti a malfunzionamenti, **vengono tenuti dei registri di log dove sono annotate tutte le operazioni sul DB**.

- **A**tomicity
 - Abort-rollback-restart
 - Commit protocols
- **C**onsistency
 - Integrity checking of the DBMS
- **I**solation
 - Concurrency control
- **D**urability
 - Recovery management

Figure 2: Transaction Properties and Mechanisms

2 Concurrency control

- Esecuzione concorrente è importante e può causare anomalie

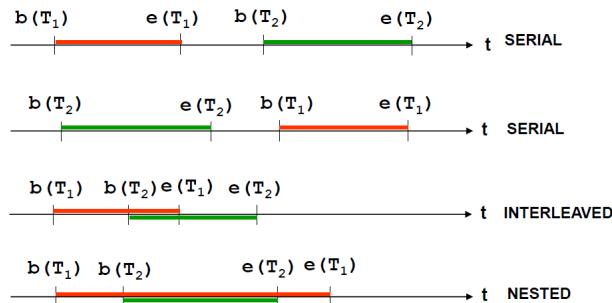


Figure 3: Concurrent Executions

- Problemi che si possono manifestare
 - **Execution with Lost Update**
 - * Un aggiornamento perso si verifica quando due transazioni diverse tentano di aggiornare contemporaneamente la stessa colonna sulla stessa riga all'interno di un database. In genere, una transazione aggiorna una particolare colonna in una particolare riga, mentre un'altra che è iniziata poco dopo non ha visto questo aggiornamento prima di aggiornare lo stesso valore. *Il risultato della prima transazione viene quindi "perso", poiché viene semplicemente sovrascritto dalla seconda transazione.*

X=100

- 1 T1: R(X,V1) La transazione T1 legge X e la pone in V1
- 2 V1 = V1 + 3 V1 = 103
- 3 T2: R(X,V2) La transazione T2 legge X e la pone in V2
- 4 V2 = V2 + 6 V2 = 106
- 5 T1: W(V1,X) X=103 La transazione T1 scrive V1 in X
- 6 T2: W(V2,X) X=106! La transazione T2 scrive V2 in X

Figure 4: Il valore scritto dalla T1 viene perso perché sovrascritto da T2

– Dirty read

- * Il termine dirty read, o uncommitted dependency, si intende un errore nella gestione della concorrenza tra transazioni dovuta ad una dipendenza write → read.
- * Il dirty read è una perdita di modifiche in una base di dati causato da una dipendenza di una transazione da un'altra non completata con successo con conseguente lettura di dati inconsistenti.

X=100

- 1 T1: R(X,V1)
- 2 T1: V1 = V1 + 3
- 3 T1: W(V1,X) X=103
- 4 T2: R(X,V2)
- 5 T1: ROLLBACK
- 6 T2: V2 = V2 + 6
- 7 T2: W(V2,X) X=109!

Figure 5: lettura incosistente

– Nonrepeatable Read

- * Nel corso di una transazione, una riga viene recuperata due volte e i valori all'interno della riga differiscono tra le letture.

```

X=100

1 T1: R(X,V1)
2 T2: R(X,V2)
3 T2: V2 = V2 + 6
4 T2: W(V2,X)      X=106
5 T1: R(X,V3)      V3<>V1!

```

Figure 6: T1 legge valori diversi a causa di una scrittura di T2

– Ghost update

- * La lettura di Y avviene per entrambe le transazioni, per cui la modifica fatta successivamente da T2 non viene applicata a T1;
- * Mentre la modifica a Z datta da T2 viene letta successivamente da T1;

```

X+Y+Z=100, X=50, Y=30, Z=20

T1: R(X,V1), R(Y,V2)
T2: R(Y,V3), R(Z,V4)
T2: V3 = V3 + 10, V4=V4-10
T2: W(V3,Y), W(V4,Z)      (Y=40, Z=10)
T1: R(Z,V5)                (for T1, V1+V2+V5=90!)

```

Figure 7: modifica non letta da parte di T1

– Phantom Insert

- * Se una transazione seleziona un insieme di righe, un'altra transazione inserisce righe che soddisfano gli stessi criteri, quando la prima transazione riesegue la query, ne risulta un insieme diverso. Si tratta di un problema di incoerenza all'interno di una transazione e affrontato dai livelli di isolamento.

```

T1: C=AVG(B: A=1)
T2: Insert (A=1,B=2)
T1: C=AVG(B: A=1)

```

Figure 8: i valori returnati la seconda volta sono diversi

Anomalies

Lost update	R1-R2-W2-W1
Dirty read	R1-W1-R2-abort1-W2
Nonrepeatable read	R1-R2-W2-R1
Ghost update	R1-R1-R2-R2-W2-W2-R1
Phantom insert	R1-W2 (new data)-R1

Figure 9: anomalie che si possono manifestare

- Principles of Concurrency Control
 - **Schedule** è una sequenza di operazioni eseguite da transazioni concorrenti
 - **Scheduler** è un componente che accetta o rifiuta le operazioni eseguite da una transazione
 - * Le decisioni dello scheduler vengono prese at running time
 - **Serial schedule** le azioni svolte dalle transazioni in maniera temporale **non si sovrappongono**

$S_2: r_0(x) \text{ (orange)} r_0(y) \text{ (orange)} w_0(x) \text{ (orange)} r_1(y) \text{ (green)} r_1(x) \text{ (green)} w_1(y) \text{ (green)} r_2(x) \text{ (pink)} r_2(y) \text{ (pink)} r_2(z) \text{ (pink)} w_2(z) \text{ (pink)}$

Figure 10: Serial schedule

- **Non serial schedule**, in questo caso le operazioni svolte dalle trasazioni sono **interleaved**.
 - * Aumento del problema della concorrenza;
 - * Le transazioni vengono eseguite in modo non seriale, mantenendo il risultato finale corretto e uguale alla pianificazione seriale;
 - * A differenza della pianificazione seriale in cui una transazione deve attendere che un'altra completi tutte le sue operazioni, nella pianificazione non seriale, l'altra transazione procede senza attendere il completamento della transazione precedente
 - * Può essere di due tipo
 - **Serializable schedule**
 - Non-Serializable schedule

- **Serializable schedule** Viene utilizzato principalmente nella **Non serial schedule** per verificare se lo schedule porterà in uno stato di inconsistenza o meno.
 - * **Serial schedule** non ne ha bisogno perché segue una transazione solo quando la transazione precedente è completa
 - * Una pianificazione serializzabile aiuta a migliorare sia l'utilizzo delle risorse che il throughput della CPU. Questi sono di due tipi:
 - View Serializable
 - Conflict Serializable

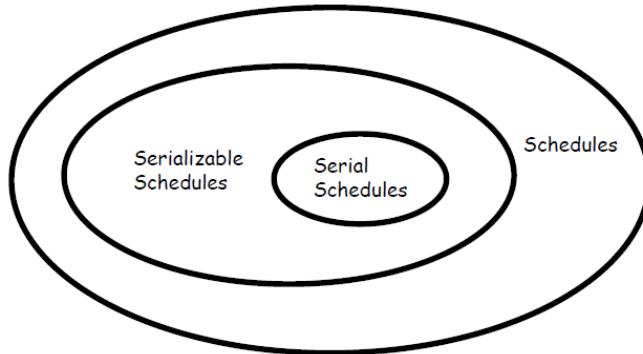


Figure 11: Basic idea

• View-serializability

- $r_i(x)$ **Reads from** $w_j(x)$ in uno schedule se:
 - * la scrittura $w_j(x)$ precede $r_i(x)$
 - * non c'è nessun'altra scrittura $w_k(x)$ tra $w_j(x)$ e $r_i(x)$
- $w_i(x)$ in uno schedule si dice **final write** se è l'ultima scrittura di x all'interno dello schedule
- Due schedule si dicono **view equivalent** se hanno gli stessi **final writes** e **reads from**
- Uno schedule si dice **view serializable** se è equivalente ad un serial schedule
- View equivalent \rightarrow view serializable
- **VSR** sono i schedule view serializable

T0						
$S_3 : w_0(x)$	$r_2(x)$	$r_1(x)$	$w_2(x)$	$w_2(z)$		
$S_4 : w_0(x)$	$r_1(x)$	$r_2(x)$	$w_2(x)$	$w_2(z)$		Stesse scritture finali T2
$S_5 : w_0(x)$	$r_1(x)$	$w_1(x)$	$r_2(x)$	$w_1(z)$		R2(x) legge W1(x) - R1(x) legge W0(x)
$S_6 : w_0(x)$	$r_1(x)$	$w_1(x)$	$w_1(z)$	$r_2(x)$		R2(x) legge W1(x) - R1(x) legge W0(x)
	$T0$					

Figure 12: esempio

- S_3 è view equivalent a S_4 per cui risulta essere view serializable
- s_5 allo stesso modo con S_6
- se uno schedule presenta le anomalie, non è view serializable
 - * Lost update
 - * Dirty read
 - * Nonrepeatable read
 - * Ghost update
 - * Phantom insert
- NB: due schedule per essere view equivalent devono leggere e scrivere dalle stesse transazioni in entrambi i schedule per quelle determinate variaibili
- Decidere se due schedule siano view equivalence viene effettuato in **polynomial time**
- Mentre decidere se uno schedule sia view serializability è un problema **NP - complete**

• Conflict-serializability

- Action a_i is **conflicting** with a_j se sono soddisfatte tutte le seguenti condizioni:
 - * devono appartenere a transazioni diverse $i \neq j$
 - * operano sulla stessa variaibile
 - * e una delle due azioni è un **write operation**
 - * Esempio:
 - rw e $wr \rightarrow$ read - write conflicting
 - $ww \rightarrow$ write - write conflict
- Due schedules si dicono **Conflict equivalent** se contengono le stesse operazioni e tutte le coppie in conflitto si verificano nello stesso ordine

- Uno schedule si dice **conflict serializable** se è conflict equivalent a uno schedule seriale
 - * Uno schedule si chiama conflcit serializable sepuò essere trasformato in un serial schedule facendo lo **swapping** tra operazioni che non risultano essere in conflitto
- **CSR** sono tutti gli schedule conflcit serializable
- **Ogni CSR è anche VSR, ma non è sempre vero il contrario.**

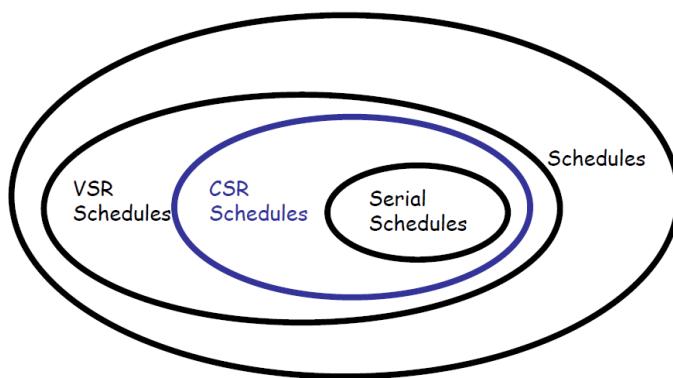


Figure 13: schema

- Per testare la conflict serializability si utilizza il **Conflict graph**
 - * I nodi indicano le transazioni (T_1, T_2, \dots, T_n)
 - * Un arco da una transazione T_i verso T_j implica l'esistenza di un conflitto tra un'azione a_i di T_i e un'azione a_j di T_j , la freccia viene rivolta verso l'ultima azione di conflitto. (segue le politiche di precedenza da a_{prima} a a_{dopo})
 - i conflitti si propagano lungo tutto lo schedule non solo tra le azioni adiacenti
 - si prende quindi la prima azione dello schedule e la si confronta **solo** con tutte le azioni **precedenti**
 - * Effettuando la **topological sort** sul grafo si viene a generare un grafo seriale equivalente a quello valutato
 - * **Teorema** un schedule si dice CSR se e solo se il conflict graph risulta essere **aciclico**
 - NB: guardare le frecce dei nodi per verificarne la ciclicità

- Se il grafo risulta essere aciclico allora possiede un **topological sort**, e solitamente si parte dalla transazione con grado minore.

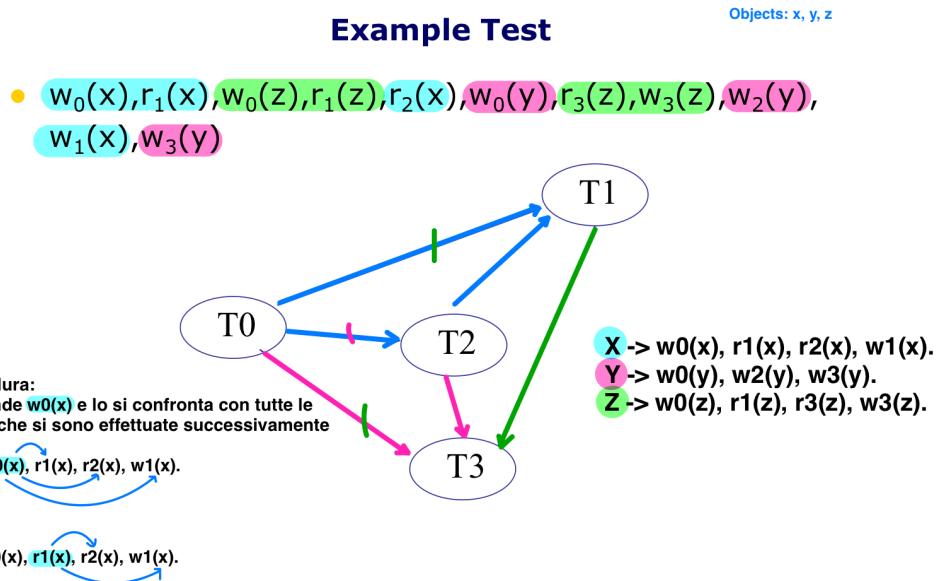


Figure 14: esempio

- Partendo dall'esempio sopra citato una possibile soluzione dell'ordine topologico risulta essere: $T_0 < T_2 < T_1 < T_3$

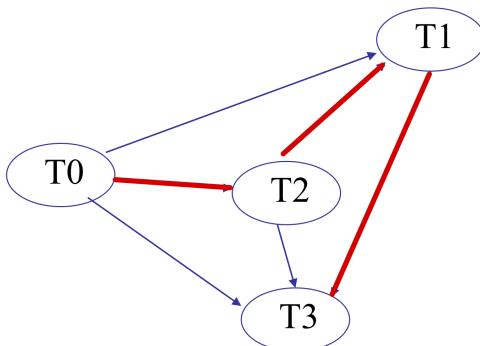


Figure 15: possibile soluzione

- **Locking**

- Una transazione si dice **well formed wrt locking** se
 - * le ***read operations***
 - sono precedute da **r_lock** (SHARED LOCK)
 - e seguite da **unlock**
 - * le ***write operations***
 - sono precedute da **w_lock** (EXCLUSIVE LOCK)
 - e seguite da **unlock**
- se una transazione prima legge e poi scrive un oggetto, modifica il **r_lock** in **w_lock** (*lock escalation*)
 - Primitives:
 - **r-lock**: read lock
 - **w-lock**: write lock
 - **unlock**
 - Possible states of an object:
 - **free**
 - **r-locked** (locked by a reader)
 - **w-locked** (locked by a writer)

Figure 16: Lock Primitives

- il **lock manager** riceve le primitive dalle transazioni e fornisce le risorse richieste secondo la **conflict table**
 - * quando il lock è garantito su una determinata risorsa, la risorsa viene acquisita
 - * mentre l'unlock rilascia

REQUEST	RESOURCE STATE		
	FREE	R_LOCKED	W_LOCKED
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 DEPENDS	OK FREE

Figure 17: conflict table

- **Two Phase Locking**

- Si dice che una transazione segua il Two Phase Locking protocollo se il locking e l'unlocking possono essere eseguiti in due fasi.
 - * **Growing Phase:** È possibile acquisire nuovi lock su elementi di dati, ma nessuno può essere rilasciato
 - * **Shrinking Phase:** I blocchi esistenti possono essere rilasciati ma non è possibile acquisirne di nuovi

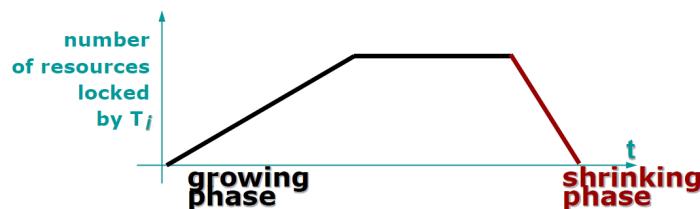


Figure 18: Two-Phase Locking

- Se lo scheduler
 - * Utilizza la well formed transaction
 - * Garantisce i lock secondo i conflitti
 - * Ed è two phase
 - allora lo scheduler produce schedule chiamate **2PL**
 - Le schedule in **2PL** sono Serializable
- *Se schedule 2PL sono CSR ma non è necessariamente vero il contrario*
- Deadlocks e Starvation si possono verificare.

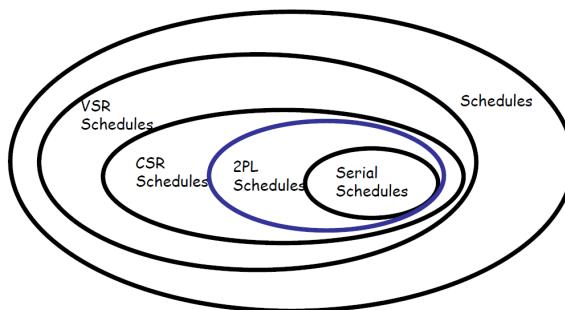


Figure 19: basic idea

- Strict 2PL

- Aggiungiamo un'altra regola a 2PL
- *I lock possono solamente essere rilasciati dopo un commit/rollback*

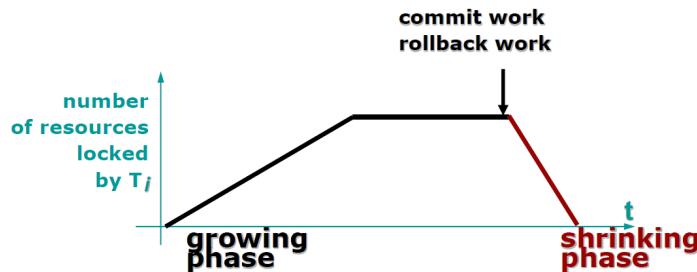


Figure 20: Strict 2PL

- Implementazione del 2-Phase Locking

- Lock tables in realtà strutture dati presenti in RAM
- Per tenere conto delle letture effettuate sugli oggetti viene utilizzato un **read counter**
- Una transazione alla richiesta di un lock può ottenerlo, oppure sospesa e inserita in attesa all'interno di una **queue**.
 - * La queue è first in first out e c'è pericolo che si vengano a formare
 - **Deadlock:** attesa infinita
 - **Starvation:** transazioni che non riceveranno mai le risorse richieste
 - * La starvation si verifica se le transazioni vogliono scrivere ma le risorse richieste sono altamente utilizzate nella scrittura

- Hierarchical Locking

- In molti sistemi reali, i lock sono specificati con **differenti gradi di granularità** (database, table, fragment, page, tuple, field). Queste risorse sono predisposte all'interno di una **gerarchia (o in DAG)**.
- La scelta dei lock level dipende dall'applicazione
 - * Livello alto nella gerarchia: molte risorse bloccate

- * Livello basso nella gerarchia: molte richieste di blocco
- Funzionamento
 - * Il locking può essere fatto a diversi livelli di granularità
 - La richiesta delle risorse viene fatta **top - down** fino al raggiungimento del livello desiderato
 - Il rilascio avviene **bottom - up**
 - * **L'obiettivo:** è quello di minimizzare il numero di locks e riconoscere i conflitti il prima possibile

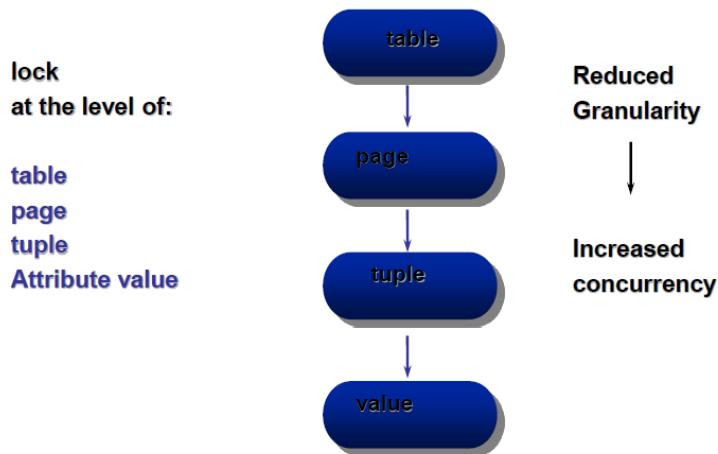


Figure 21: Livelli

- **5 Lock modes**

- **SL:** shared locks
- **XL:** exclusive locks
- **ISL:** Intention of locking in shared mode
- **IXL:** Intention of locking in exclusive mode
- **SIXL:** Lock in shared mode with intention of locking in exclusive mode (SL + IXL)
- La richiesta di lock parte dall'alto per poi scendere
- I locks sono rilasciati partendo dal basso verso l'alto
- Richiesta di lock **SL o ISL** sulle foglie, la transazione deve mantere un **ISL o IXL** lock sui **parent node**
- Richiesta di lock **IXL, XL o SIXL** sulle foglie, la transazione deve mantere un **SIXL o IXL** lock sui **parent node**

		Resource state				
Request		ISL	IXL	SL	SIXL	XL
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	

Figure 22: Conflicts in Hierarchical Locks

Example						
PAGE 1		PAGE 2				
t1		t5		Page 1: t1,t2,t3,t4		
t2		t6		Page 2: t5,t6,t7,t8		
t3		t7		Transaction 1:		
t4		t8		read(P1) write(t3) read(t8)		
				Transaction 2:		
				read(t2) read(t4) write(t5) write(t6)		
				They are NOT in conflict!		

Figure 23: esempio

Lock Sequences

t1	t5	Transaction 1:
t2	t6	<ul style="list-style-type: none"> IXL(root) SIXL(P1) XL(t3) ISL(P2) SL(t8)
t3	t7	<ul style="list-style-type: none"> La mia intenzione è quella di poter scrivere. Per cui accedo i vari livelli diminuendo sempre di più i permessi per parent node
t4	t8	<ul style="list-style-type: none"> Intenzione di leggere (più permessi - parent node) Lettura (meno permessi - foglia)
t1	t5	Transaction 2:
t2	t6	<ul style="list-style-type: none"> IXL(root) ISL(P1) SL(t2) SL(t4) IXL(P2) XL(t5) XL(t6)
t3	t7	<ul style="list-style-type: none"> Utilizzo IXL perché io poi voglio poter scrivere
t4	t8	

They are NOT in conflict!

Figure 24: esempio

• Deadlock

- Si verifica perché le transazioni concorrenti tengono e, a turno, richiedono risorse detenute da altre transazioni
- Viene rappresentato attraverso l'utilizzo di un grafico di attesa sulle risorse (wait-for graph)

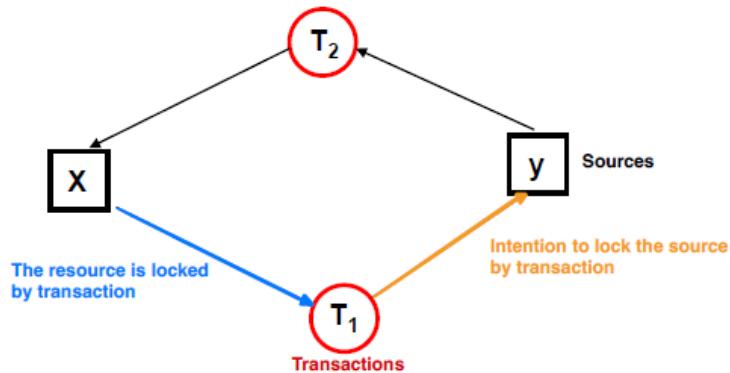
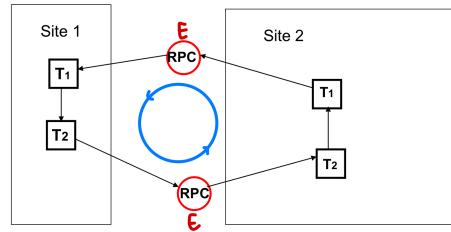


Figure 25: grafico

- **Deadlock Resolution Techniques**
 - **Timeout Method**
 - * La transazione viene interrotta dopo una lunga attesa
 - * Il problema risiede nel scegliere un giusto valore
 - Troppo lungo: attesa inutile
 - Troppo corto: interruzione non necessarie
 - **Deadlock Prevention**
 - * *Uccide le transazioni che potrebbero causare deadlock (cicli)*
 - * Una possibile scelta è quella di assegnare alle transazioni un numero che identifichi → **age**, in questo modo si le transazioni possono essere classificate in:
 - Older transaction
 - Youger transaction
 - * Le transazioni che vengono sopresse sono le youger, perchè le older transaction hanno una priorità maggiore
 - * Opzioni per scegliere la transazione da uccidere
 - **Pre-emptive:** sopprimere la transazione in attesa
 - **Non-pre-emptive:** sopprimere la transazione richiedente
 - * *Il problema: troppi "uccisioni"*
 - **Deadlock detection**
 - * *Uccide le transazioni quando c'è deadlock*
 - * Viene utilizzato un algoritmo per trovare i cicli nel grafico di attesa → **Obermark's algorithm**
- **Obermark's algorithm**
 - Siamo in presenza di un sistema distribuito composto da nodi/sites che si servono di risorse distribuite
 - L'algoritmo viene eseguito periodicamente per ogni nodo, consiste in 4 passi:
 1. Cerca eventuali deadlock dai nodi precedenti
 2. Integra i deadlock attraverso l'utilizzo di un local wait-for graph
 3. Cerca i deadlock e se ne trovi uno elimina la transazione
 4. Crea un potenziale deadlock e trasmettilo al prossimo nodo
 - **Obiettivo secondario:** rilevare ogni ciclo solo una volta
 - *Potenziale deadlock $E - T_i - T_j - E$ viene trasmesso se e solo se $i < j$*



Potential Deadlock: at Site 1: E - T1 - T2 - E
 at Site 2: E - T2 - T1 - E

Figure 26: Distributed Deadlock Detection

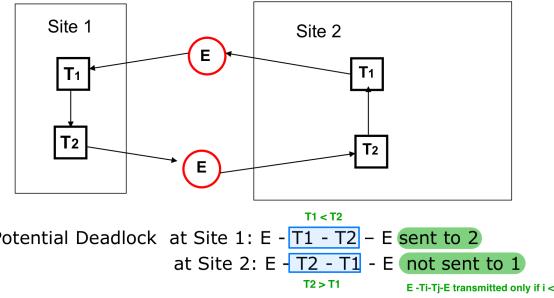


Figure 27: esempio

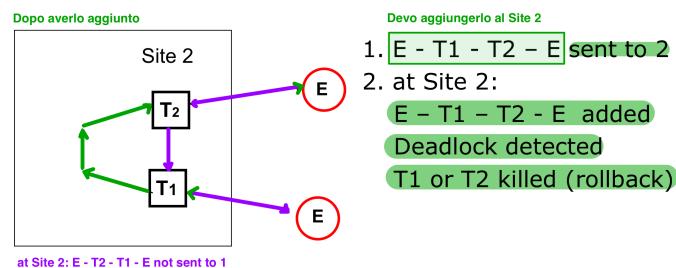


Figure 28: esempio

- **Update Lock**

- Il deadlock solitamente si manifesta quando 2 o più transazioni concorrenti leggono la stessa risorsa e poi decidono di scrivere o di write-lock.
- Per eliminare questo problema i sistemi offrono il **UPDATE LOCK (UL)**

		State		
Request		SL	UL	XL
SL	SL	OK	OK	No
	UL	OK	No	No
	XL	No	No	No

Figure 29: update lock

- **Concurrency Control Based on Timestamps**

- alternativa al 2PL
- Il timestamp è un indicatore di tempo
- Una funzione del sistema ritorna i timestamp su richiesta
- Ogni transazione ha un timestamp che rappresenta il momento in cui inizia la transazione
- *Uno schedule è accettato solamente se l'ordine dei timestamp riflette uno schedule seriale*
- **Sintassi:** *event-id.node-id*
 - * l'event-id è unico per ogni nodo
- La sincronizzazione avviene attraverso lo scambio di messaggi
- **Algoritmo:** è impossibile ricevere messaggi dal futuro quindi con timestamp maggiore, se questo dovesse succedere, a causa del fatto che i clock interni non sono sincronizzati, viene eseguita la **bumping rule**
 - * **Esempio:** il ricevitore riceve un timestamp maggiore del proprio

- * **La bumping rule:** si occupa di andare a fare il *bump* del timestamp del ricevitore oltre il timestamp del trasmettitore

- **Timestamp mechanism**

- Vengono utilizzati 2 contatori
 - * **RTM(x)** → per la lettura dell'oggetto x
 - * **WTM(x)** → per la scrittura dell'oggetto x
- Lo scheduler riceve richieste di letture e scritture dagli timestamp
 - * ***read(x, ts)***
 - Viene effettuata una lettura dell'oggetto x al timestamp ts
 - Se $ts < WTM(x)$ la richiesta viene rigettata e la transazione uccisa
 - Nel caso contrario la transazione è eseguita e $RTM(x) = \max(RTM(x), ts)$
 - * ***write(x, ts)***
 - Viene effettuata una scrittura dell'oggetto x al timestamp ts
 - Se $ts < WTM(x)$ oppure $ts < RTM(x)$ la richiesta viene rigettata e la transazione uccisa
 - Nel caso contrario la transazione è eseguita e $WTM(x) = ts$

Assume RTM(x) = 7 | WTM(x) = 4 Contatori

Request		Response	New value
<code>read(x,6)</code>	<small>6 > WTM</small>	ok	<small>RTM = max(7, 6) = 7</small>
<code>read(x,8)</code>	<small>8 > WTM</small>	ok	$RTM(x) = 8$
<code>read(x,9)</code>	<small>9 > WTM</small>	ok	$RTM(x) = 9$
<code>write(x,8)</code>	<small>8 < RTM</small>	no	t_8 killed
<code>write(x,11)</code>	<small>11 > WTM 11 > RTM</small>	ok	$WTM(x) = 11$
<code>read(x,10)</code>	<small>10 < WTM</small>	no	t_{10} killed

Figure 30: esempio

- **2PL vs TS**

- **Sono incompatibili**
- In 2PL le transazioni possono aspettare all'interno delle queue, mentre in TS sono direttamente uccise e restartate
- L'ordine di serializzazione in 2PL tiene conto dei conflitti, mentre in TS è diretta dai timestamps
- A causa delle queue il commit delle transazioni può avvenire in tempi lunghi
- In 2PL si possono generare deadlock, ma anche in TS se non si presta attenzione
- Fare il restart di una transazione costa in termini computazionali di più rispetto alla semplice attesa
- **2PL risulta essere il vincitore**

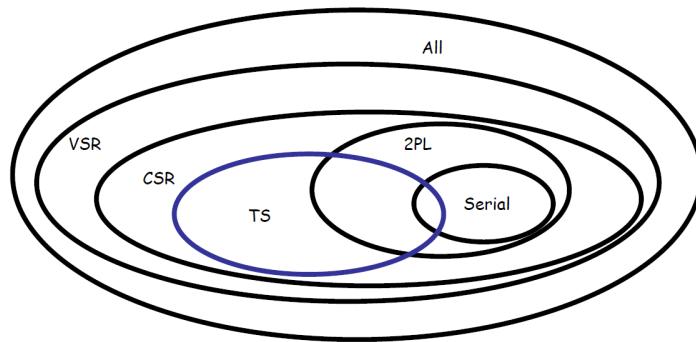


Figure 31: CSR, VSR, 2PL e TS

- **TS-based concurrency control: a variant (Thomas Rule)**

- L'unico cambiamento riguarda il write
 - * Se $ts < RTM(x)$ la richiesta viene rigettata e la transazione uccisa
 - * Se $ts < WTM(x)$ allora la scrittura è obsoleta e viene **skip-pata** la transazione
 - * Nel caso contrario la transazione è eseguita e $WTM(x) = ts$

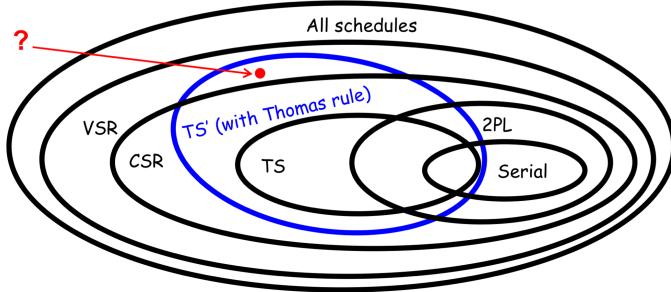


Figure 32: TS with Thomas Rule

- **Multiversion Concurrency Control - TS**

- L'idea alla base di questo protocollo
 - * Le scritture generano nuove copie, ognuna con un nuovo WTM
 - * Le letture accedono alla giusta copia
- Ogni oggetto x , possiede $N > 1$ coppie attive con un proprio $WTM_N(x)$
- C'è un solo unico globale $RTM(x)$, per oggetto x
- Le vecchie copie sono eliminate quando non ci sono più transazioni che accedono

- **Mechanism - Multiversion Concurrency Control**

- $read(x, ts)$
 - * È sempre eseguita e una coppia x_k è selezionata per la lettura
 - * Se $ts > WTM_N(x)$, allora $k = N$, quindi prendo l'ultima coppia appena scritta
 - * Altrimenti viene presa la versione k in questo modo $WTM_k(x) < ts < WTM_{k+1}(x)$
- $write(x, ts)$
 - * Se $ts < RTM(x)$, allora la richiesta non viene accettata
 - * Altrimenti viene creata una nuova copia (N viene incrementato), con $WTM_N(x) = ts$

Request	Response	New Value
<code>read(x,6)</code>	ok	<code>Max(7, 6)</code>
<code>read(x,8)</code>	ok	$RTM(x) = 8$
<code>read(x,9)</code>	ok	$RTM(x) = 9$
<code>write(x,8)</code>	no	<code>t₈ killed</code> $8 < RTM$
<code>write(x,11)</code>	ok	$N=2$, $WTM(x_2) = 11$
<code>read(x,10)</code>	ok on 1	$RTM(x) = 10$
<code>read(x,12)</code>	ok on 2	$RTM(x) = 12$
<code>write(x,13)</code>	ok	$N=3$, $WTM(x_3) = 13$

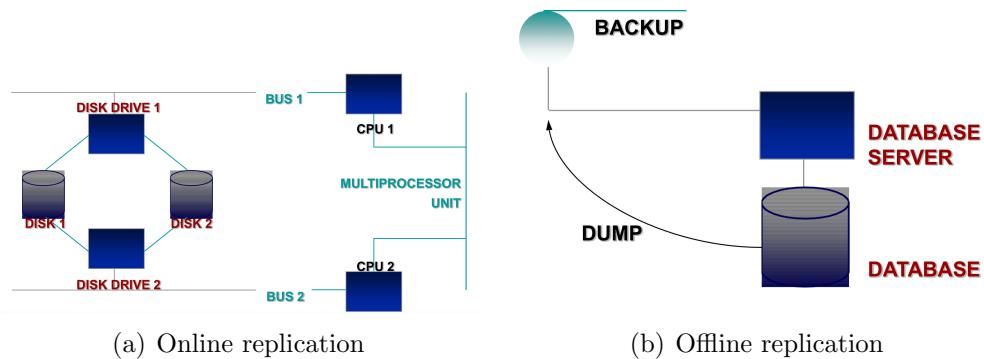
Figure 33: Esempio

- **Snapshot isolation**

- Grazie al Multi-TS si è potuto introdurre un’altro livello di isolation → **SNAPSHOT ISOLATION**
- In questo livello vengono usati solo i *WTMs*
- Ogni transazione legge la versione consistente rispetto al proprio *ts*
- Se una transazione rivela che i proprio writes danneggiano le scritture del snapshot viene eseguito l’abort → **OPTIMISTIC approach**
- *La snapshot isolation non garantisce la serializability*

3 Reliability

- Tipologie di memorie
 - **Memoria centrale** → Non persistente, volatile
 - **Memoria di massa** → Persistente ma può essere danneggiata
 - **Memoria stabile** → Persistente e non può essere danneggiata, è un'astrazione
- Come garantire memorie stabili
 - **On-line replication** → viene effettuato il mirroring su 2 dischi
 - **Off-line replication** → viene fatto il backup su tape unit



- Gestione della memoria centrale
 - Riutilizzo dei dati nel buffer
 - Avviene una **scrittura differita** nel database, le scritture vengono salvate nel buffer per poi essere eseguite in maniera asincrona. Il rischio è perdere i dati che sono stati eseguiti

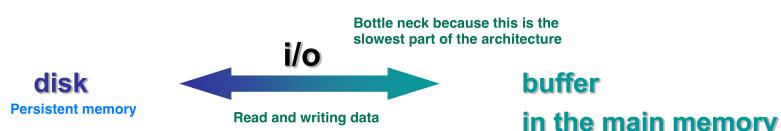


Figure 34: Main memory management

- Gestione del buffer

- Esistono 4 primitive

- * **fix**

- Utilizzato per caricare una pagina nel buffer. Dopo l'operazione, la pagina viene assegnata a una transazione
- Restituisce alla transazione l'ID dell'elemento che è stato caricato nel buffer

- * **use**

- Utilizzo di una pagina nel buffer

- * **unfix**

- Deallocazione della pagina

- * **force**

- In modo sincrono viene trasferita una pagina dalla memoria centrale al database

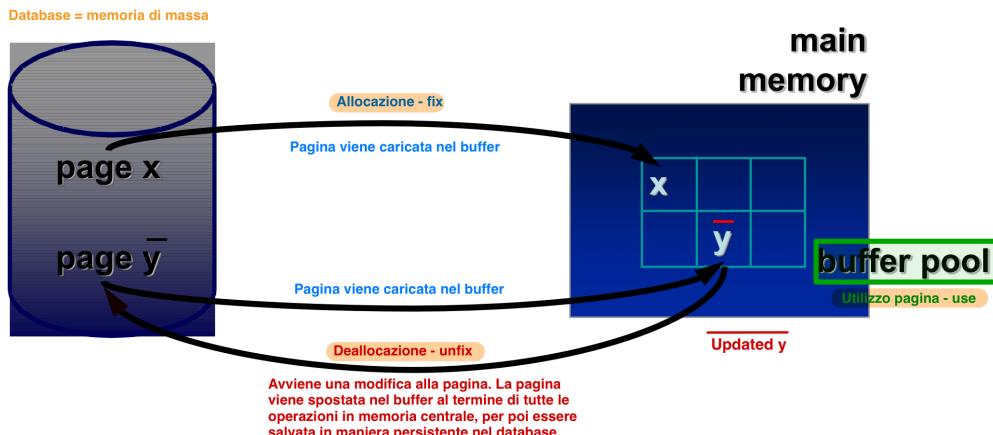


Figure 35: Main memory management

- Buffer e transazioni

- Segue il seguente schema

- * **fix**

- * utilizzo di **use** fino a quando la transazione non termina

- * **unfix**

- le pagine vengono scritte dal buffer al database in maniera asincrona
- **flush**
 - * questa primitiva è controllata dal buffer manager
 - * in maniera asincrona trasferisce le pagine dalla memoria centrale al database

- **Esecuzione della primitava *fix***

- La transazione segnala al buffer manager ID della pagina nel disco e il buffer manager si occupa di copiarla nel buffer
- Cercare la pagina target
 - * Selezione di una pagina libera
 - * Altrimenti, viene selezionata un pagina deallocated, che è necessaria
 - * Altrimenti (**se STEAL POLICY**), una pagina viene portata via da una transazione attiva
 - * Altrimenti (**se NO STEAL POLICY**) la ricerca fallisce
- lettura
 - * Se la pagina target esiste viene letta dal database e copiata nel buffer per poi essere usata nella memoria centrale

- **Politiche di gestione del buffer**

- **STEAL**
 - * le pagine vengono portate via da una transazione attiva
- **NO STEAL**
- **FORCE**
 - * le pagine vengono scritte quando avviene un commit-work
- **NO FORCE**
- *soltanente NO STEAL, NO FORCE*
- **PRE-FETCHING**
 - * anticipa le letture delle pagine
 - * molto utili nelle letture sequenziali
- **PRE-FLUSHING**
 - * anticipa le scrittura delle pagine deallocate
 - * utile per accelerare le operazioni di *fix*

• Transaction Log

- Il file di log è un file sequenziale di record che descrivono le azioni che sono state eseguite dalle varie transazioni
- Le azioni sono scritte in base al tempo di esecuzione delle transazioni, le azioni appena eseguite sono in cima al file
 - * Scrittura sequenziale
 - * Top = current instant

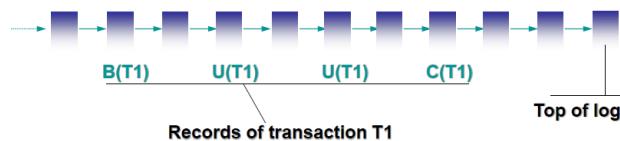


Figure 36: File di log

- Le azioni eseguite dalle varie transazioni vengono registrate nella **stable memory**

```
If UPDATE (U)
transforms o from value o1 to value o2

then the log records:
  BEFORE-STATE(U) = o1
  AFTER-STATE(U) = o2

  Update operation that
  modify the object O1/2
```

Describe a modification to the database state:
 - the before state of the update
 - the after state

Figure 37: esempio

• Using the Log

- After rollback-work or failure
 - UNDO T1: $o = o_1$ Transaction T1 rollback the object o to the previous state which is o_1
- After failure after commit
 - REDO T1: $o = o_2$
- Idempotency of UNDO and REDO:
 - $\text{UNDO}(T) = \text{UNDO}(\text{UNDO}(T))$
 - $\text{REDO}(T) = \text{REDO}(\text{REDO}(T))$

Execute multiple times the undo and redo operations and I will always obtain the same outcome.

Figure 38: esempio

- Types of Log Records

- Records relevant to transactional commands:
 - begin
 - commit Identity of the transaction
 - abort
- Records relevant to operations
 - insert
 - delete
 - update
- Records relevant to recovery actions
 - dump
 - checkpoint

Figure 39: esempio

- Records relevant to transactional commands:
 - $B(T)$, $C(T)$, $A(T)$
- Records relevant to operations
 - $I(T, O, AS)$, $D(T, O, BS)$, $U(T, O, BS, AS)$
- Records relevant to recovery actions
 - DUMP, CKPT(T_1, T_2, \dots, T_n)
- Record fields:
 - T : transaction identifier
 - O : object identifier
 - BS, AS : before state, after state

Figure 40: esempio

- Transactional Rules

- Write-Ahead-Log
 - * Devono essere scritti i *before state* prima dell'esecuzione di un'azione sul database
 - * L'azione può essere **undone** (riportata allo stato before)
- Commit Rule
 - * Devono essere scritti gli *after state* prima dell'esecuzione del commit

- * L'azione può essere **redo** (rieseguita nel caso non andasse a buon fine)

- **Scrittura nel database prima del commit**

- Il *Write-Ahead-Log* viene rispettato, vengono eseguiti degli update prima delle write operations.
- Non c'è bisogno di alcun supporto per la REDO transaction, questo perchè il commit viene eseguito come ultimo passo dopo i 2 update. Al massimo vengono rieseguite le UNDO per le update.
- Una volta che la trasazione esegue il commit so di certo che è andato tutto liscio.

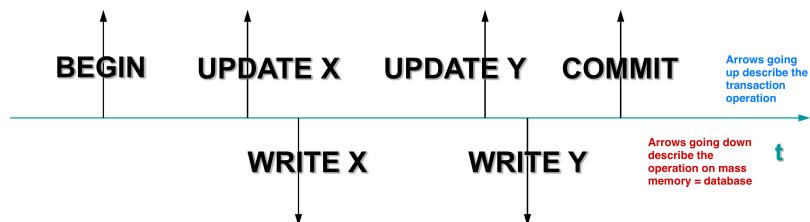


Figure 41: esempio

- **Scrittura nel database dopo il commit**

- Dopo che la transazione ha eseguito il commit, vengono eseguiti le write operations.
- Non c'è bisogno di aver alcun supporto per le UNDO transaction, questo perchè se la transazione abortisce non viene eseguito il commit (per cui nemmeno le scritture).

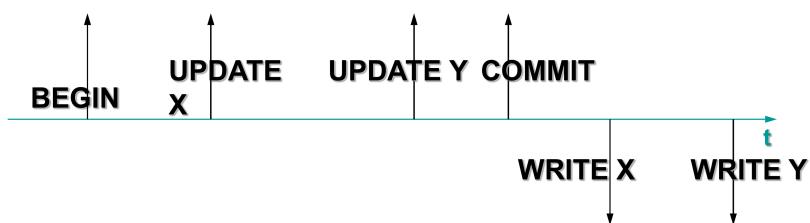


Figure 42: esempio

- Scrittura nel database in un tempo arbitrario

- Maggior flessibilità
- Dobbiamo considerare sia le UNOD che le REDO operations
- Consente di ottimizzare la gestione del buffer

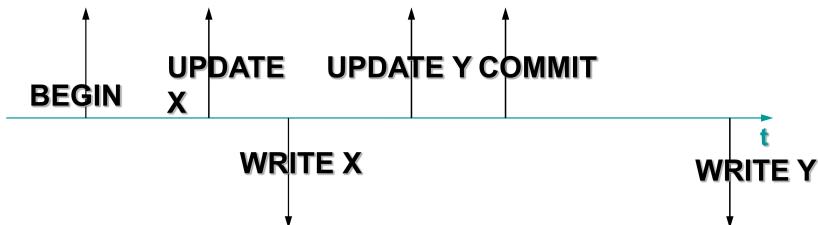


Figure 43: esempio

- **Fallimento** → *sistema smette di funzionare*

- **Soft failure**
 - * Perdita del contenuto della memoria centrale
 - * Necessita di **warm restart**
 - * Sono tutte quelle transazioni che sono state eseguite ma non hanno ancora raggiunto il commit e il sistema smette di funzionare
 - * Nessun danno alla memoria di massa
- **Hard failure**
 - * Fallimento sulla memoria di massa (database?)
 - * Si danneggia la memoria di massa
 - * Necessita di **cold restart**

- **Checkpoint**

- Si tratta di un punto consistente
- Tutte le transazioni fino a quel punto sono state completate correttamente dal sistema
- Le operazioni delle transazioni sono state scritte dal buffer al database
- Le transazioni sono state annotate all'interno del log

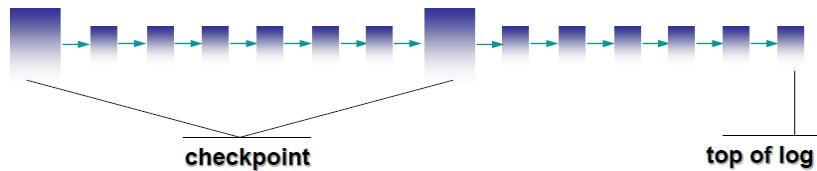


Figure 44: Checkpoint

- **Dirty pages:** le pagine che vengono lette e poi modificate nella cache del buffer, quindi sono diverse da quelle che risiedono all'interno del database.

- **Gestione dei checkpoint**

- Ci sono numerose possibilità, la più semplice:
 1. L'accettazione delle richieste di commit sono sospese
 2. Tutte le *dirty pages* scritte dai commit delle transazioni sono trasferite tramite **FORCE** al database
 3. L'identificativo delle *transazioni in corso* sono annotate nel log tramite **FORCE**, nessuna nuova transazione può iniziare durante l'annotazione
 4. L'accettazione viene ripresa
- Tutte le transazioni committed sono state scritte nel database
- *le transazioni “half-way” sono elencate nel checkpoint*

- **Dump**

- Si tratta di un punto nel tempo (Time point) nel quale si è effettuata una copia completa del database
- solitamente viene effettuato durante la notte o nel weekend (basso flusso di richieste)
- Il dump una volta completato viene annotato

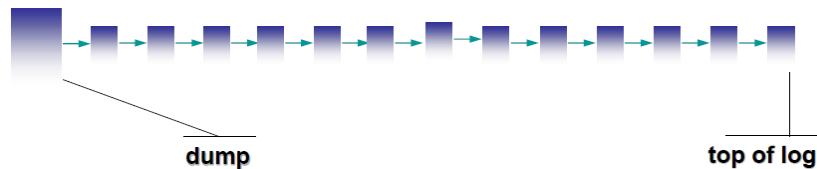


Figure 45: Dump

- **Warm restart**

- I record del log sono letti a partire dal checkpoint
- Le transazioni si suddividono in
 - * UNDO set
 - * REDO set
- Le azioni UNDO e REDO vengono eseguite

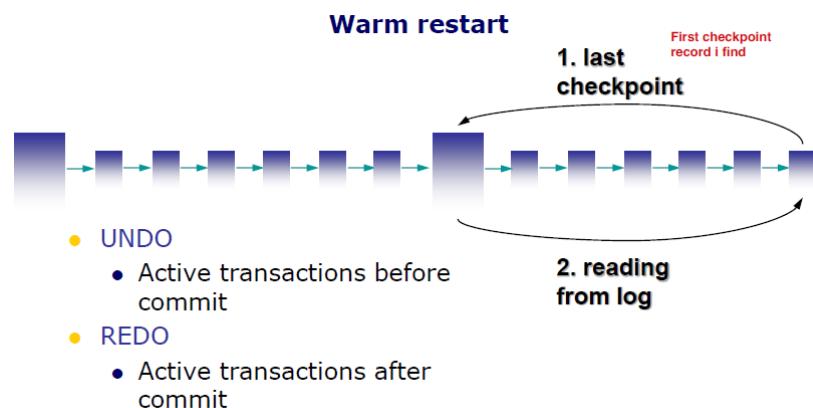


Figure 46: Esempio

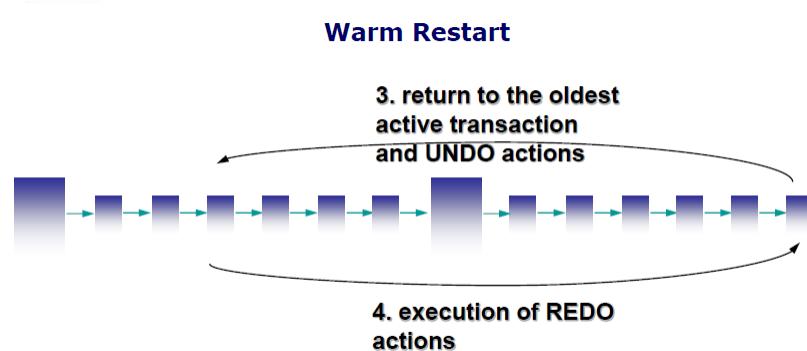


Figure 47: Esempio

- Example of Warm Restart

- Dobbiamo individuare il *checkpoint* dopo che avvive un *failure*
- Una volta individuato rieseguiamo al contrario le operazioni andando a recuperare il loro *BEFORE STATE* nel log
- Una volta completato tutto fino al checkpoint si va dopo il checkpoint

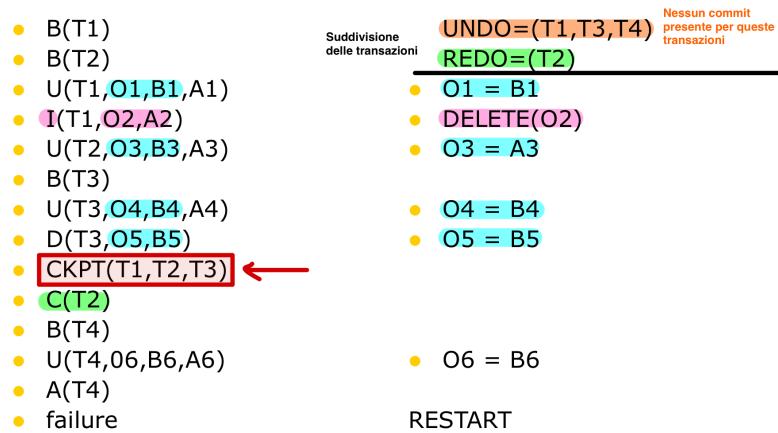


Figure 48: Esempio

- Cold Restart

- I dati sono salvati a partire dal backup
- Le operazioni registrate sul log fino al guasto sono eseguite
- Viene eseguito un riavvio a caldo

4 Commit Protocols

- Una transazione distribuita include una o più istruzioni che, singolarmente o come gruppo, aggiornano i dati su due o più nodi distinti di un database distribuito.
- I vari nodi sono tra loro separati e sono connessi sulla rete
- ACID è più comunemente associato alle transazioni su un singolo server di database, ma le transazioni distribuite estendono tale garanzia su più database. ACID comprende:
 - Atomicity
 - Consistency
 - Isolation
 - Durability

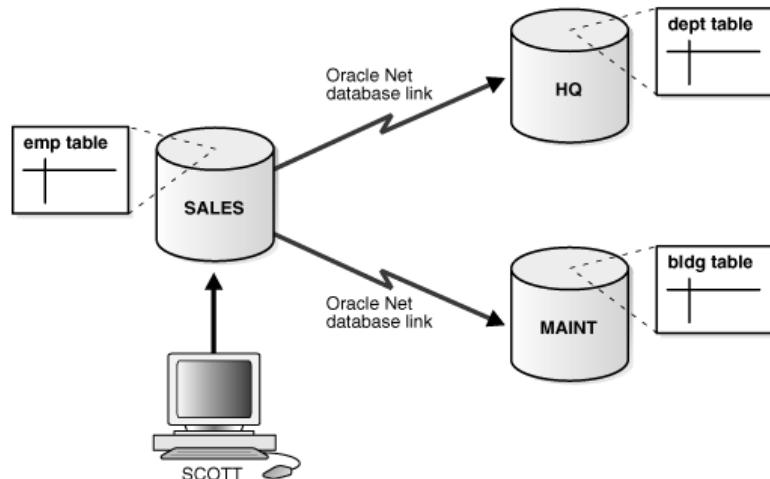


Figure 49: Esempio

- **Faults in a Distributed System**
 - I nodi falliscono
 - I messaggi vanno perduti sulla rete
 - A causa della network partitioning



Figure 50: Esempio

- Si tratta di un'architettura TCP-IP

- utilizzo protocollo tcp tra i vari nodi

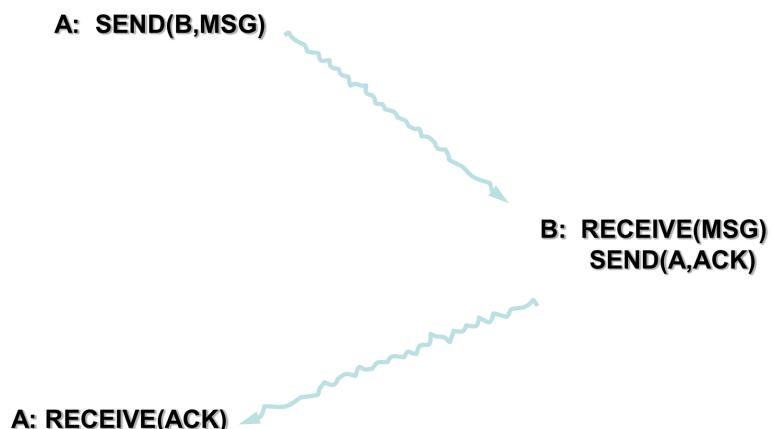


Figure 51: Esempio

- Two-phase Commit Protocol

- Protocollo che garantisce l'**atomicità** delle transazioni distribuite
- Protagonisti
 - * Il coordinatore (Transaction Manager, TM)
 - * I partecipanti (Resource Manager, RM)
- Il protocollo funziona come un matrimonio
 - * Phase one: la decisione è dichiarata
 - * Phase two: il matrimonio è effettuato

- **New Log Records**

- Nel log del coordinatore
 - * **prepare**: l'identità dei partecipanti
 - * **global commit/abort**: decisione
 - * **complete**: dichiara la fine del protocollo (tutti i partecipanti vengono informati)
- Nel log dei partecipanti
 - * **ready**: disponibilità a partecipare al commit
 - * **local commit/abort**: decisione ricevuta

- **Diagram of the Two-phase Commit Protocol**

- La transazione viene eseguita in un sistema distribuito, basato sul protocollo *tcp*
- **PREPARE**
 - * identità di tutti i partecipanti
 - * *prepare* message chiede ai partecipanti se accettano il commit della transazione
- **READY**
 - * viene scritto un *ready* message nel log dei partecipanti
- **MSG**
 - * contiene l'*agreement* dei partecipanti
- **GLOBAL DECISION**
 - * Se tutti i partecipanti hanno dato *l'agree* l'azione finale è un *commit*
 - * Mentre se anche uno solo dei partecipanti non risponde al messaggio, oppure non è d'accordo, allora l'azione finale è un *abort*
- **DECISION**
 - * la decisione presa dal coordinatore viene inviata a tutti i partecipanti
- **LOCAL DECISION**
 - * La decisione globale viene salvata nel log locale di ogni partecipante
- **ACK**
 - * Per confermare da parte dei partecipanti la scrittura nel log
 - * Stiamo utilizzando *tcp come lvl. di trasporto*

- COMPLETE

- * Termine della procedura

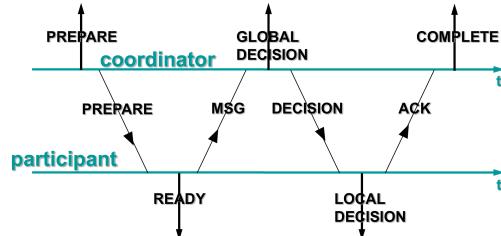


Figure 52: Two-phase Commit Protocol

- Protocol with Time-out and Window of Uncertainty

- timeout

- * viene gestito dal coordinatore in modo da ricevere tutti i messaggi dei partecipanti
 - siamo su una rete, i partecipanti possono essere dislocati in punti diversi, quindi avranno tempi diversi di invio e ricezione dei messaggi

- window of uncertainty

- * Periodo dal *READY* a *LOCAL DECISION*
- * In questo periodo il *resource manager / partecipanti* non conoscono il risultato della transazione

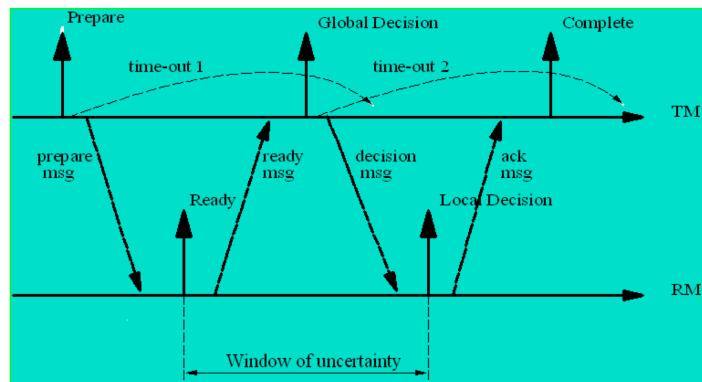
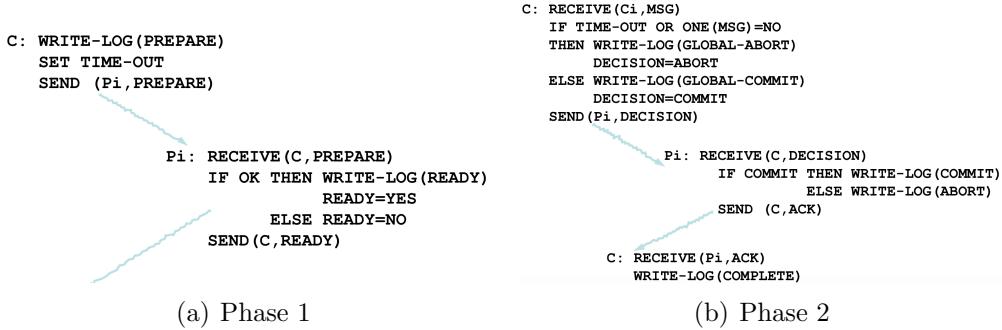


Figure 53: Time-out and Window of Uncertainty

- Fasi del Two-phase Commit Protocol



- Complessità del protocollo

- Il coordinatore può fallire
- Possono fallire i partecipanti
- I messaggi possono andare persi nella rete

- Recovery of participants

- Viene eseguito un **warm restart**. Dipende dall'ultimo record scritto nel log
 - * se è *abort* l'azione è *undone*
 - * se è *commit* l'azione è *redone*
 - * se è *ready* è fallito durante il two phase protocol, i partecipanti sono in *dubbio* sul risultato della transazione
 - viene contattato il *TM* per avere delle indicazioni sul risultato della transazione → *remote recovery requested*
 - questo avviene nella *window of uncertainty*

- Recovery of the Coordinator

- Quando l'ultimo record nel log è *prepare*, allora alcuni *RMs* possono essere bloccati nella fase iniziale del protocollo.
- Abbiamo 2 possibili opzioni di recovery
 1. Viene scritto *golbal abort* nel log e viene fatta proseguire la seconda fase del *two phase protocol*
 2. Altrimenti viene ripetuta la prima fase del protocollo cercando di ottenere un *global commit*

- Quando invece l'ultimo record nel log del TM è *una global decision*, alcuni *RMs* potrebbero essere bloccati nel passo finale. Il *TM* ripete la seconda fase del protocollo

- **Message Loss and Network Partitioning**

- La perdita di un *prepare msg* o di un *ready msg* è indistinguibile dal *TM*.
 - * In entrambi i casi viene superato il *timeout* e viene eseguita una *global abort*
- La perdita di un *decision msg* o di un *ack msg* è indistinguibile dal *TM*.
 - * In entrambi i casi viene superato il *timeout*, viene ripetuta la seconda fase del protocollo

- **Presumed Abort Protocol**

- Un'ottimizzazione utilizzata da molti DBMSs
 - * Se il *TM* riceve una *remote recovery* da parte di un *RM* (dubbio) e il *TM* non conosce il risultato della transazione risponde per **default global abort**
 - * Come conseguenza di questo approccio non è necessario scrivere in maniera sincrona (**force**) il *prepare* e il *global abort*
- il *complete* record può essere rimosso
- i record che devono essere **force** sono:
 - * *ready*
 - * *global commit*
 - * *local commit*

- **Read-only Optimization**

- Quando si scopre che un partecipante ha eseguito solo operazioni di lettura (nessuna operazione di scrittura)
 - * Risponde al *prepare msg* con **read-only** e sospende l'esecuzione del protocollo two-phase
 - * Il *TM* ignora tutte le operazioni **read-only** nella seconda fase

- **Blocking, Uncertainty, Recovery Protocols**

- Quando un *RM* si trova in *ready state*, deve aspettare la *global decision* del *TM*.
- Un errore di sistema da perte del *TM* lascerebbe in una situazione di stallo gli *RMs*
- Le risorse acquisite rimangono bloccate a causa del *lock* su di esse
- L'intervallo tra il *RMs log ready* e il *write local decision* viene chiamata → **window of uncertainty**
 - * Il protocollo deve mantere questo intervallo al minimo
- Nel caso di un errore di sistema da parte del *TM* o *RM* viene fatto il recovery in base alla decisione globale effettuata

- **Four-phase Commit Protocol**

- Il processo del **TM** viene replicato con un processo di backup su un'altro nodo
 - * Il *TM* principale informa prima la copia della decisione presa per poi comunicarla ai *RMs*
- In questo modo se dovesse presentarsi un problema sul *TM principale* viene utilizzato il backup
 - * Quando il *backup* diventa *TM*, viene attivato un altro processo di backup su un'altro nodo
 - * Per continuare l'esecuzione del protocollo

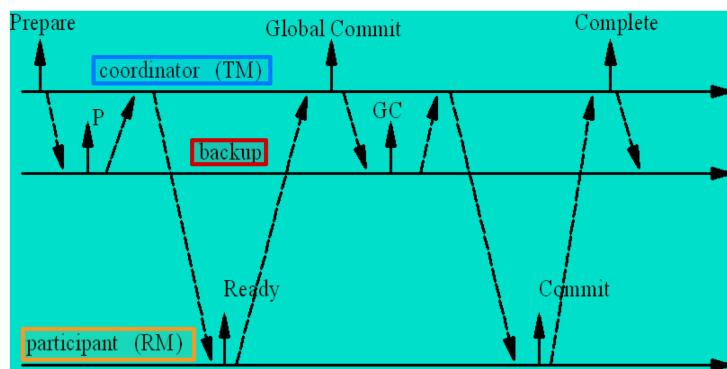


Figure 54: Diagram of the Four-phase Commit Protocol

- **Three-phase Commit Protocol**

- Grazie al *three-phase commit protocol*, ogni participant-*RMs* può diventare *TM*
- Quando viene **eletto** il nuovo *TM* si guarda l'ultimo record nel log
 - * se l'ultimo record è **ready**, allora può essere imposto un **global abort**
 - * se l'ultimo record è **pre-commit**, allora può essere imposto un **global commit**
- Questo protocollo non viene utilizzato per 2 problemi principali
 - * la *window of uncertainty* è più lunga
 - * se abbiamo una malfunzione all'interno della rete e si possiede più *RMs* all'interno della rete dislocati in diversi punti, **ci può essere un'elezione multipla**. Verrebbero eletti più *TMs* e potrebbero svilupparsi diverse esecuzioni per la stessa transazione

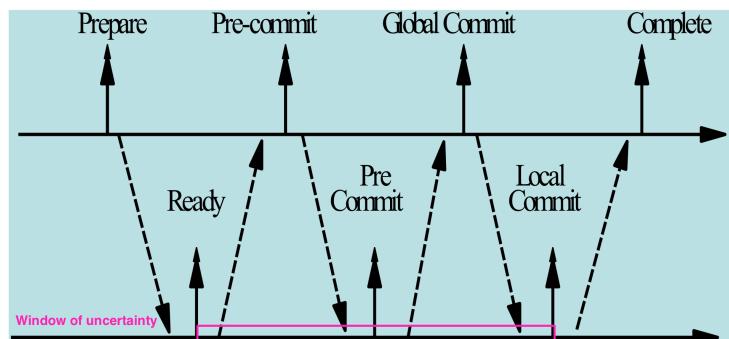


Figure 55: Diagram of the Three-phase Commit Protocol

- **Paxos Commit Protocol**

- Combina un protocollo di consenso con un commit protocol per garantire una decisione anche in presenza di partizioni
- **Paxos consensus protocol**
 - * Una rete che non può fallire più di F volte
 - * Stabilire un iniziatore
 - * Stabilire $(2 \cdot F + 1)$ accettatori

- * Il protocollo garantisce **consenso** con una maggioranza di $(F + 1)$ accettatori
- * Utilizza il *three-phase protocol*
- **Paxos commit protocol**
 - * Seleziona alcuni *RMs* (tra N) come accettatori
 - * Garantisce decisioni di commit/rollback con $(N \cdot (F + 3) - 3)$ messaggi
- Gli accettatori devono essere coordinati
- La maggior parte dei messaggi vengono salvati grazie alla promozione dei *RM* ad accettatori
- Il three-phase protocol è un caso particolare di paxos commit con $F = 0$ (coordinatori = accettatori)

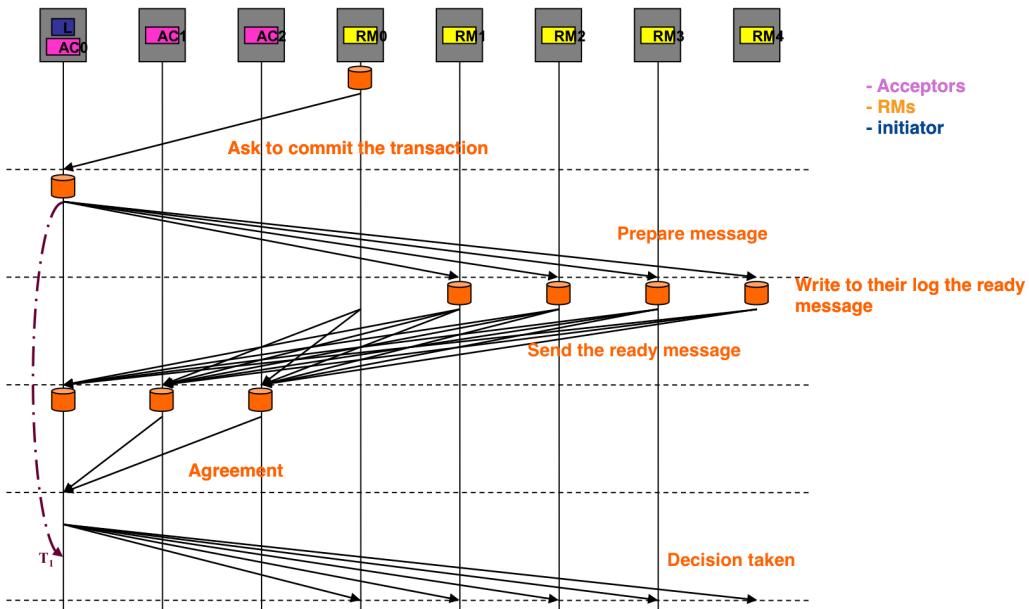


Figure 56: Paxos Commit (base)

5 Distributed Databases

- Client-Server Paradigm
 - Sono compresi 2 sistemi
 - * **Client:** invoca i servizi
 - * **Server:** fornisce il servizio invocato
 - Un'interfaccia di servizio è pubblicato dal server

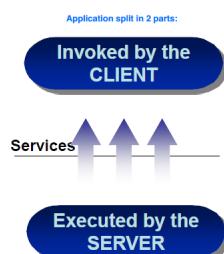


Figure 57: client - server

- Client-Server in Information Systems
 - Vi è una separazione
 - * **Client:** il livello di presentazione
 - * **Server:** gestione dei dati
 - Esempio: SQL
 - * Client: formula le query e mostra i risultati
 - * Server: esegue le query e calcola i risultati
 - * Network: sferisce i comandi e i risultati delle query

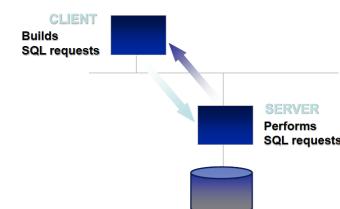


Figure 58: Traditional Client-Server architecture

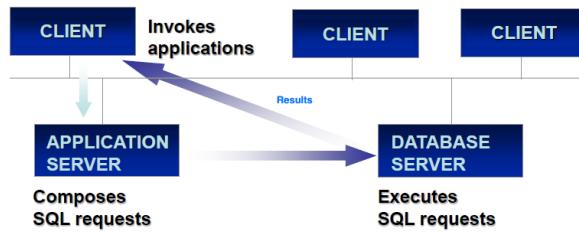


Figure 59: Application Server Architecture

- Processamento delle richieste
 - Le richieste da parte del client sono inserite in una **coda**
 - Il dispatcher si occupa di indirizzare le richieste verso il software service appropriato
 - Le richieste processate che hanno generato un messaggio vengono inserite in un coda di uscita e verranno poi indirizzate verso il client
 - Tutto questo è possibile attraverso il **parallelismo e il multiprocesssing**
- Data distribution
 - Non solo molti database ma anche applicazioni che utilizzano i dati da differenti data sources
 - **Distributed database**

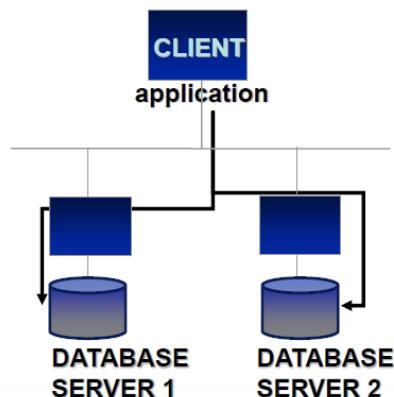


Figure 60: Distributed Database

- Distributed Database Types
 - classificazione in base all rete
 - * LAN (Local Area Network)
 - * WAN (Wide Area Network)
 - classificazione in base al database utilizzato
 - * Homogeneous system: tutti hanno lo stesso DBMS
 - * Heterogeneous system: DBMS diversi

- Problemi database distribuiti

- Indipendenza e cooperazione
- Trasparenza
- Efficienza
- Affidabilità

- **Indipendenza e cooperazione**

- I bisogni di indipendenza sono guidati da:
 - * Portare conoscenza e controllo locale nel luogo dove i dati vengono prodotti, utilizzati e gestiti
 - * Localizzare la maggior parte dei flussi di dati e dare autonomia locale di processamento
- I bisogni di cooperazione sono guidati da:
 - * Grandezza della compagnia o business-to-business applications
 - * Integrare diversi bisogni in un singolo bisogno di alto livello

- **La frammentazione dei dati**

- Decomposizione delle tabelle per permettere la loro distribuzione
- Proprietà
 - * **Completezza:** ogni data item di una tabella T deve essere presente in **uno dei suoi frammenti** T_i
 - * **Ricostruzione:** dall'unione di tutti i frammenti T_i otteniamo T

- **Frammentazione orizzontale**
 - * **Frammenti:** set di tuple
 - * **Completezza:** disponibilità di tutte le tuple
 - * **Ricostruzione:** UNION



Figure 61: Horizontal Fragmentation

- **Frammentazione verticale**
 - * **Frammenti:** set di attributi
 - * **Completezza:** disponibilità di tutti gli attributi
 - * **Ricostruzione:** JOIN sulla chiave

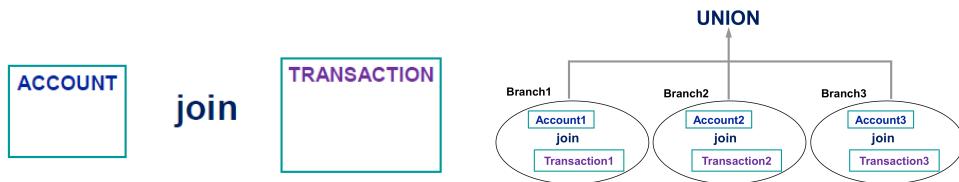


Figure 62: Vertical Fragmentation

- *Esempio - Fragment Allocation : pag 20 - 29*

- **Distributed Join**

- È l'operazione di analisi dei dati distribuiti più costosa
- Esempio



- I domini degli attributi devono essere partizionati e ogni partizione deve essere assegnata ad un paio di frammenti

Example: for numeric values between 1 and 30,000:

- Partition 1 to 10,000
- Partition 10,001 to 20,000
- Partition 20,001 to 30,000

Figure 63: Esempio

- **Esempi problematici**

- customers with more than one account
 - * self join su account
 - * Il self join non è quello usato per guidare la frammentazione
 - * Le coppie di account corrispondenti possono trovarsi su qualsiasi nodo

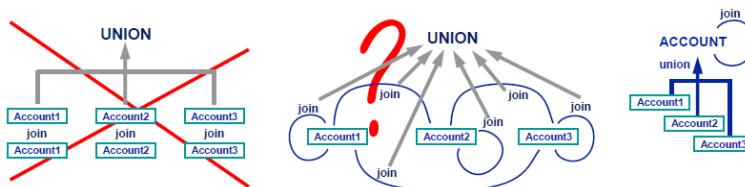


Figure 64: Esempio

- **Livello di trasparenza**

- Diversi modi di comporre le query, offerte dai database commerciali

- Ci sono 3 livelli significativi di trasparenza
 - * Trasparenza di frammentazione
 - * Trasparenza di allocazione
 - * Trasparenza di linguaggio
- In assenza di trasparenza, ogni DBMS accetta il proprio *dialetto SQL*
- Se il sistema è eterogeneo
 - * Non supporta nessun livello di trasparenza
 - * Non supporta un standard di interoperabilità comune

- **Problema di progettazione della distribuzione**

- Determinare la miglior frammentazione e allocazione delle tabelle
- L'allocazione dovrebbe fornire il grado ideale di ridondanza
 - * La ridondanza accelera il recupero e rallenta gli aggiornamenti
 - * La ridondanza aumenta la disponibilità e la robustezza
- Il monitoraggio delle query aiuta ad identificare la miglior scelta

- **Efficienza**

- Ottimizzazione delle query
- Tempo di esecuzione
 - * Esecuzione seriale
 - * Esecuzione parallela

- **Serial execution**

- Se devo accedere a n nodi, l'accesso viene eseguito in maniera seriale, ossia uno per volta
- In questo modo se le informazioni sono nei nodi iniziali non devo eseguire l'accesso sui nodi rimanenti
- Nel caso peggiore devo effettuare l'accesso su tutti

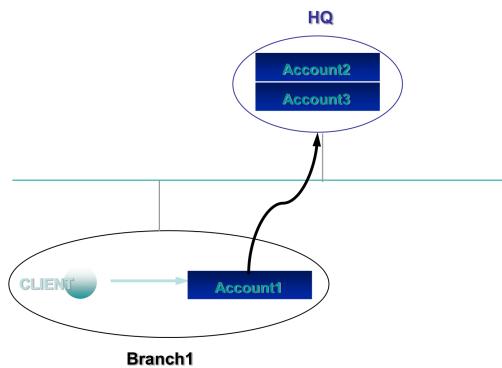


Figure 65: Serial execution, se non trovo le info in Branch1 mi sposto su HQ

- **Parallel Execution**

- Se non possiedo le informazioni sull'architettura del database distribuito adotto questa tipologia di esecuzione
- **Il client manda concorrentemente la stessa subquery a tutti i nodi e ne colleziona i risultati**

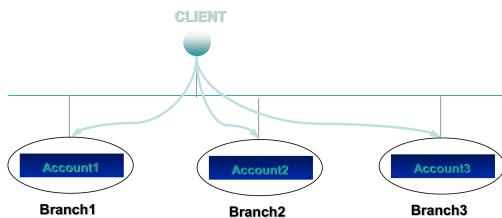


Figure 66: Parallel Execution

- **Distributed optimization with negotiation**

- Questo approccio non è reale

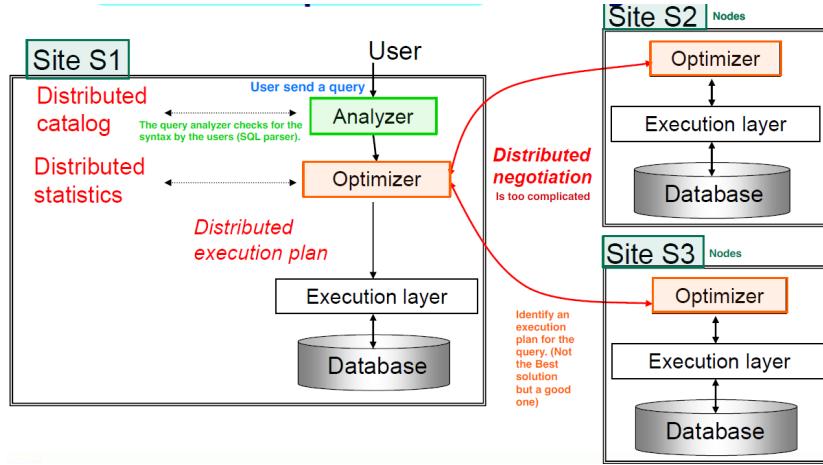
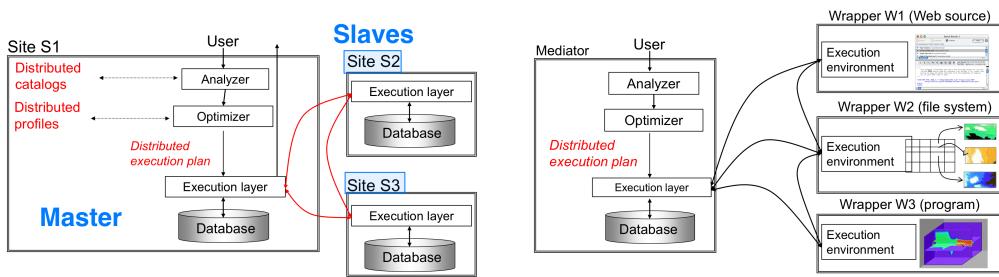


Figure 67: schema



(a) Distributed database with master-slave (b) Distributed system with mediator and optimizers

• Legacy systems

- Sistema architettonico basato su **mainframes** (potenti computer centralizzati)
- I client consistono in terminali
- In molti casi non c'è codice sorgente e documentazioni → i mainframe sono molto difficili da cambiare
- Però portano alte performance
- Tipicamente sono sistemi obsoleti, ma si occupano di gestire applicazioni importanti (grandi applicazioni bancarie,)
- I sistemi legacy sono comunque affidabili quando si tratta di eseguire operazioni continue 24/7
- *Non supportano le applicazioni, ma solo batch applications*

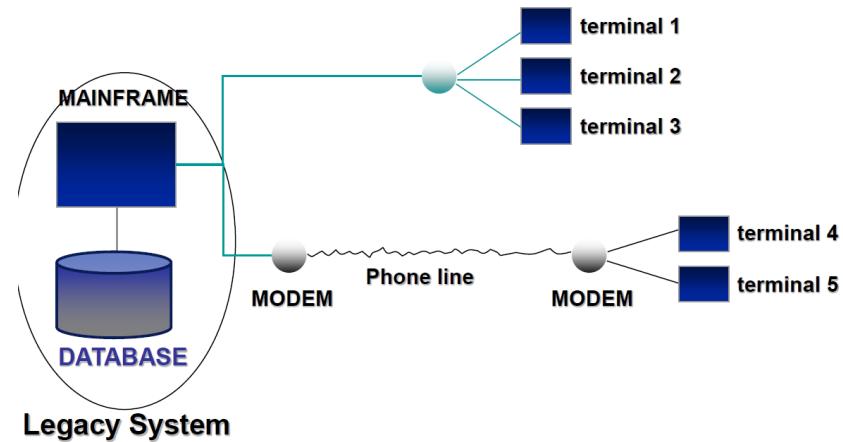


Figure 68: Legacy Systems

- **Gateway (Wrapper)**

- Si tratta di un software system che è in grado di trasferire un richiesta effettuata in input verso un output
- Fornisce capacità server verso gli input system e capacità client verso gli output system → **proxy**
- Esegue la conversione necessaria per i diversi **formati e linguaggi** dei due contesti
- Posizione
 - * in mezzo ai sistemi di transazione
 - * davanti al mainframe

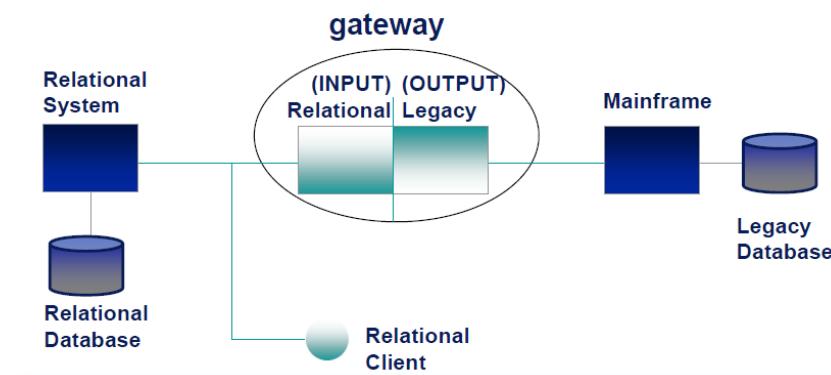


Figure 69: Usage of Gateway Systems

- **ACID vs BASE**

- Per sistemi su larga scala viene utilizzato l'approccio **BASE**
 - * **Basic**
 - * **Availability**
 - * **Soft state**
 - * **Eventual consistency**
- **Consistenza** → le query possono portare a risultati diversi se nello stesso tempo esistono problemi di rete nelle reti
- *Sono più importanti le performance rispetto a ACID*

6 Parallel and Replicated Databases

- Utilizzo del parallelismo all'interno dei server
 - Macchine con multiprocessori
 - Calcolo identico su ogni processore
 - Obiettivo: incrementare le performances

➔ PARALLEL DATABASE

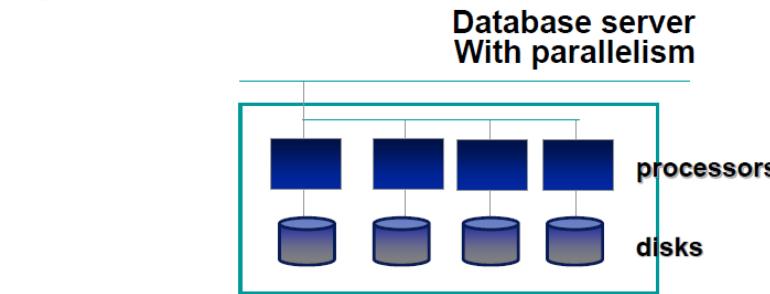


Figure 70: PARALLEL DATABASE

- Confronto architetture

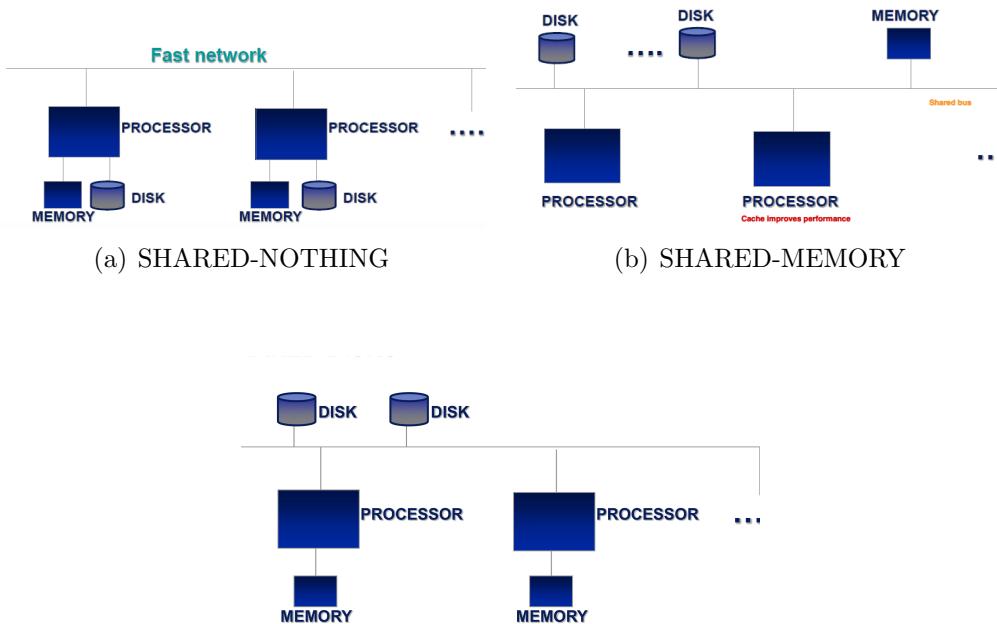


Figure 71: SHARED-DISKS

- **Application scalability**

- **Carico di lavoro / Load:** set di tutte le applicazioni (queries)
- **Scalabilità:** capacità del sistema di incrementare le performance sotto un incremento del carico di lavoro
- **All'aumentare delle dimensioni del carico**
 - * incrementato delle queries da svolgere
 - * Complessità delle queries

- **Two Load Types**

- **Transactional**
 - * *Carico:* transazioni correttamente
 - * *Sistema di misura:* tps (transazioni per secondo)
 - * *Tempo di risposta:* alcuni secondi
- **Data analysis**
 - * *Carico:* query SQL complesse
 - * *Tempo di risposta:* variabile

- **Parallelismo**

- Viene ottenuto grazie alla cooperazioni dei processori che vengono installati su una singola architettura
- Esistono due tipi di parallelismo
 - * **Inter-query**
 - Ogni query viene eseguita da un singolo processore (*per le transactional loads*)
 - * **Intra-query**
 - Ogni query viene eseguita da diversi processori (*per le data-analysis loads*)

- **Benchmark**

- Si tratta di un metodo per confrontare due o più sistemi in competizione tra loro
- Standardizzazione
 - * Del database
 - * Del carico

- Codice delle transazioni
- Transmissione
- Frequenza
- * Delle codinzioni di misura
- Diffenrenti tipi di carico
 - * **Tpc-a:** transactional
 - * **Tpc-b:** mixed
 - * **Tpc-c:** data analysis
- **Speed-up curve**
 - Misura l'incremento dell'Efficienza in relazione al numero di processori

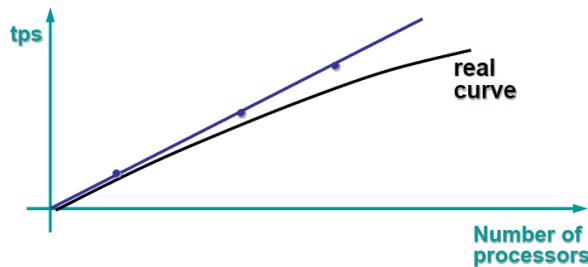


Figure 72: Speed-up curve

- **Scale-up curve**
 - Misura il costo totale / complessità computazionale per transazione in relazione al numero di processori

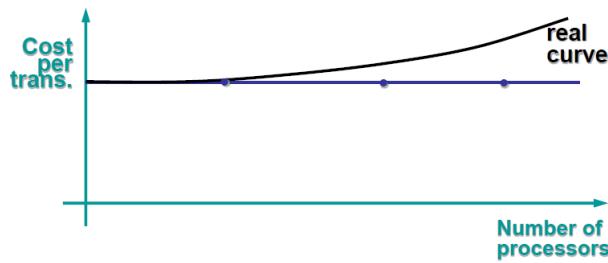


Figure 73: Scale-up curve

- **Data Replication**

- Questo per
 - * *Maggior disponibilità*: continuità del servizio
 - * *Maggior efficienza*: CDN
 - * *Maggior affidabilità*: no perdita dati

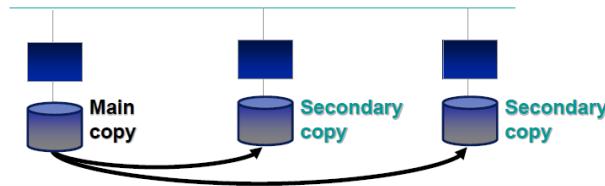
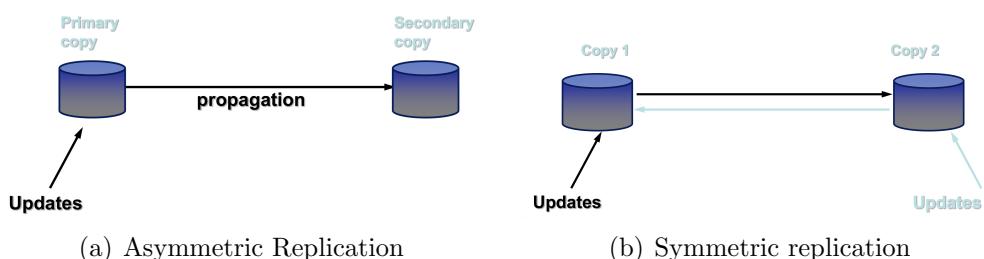


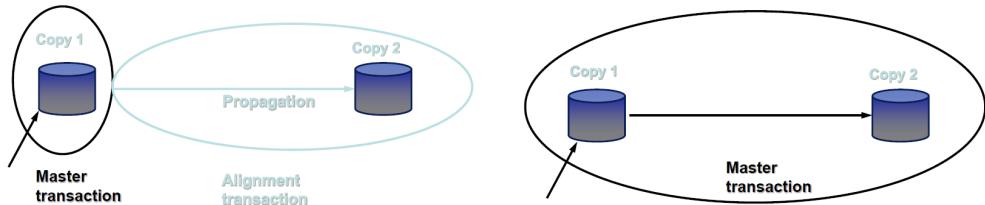
Figure 74: Data Replication

- **Replication Methods**

- ***Asymmetric Replication***
 - * Architettura master-slave
 - * I nodi hanno diversi ruoli
 - * Esistono diverse copie di backup dati
 - La principale
 - Le secondarie
 - * SE il sistema gestisce un numero di update molto grande si genera *bottle neck*
- ***Symmetric replication***
 - * I nodi hanno lo stesso ruolo
 - * Competizione è ridotta



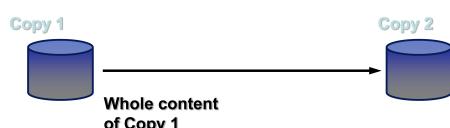
- Transmission of Updates



(c) Asynchronous transmission: main solution
(d) Synchronous transmission: too expensive

- Alignment Techniques

- Possono essere
 - * Periodici
 - * Tramite comando
 - Il trasferimento dei nuovi valori sulle copie secondarie avvive tramite amministratore
 - * Su accumulo di aggiornamenti
- **Refresh:** si trasferiscono l'intera copia
- **Incremental:** si trasferiscono solo le differenze



(e) Refresh



(f) Incremental

- Replication Mechanisms

- asymmetric, asynchronous, incremental
- **Product:** replication manager
- **2 Moduli:** tramite trigger
 - * Capture
 - * Apply
- Catturare la variazione all'interno delle tabelle in modo trasparente → **trigger**

7 Data Warehouses

- DATA WAREHOUSE

- Si intende in generale una collezione o aggregazione di dati strutturati, provenienti da fonti interne operazionali (DBMS) ed esterne al sistema informativo aziendale, utili ad analisi e rapporti informativi

- ON-LINE ANALYTICAL PROCESSING (OLAP)

- Il nome dato alle attività di analisi (è contrapposto a On Line Transaction Processing, OLTP)
- Designa un insieme di tecniche software per l’analisi interattiva e veloce di grandi quantità di dati, che è possibile esaminare in modalità piuttosto complesse. Questa è la componente tecnologica base del data warehouse
- Gli strumenti OLAP si differenziano dagli OLTP per il fatto che i primi hanno come obiettivo la performance nella ricerca e il raggiungimento di interrogazioni quanto più articolate sia possibile; i secondi, invece, mirano ad una garanzia di integrità e sicurezza delle transazioni.

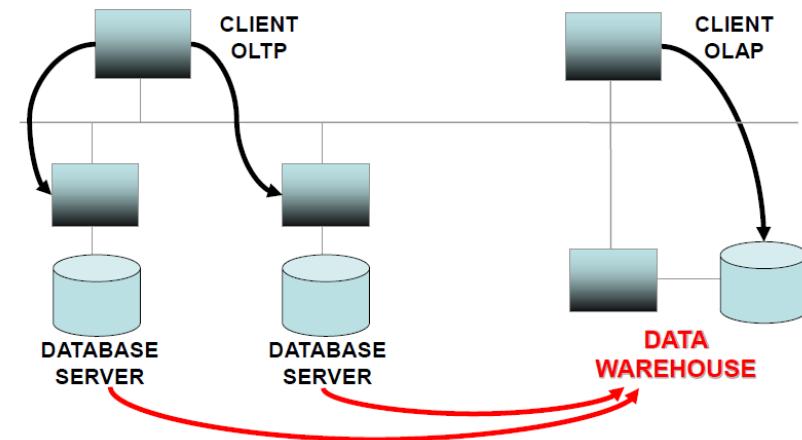


Figure 75: Interaction between OLTP and OLAP

- An Architecture for Data Warehousing

- *Data Mart*: è un raccoglitore di dati di un particolare topic

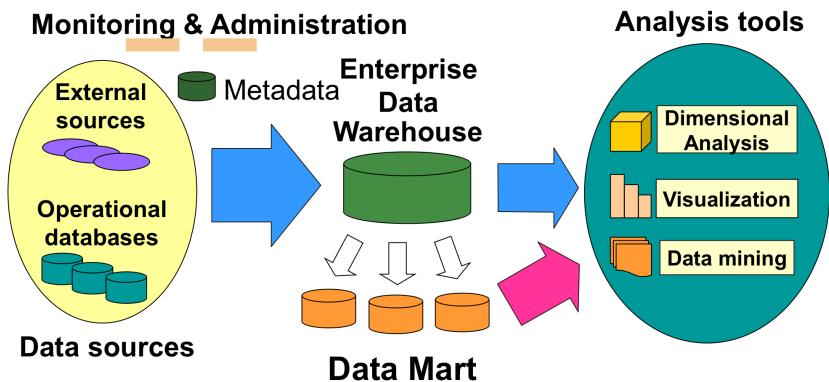


Figure 76: schema

- **Data Warehouse (DW) and Data Mart (DM)**
 - Una DW integra diversi DM
 - Gli utenti in genere si rivolgono a un Data Mart specifico
 - I DM condividono dati comuni
 - Ogni Data Mart è responsabile di un aspetto specifico di l'impresa aziendale
- **Star model or multi-dimensional schema**
 - Lo star model viene usato per ogni DM
 - Si tratta di un modello concettuale che impone alcune restrizioni
 - Vantaggi
 - * Disponibilità di interfacce di query specifiche adatte
 - * Buone performance
 - * Progettazione lineare dello schema relazionale
 - Concetti rilevanti
 - * ***fact***: un aspetto cruciale per l'analisi - cosa vogliamo analizzare
 - * ***measure***: una proprietà anatomica di un *fact*
 - * ***dimension***: una prospettiva specifica per l'analisi

- *Esempio - Retail shops:*

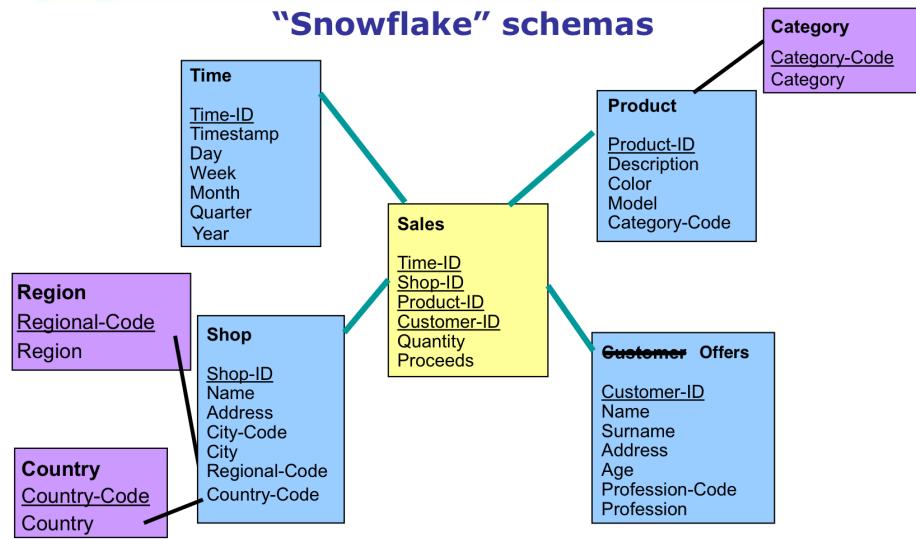
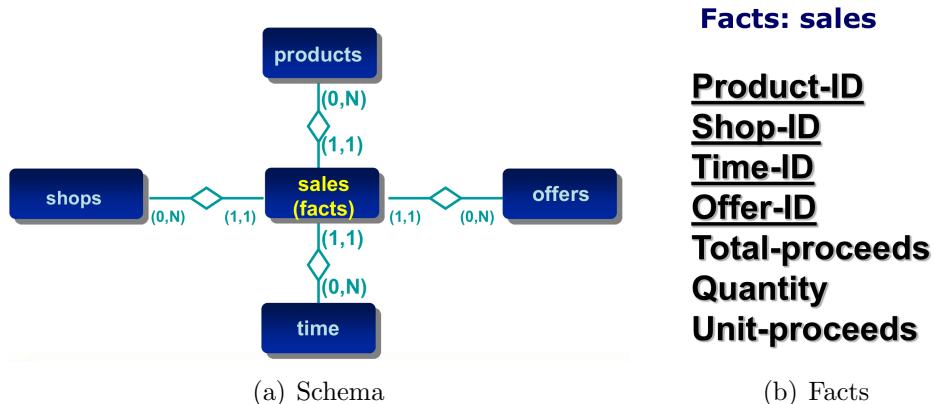


Figure 77: schema



– Considerazioni

- * Una tabella per ogni dimensione / entità
- * Tutti gli attributi della fact entity sono *measure*
- * *Fact: sales*
- * *Dimension: Product, Time, Shop, Offers*

- Multi-dimensional data representation

- * Prendiamo 3 dimensioni per semplicità di rappresentazione
- * **Fact:** sales
- * **Measure:** Quantity, Price
- * **Dimension:** Product, Time, Shop

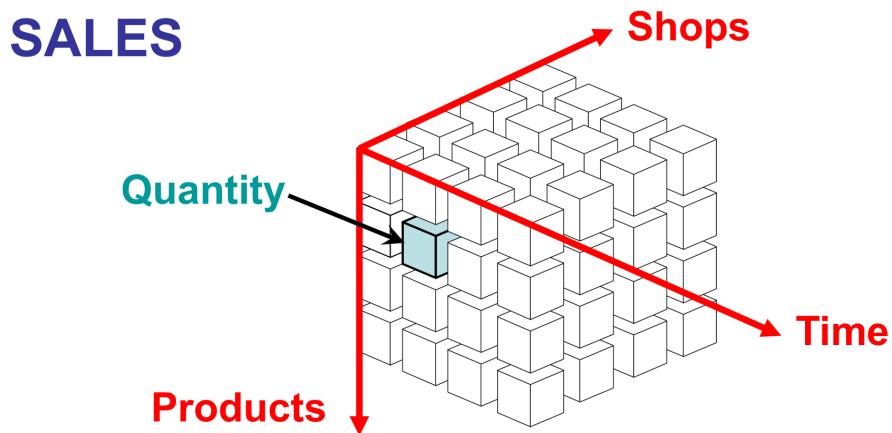


Figure 78: schema

- Il regional manager studia le *vendite* di **tutti** i *prodotti* durante **tutta la continuità** nel tempo rispetto a **un** determinato shop nella sua regione

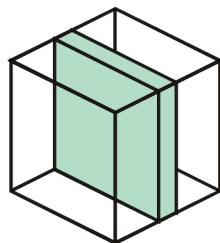


Figure 79: schema

- Il product manager studia le *vendite* di **un prodotto** durante **tutta la continuità nel tempo** rispetto a **tutti** i shop

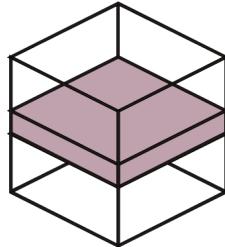


Figure 80: schema

- Il financial manager studia le *vendite* di **un prodotto** durante **tutta la continuità nel tempo** rispetto a **tutti** i shop

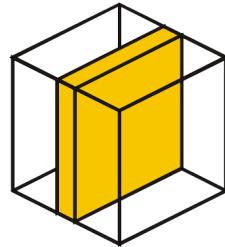


Figure 81: schema

- Il strategic manager studia le *vendite* di **una categoria** di *prodotti* durante **un determinato periodo** nel tempo rispetto a **determinati** shop

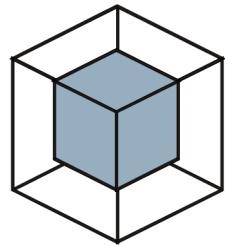


Figure 82: schema

- **Data Visualization**

- I dati sono renderizzati e sono visibili graficamente, in questo modo è più semplice da interpretare il risultato finale

- **Operations over multi-dimensional data**

- **Roll up:** aggrega i dati (riduce i dettagli)
 - * Rimuovendo una dimensione
- **Drill down:** disaggrega i dati (aggiunge dettagli)
 - * Aggiungendo una dimensione
- **Slice and dice:** selezione e proiezione
- **Pivot:** cambia l'orientamento del cubo di dati

Month	Product	Zone	Sum of quantity
February	pasta	north	15.000
February	pasta	east	17.000
February	pasta	west	13.000
March	pasta	north	18.000
March	pasta	east	18.000
March	pasta	west	14.000
April	pasta	north	18.000
April	pasta	east	17.000
April	pasta	west	16.000

(a) Drill Down: adding one dimension (Zone)

Product	Zone	Sum of quantity
pasta	north	51.000
pasta	east	52.000
pasta	west	43.000

(b) Roll-up: removing one dimension (Month)

- **Aggregates in SQL: data cube**

- Esprime tutte le possibili aggregazioni delle tuple di una tabella
- Viene utilizzato un nuovo valore polimorfico specifico per lo scopo → **ALL**

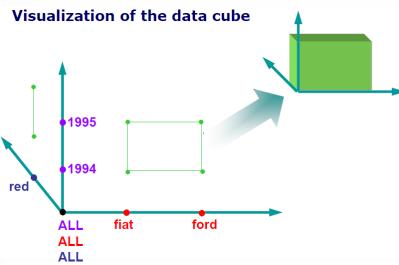


Figure 83: schema

model	year	color	sum (quantity)
fiat	1994	red	50
fiat	1995	red	85
fiat	1994	ALL	50
fiat	1995	ALL	85
fiat	ALL	red	135
fiat	ALL	ALL	135
ford	1994	red	80
ford	1994	ALL	80
ford	ALL	red	80
ford	ALL	ALL	80
ALL	1994	red	130
ALL	1995	red	85
ALL	ALL	red	215
ALL	1994	ALL	130
ALL	1995	ALL	85
ALL	ALL	ALL	215

select Model, Year,
 Color, sum(Quantity)
from Sales
where Model in ('Fiat', 'Ford')
 and Color = 'Red'
 and Year between 1994 and 1995
group by Model, Year, Color
with cube

(a) Data cube in SQL

(b) Data cube results

8 Active Databases

- Un database che supporta le **regole attive** (chiamate anche **trigger**)
- *Paradigma*
 - **Event:** quando l'evento si verifica
 - **Condition:** e la condizione è vera
 - **Action:** viene eseguita un'azione
- **Event**
 - Normalmente è una modifica dello stato del database
 - * *insert*
 - * *delete*
 - * *update*
 - quando l'evento si verifica il trigger è **attivato**
- **Condition**
 - Un predicato che identifica le situazioni in cui il è necessaria l'applicazione di un trigger
 - Quando la condizione viene valutata, il trigger è **considerato**
- **Action**
 - Un'istruzione di aggiornamento generica o una stored procedure

- Quando l'azione viene elaborata, il trigger è **eseguito**

- Triggers in SQL:1999, Syntax

- Ogni trigger è caratterizzato da:

- * nome
- * nome della tabella target monitorata
- * modalità di esecuzione: *before*, *after*
- * eventi monitorati: *insert*, *delete* or *update*
- * granularity (*statement-level* or *row-level*)
- * nomi e alias per i valori di transizione e tabelle di transizione
- * azioni
- * creazione timestamp

```
create trigger TriggerName
  <before | after >
  < insert | delete | update [of Column] > on Table
  [referencing
    <[old_table [as] OldTableAlias]
     [new_table [as] NewTableAlias] > | 
    <[old [row] [as] OldTupleName]
     [new [row] [as] NewTupleName] >]
  [for each < row | statement >]
  [when Condition]
SQLStatements
```

Optional
Different options

If you write a trigger without a condition this is
a execute everytime the events occur

Figure 84: Triggers in SQL:1999, Syntax

- tipi di eventi

- BEFORE

- * Il trigger è considerato e possibilmente eseguito *prima dell'evento*
- * I before triggers non possono cambiare lo stato del database
- * Possono però cambiare le variabili delle transazione nella modalità *row level mode*
 - set t.new = expression
- * Normalmente questa modalità viene utilizzata quando si vuole controllare una modifica prima che avvenga, e possibilmente apportare una modifica alla modifica stessa.

- AFTER
 - * Il trigger è considerato e possibilmente eseguito dopo l'evento

- Granularity of events

- Statement level mode (*for each statement option*) → di default
 - * Il trigger viene *considerato* ed eventualmente *eseguito solo una volta per ogni istruzione che lo ha attivato*, indipendentemente dal numero di tuple modificate
- Row-level mode (*for each row option*)
 - * Il trigger viene *considerato* ed eventualmente *eseguito una volta per ogni tupla modificata dall'istruzione*

- The *referencing* clause

- Dipende dalla granularità
 - * *For row-level mode*
 - Ci sono 2 *variabili* di transizione: **old**, **new**
 - Che rappresentano il valore prima e dopo la modifica della tupla in esame
 - * *For statement-level mode*
 - Ci sono 2 *tabelle* di transizione: **old table**, **new table**
 - Che contengono il vecchio o il nuovo valore di tutte le tuple modificate
- Le variabili **old** e **old table** non possono essere utilizzate con trigger che ha come evento **insert**
- Le variabili **new** e **new table** non possono essere utilizzate con trigger che ha come evento **delete**

- Execution of Multiple Triggers: Conflicts between Triggers

- Se diversi trigger sono associati allo stesso evento, in SQL 1999 viene eseguita la seguente POLICY
 - 1. *Before triggers (statement-level and row-level)* sono eseguiti
 - 2. La modifica viene applicata e vengono verificati i vincoli di integrità definiti sul DB
 - 3. *AFTER triggers (row-level and statement level)* sono eseguiti

- Se ci sono più trigger appartenenti allo stesso categoria, l'ordine di esecuzione scelto dal sistema si basa sulla loro definizione timestamp
- I trigger più vecchi hanno priorità maggiore

- **Recursive Execution Model**

- SQL 1999 afferma che i trigger vengono gestiti all'interno di un **Trigger Execution Context (TEC)**
- L'esecuzione di un trigger può produrre eventi che attivano altri trigger, che dovranno essere valutati in un nuovo TEC interno
 - * A questo punto viene salvato lo stato del **TEC esterno** che ha causato l'esecuzione del nuovo trigger
 - * E viene eseguito il **TEC interno** del nuovo trigger attivato
 - * Una volta conclusa l'esecuzione del **TEC interno** viene ripresa l'esecuzione del **TEC esterno**

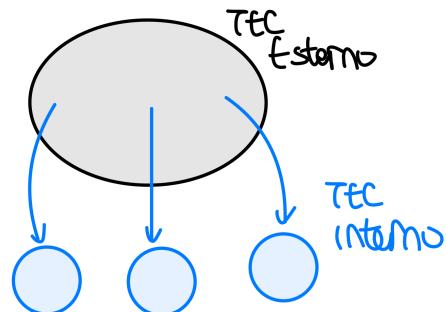


Figure 85: Recursive Execution Model

- L'esecuzione del trigger si interrompe dopo una data profondità di ricorsione, sollevando una “*nontermination exception*”
- Se si viene a verificare un qualsiasi guasto durante l'esecuzione a catena dei trigger attivati da S provoca:
 - * Il rollback parziale di S
 - * E di tutte le modifiche effettuate dai trigger della catena interni

- **Design - Trigger Properties**

- È importante garantire che le interferenze tra trigger e attivazioni a catena non producano anomalie nel comportamento del sistema
- Le 3 classiche proprietà:
 - * **Termination:**
 - per ogni stato iniziale e dopo una sequenza di modifiche, viene prodotto uno stato finale
 - nessun ciclo infinito di attivazioni
 - * **Confluence:**
 - i trigger terminano e producono uno stato finale unico, indipendente dall'ordine in cui i trigger vengono eseguiti
 - * **Determinism of observable behavior:**
 - i trigger sono confluenti e producono la stessa sequenza di messaggi
- La termination è la proprietà più importante

- **Termination Analysis**

- Per verificare la **termination** viene utilizzato il **triggering graph**
 - * Un nodo per ogni trigger
 - * Un arco dal trigger t_i al trigger t_j se l'esecuzione di t_i attiva t_j
- Se il grafo è *aciclico* il sistema garantisce la terminazione
- Se il grafo invece presenta *cicli* potrebbe non garantire la terminazione

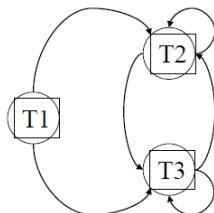


Figure 86: Il grafo è ciclico, ma continuando l'esecuzione raggiunge la termination

- **Esempio:**

- Ci sono 2 cicli ma il sistema garantisce la termination
 - * Questo perchè non un tot di cicli la condizione non viene più soddisfatta e non viene più eseguito
- per renderlo non-terminationg basterebbe porre una condizione che viene sempre soddisfatta

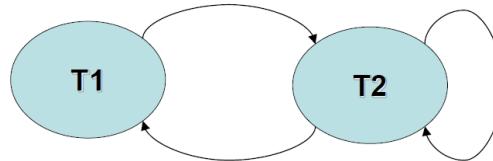
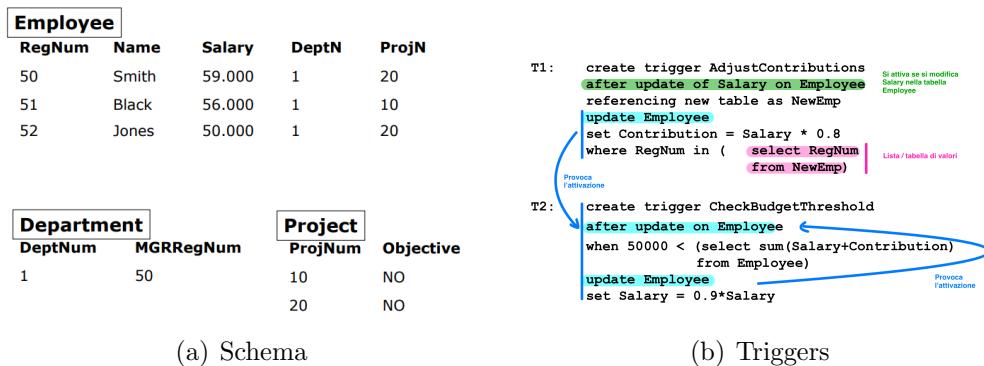


Figure 87: Triggering Graph for Previous Triggers

- **Applications of Active Databases**

- *Internal rules*
 - * Sono regole generate dal sistema e non visibili agli utenti
 - * Integrity constraint management
 - * Computation of derived and replicated data
 - * Versioning management, privacy, security
 - * Action logging, event recording
- *External rules*
 - * Sono regole generate dall'amministratore del sistema
 - * Personalization, adaptation
 - * Context-awareness
 - * Business rules

- Triggers for Materialized View Maintenance

- Coerenza delle views rispetto alle tabelle su cui si trovano definite
 - * Gli aggiornamenti della tabella di base devono essere propagati alle views
- La manutenzione delle Materialized views è gestita tramite triggers
- Replication management

```
CREATE MATERIALIZED VIEW EmployeeReplica
REFRESH FAST AS
SELECT * FROM
DBMaster.Employee@mastersite.world;
```

Figure 88: Esempio

- Recursion Management

- ricorsione non è ancora supportata sui moderni DBMSs
- Però vengono utilizzati i trigger per costruire e mantenere la gerarchia
 - * La gerarchia viene costruita su **SuperProduct** e **Level**
 - * Esempio: prodotti che non contengono altri prodotti
 - **SuperProduct = NULL**
 - **Level = 0**

```
Product(Code, Name, Description, SuperProduct, Level)
```

Figure 89: Esempio

- Execution Modes

- La modalità di esecuzione descrive la connessione tra *l'attivazione (evento)*, la *considerazione* e *l'esecuzione*
- La *condizione e l'azione di esecuzione* sono valutate insieme → il trigger è **Immediate**
- Le alternative sono

- * ***Deferred***

- Il trigger viene gestito alla fine della transazione

- * ***Detached***

- Il trigger viene gestito in una transazione separata

9 Physical data structures and query optimization

- DataBase Management System — DBMS

- Un sistema (software product) capace di gestire una collezione di dati che sono
 - * *large* (rispetto alla memoria centrale)
 - * *persisten* (con una vita che è indipendente da singole esecuzioni dei programmi che vi accedono)
 - * *shared* (in uso da diversi applicativi)
- Inoltre dev'essere
 - * *affidabile* (tolleranza a guasti software e hardware)
 - * e garantire *privacy/sicurezza*

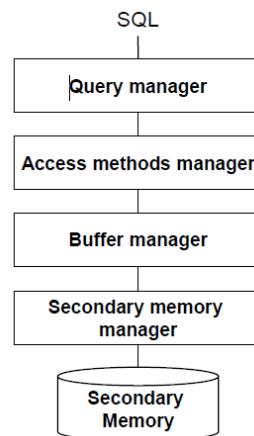


Figure 90: Access and query manager

- Main and Secondary memory

- I programmi possono fare riferimento solo ai dati memorizzati nella memoria principale
- I database devono essere archiviati (principalmente) nella memoria secondaria per 2 ragioni
 - * grandezza
 - * persistenza

- I dati archiviati nella memoria secondaria possono essere utilizzati solo se prima vengono trasferiti nella memoria principale
- I dispositivi di **memoria secondaria** sono organizzati in **blocchi** di **lunghezza fissa** (ordine di grandezza: pochi KB)
- Le uniche operazioni disponibili per tali dispositivi sono **leggere e scrivere una pagina**, ovvero un flusso di byte corrispondente a un blocco;
- Per comodità e semplicità, useremo **blocco e pagina come sinonimi**
- Accesso alla memoria secondaria
 - * **tempo di ricerca:** 10 - 50 ms (posizionamento della testa)
 - * **tempo di latenza:** 5 - 10 ms (rotazione del disco)
 - * **tempo di trasferimento:** 1 - 2 ms (trasferimento dati)
- Nelle operazioni di *I/O* il costo dipende esclusivamente dal numero di accessi alla memoria secondaria

• DBMS and file system

- Il file system **FS** è il componente del sistema operativo che si occupa di gestire l'accesso alla memoria secondaria
- I DBMSs utilizzano poche funzionalità del *FS*:
 - * creare ed eliminare file e per leggere e scrivere singoli blocchi o sequenze di blocchi consecutivi.
- Il DBMS gestisce direttamente l'organizzazione dei file, sia in termini di distribuzione dei record all'interno di blocchi e rispetto alla struttura interna di ogni blocco.
- Costruisce in tale spazio le strutture fisiche che sono implementate da **tabelle**
- *Un file è tipicamente dedicato a una singola tabella*, ma
 - * Può capitare che un file contenga dati appartenenti a più di una tabella e che le tuple di una tabella sono divisi in più di un file.

• Blocks and records

- I blocchi (i componenti "fisici" di un file) e i records (i componenti "logici") hanno dimensioni differenti
 - * La dimensione dei blocchi dipende dal file system

- * La dimensione dei record dipende dalle esigenze delle applicazioni ed è normalmente variabile all'interno di un file

- **Block Factor**

- Numero di record all'interno di un blocco
 - * S_R : Dimensione del record (costante per semplicità)
 - * S_B : Dimensione del blocco
 - * Se $S_B > S_R$ numero record all'interno di un blocco $\lfloor \frac{S_B}{S_R} \rfloor$ (ossia parte intera per difetto)
- Il resto rimante da $\lfloor \frac{S_B}{S_R} \rfloor$ può essere
 - * usato ("spanned" records)
 - * non usato ("unspanned" records)

- **Physical access structures**

- Utilizzato per l'archiviazione e la manipolazione efficienti dei dati all'interno del DBMS
- Ogni DBMS ha un numero definito e limitato di **access methods**
- **access methods** sono moduli software che forniscono accesso ai dati e primitive di manipolazione per ogni *struttura di accesso fisico*
- Considereremo 3 tipi di strutture
 - * **Sequential**
 - * **Hash-based**
 - * **Tree-based (or index-based)**

- **Organization of tuples within pages**

- Ogni metodo di accesso ha la propria organizzazione della pagina
- I metodi **Sequential** e **Hash-based** hanno la pagina in questo modo
 - * Una parte iniziale **block header** e una parte finale **block trailer**
 - contengono informazioni di controllo usati dal file system

- * Una parte iniziale ***page header*** e una parte finale ***page trailer***
 - contengono informazioni di controllo usati dal metodo di accesso
- * Un ***page dictionary***
 - che contiene puntatori a ciascun elemento di dati elementari utili contenuti nella pagina
- * Un ***useful part***
 - che contiene i dati. Il *page dictionary* e il *useful part* crescono come pile in versi opposti
- * Un ***checksum***
 - per verificare la validità dei dati

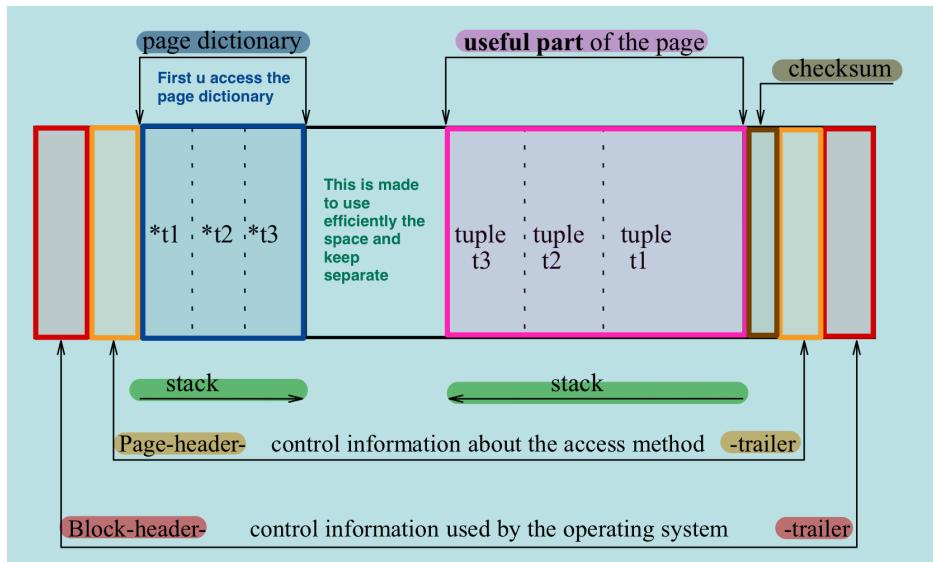


Figure 91: Organization of tuples within pages

- Page manager primitives
 - ***Insertion and update of a tuple***
 - * potrebbe richiedere una riorganizzazione della pagina se c'è spazio sufficiente per gestire i byte extra introdotti
 - ***Deletion of a tuple***
 - * spesso eseguita contrassegnando il flag all'interno del page dictionary, rispetto a quella determinata tupla come "non valida"

- * non c'è un vero e proprio elimina
- ***Access to a field of a particular tuple***
 - * viene identificata la tupla tramite chiave o offset

- **Sequential structures**

- Caratterizzato da una disposizione sequenziale di tuple nella memoria secondaria
- Abbiamo 3 casi
 - * ***entry sequenced***
 - la sequenza delle tuple è dettata dal loro ordine di entrata
 - * ***array***
 - le tuple della stessa dimensione sono disposti in un array
 - la loro posizione dipende dal valore di indice dell'array
 - * ***sequentially ordered***
 - la sequenza dipende dal valore assunto in ogni tupla da un campo che controlla l'ordinamento, noto come ***key field***

- **“Entry-sequenced” sequential structure**

- Ottimale per eseguire letture e scritture sequenziali
- Ottimale per l'occupazione dello spazio, poiché utilizza tutti i blocchi disponibili per i file e tutti gli spazi all'interno dei blocchi
- Non è ottimale invece
 - * ricerca di unità di dati specifiche
 - * update che richiedono più spazio

- **“Array” sequential structure**

- Possibile sono quando le tuple hanno un valore fisso di lunghezza
- Composto da n blocchi adiacenti
- Ogni blocco contiene m slot disponibile per tuple
- Ogni tupla ha un *indice i* e la tupla viene posizionata nella $i - esima$ posizione all'interno dell'array

- “Sequentially-ordered” sequential structure
 - Ogni tupla è posizionata in base al valore del *key field*
 - Il problema principale è dovuto agli inserimenti e aggiornamenti che aumenta lo spazio fisico, *sono richieste tecniche di ri-ordinamento delle tuple già presenti*
- Hash-based access structures
 - Garantisce un accesso efficiente basato sul valore della *key field* per la lettura del record
 - Una struttura *hash based* ha B **blocchi adiacenti**
 - Un **hash algorithm** viene applicato al campo chiave e restituisce un valore compreso tra 0 e $B-1$.
 - * Questo valore è interpretato come posizione del blocco nel file, e viene usato per leggere e scrivere il blocco
- Features of hash-based structures
 - Primitive interface: ***hash(fileId,Key): BlockId***
 - L’implementazione consiste di 2 parti
 - * ***folding***
 - trasforma i valori chiave in modo che diventano valori interi positivi.
 - * ***hashing***
 - trasforma il numero positivo in un valore compreso tra 0 e $B-1$.
 - Ottime performance se il file è più grande del necessario
 - * T numero di tuple previste per il File
 - * F numero medio di tuple memorizzate in ciascuna pagina
 - * B numero blocchi = $\frac{T}{(0.8 \times F)}$ utilizzando l’80% dello spazio disponibile
- Collisions
 - Si presenta una quando lo stesso ***block number*** viene associato a tante tuple
 - * Situazione critica perchè viene superato il numero massimo di tuple per blocco

- Le collisioni vengono risolte aggiungendo una overflow chain
 - * Ossia puntatori ad altri blocchi
 - * Ciò comporta il costo aggiuntivo della scansione della catena
- La lunghezza media della catena dipende da $\frac{T}{(F \times B)}$ e da F

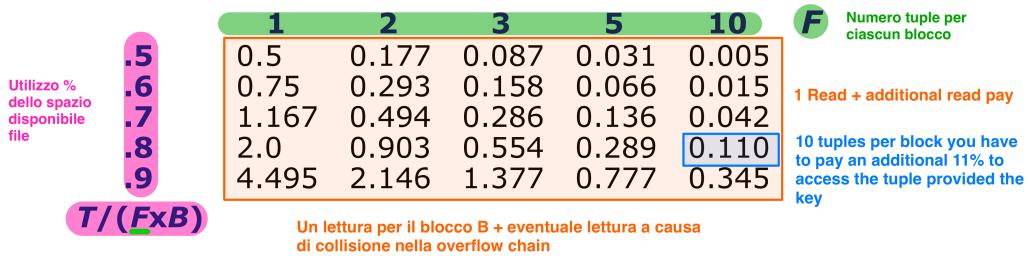


Figure 92: Collisions

• Riassunto

- **Records**
 - * Rappresenta una tupla in una relazione
 - * Un file è una sequenza di record
 - * Sono mappature logiche dei dati su un nastro
- **Blocks**
 - * I blocchi sono unità fisiche di dati su un nastro
 - * Memorizzano i record
- **File**
 - * L'organizzazione dei file si riferisce al modo in cui i record sono archiviati in termini di blocchi e il modo in cui i blocchi sono posizionati sul supporto di memorizzazione e interconnessi.

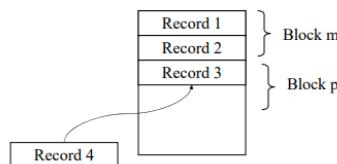


Figure 93: File

- Un record possiede un **field key** e viene memorizzato in un determinato blocco generato dal **hash algorithm**
- B è il numero di blocchi per file e utilizza l'80% dello spazio disponibile
- Nel caso di collisione viene effettuata una lettura per il blocco + una lettura per la overflow chain

40 records

hash table with 50 positions: Per file

- 1 collision of 4 values
- 2 collisions of 3 values
- 5 collisions of 2 values

Si generano collisioni perché per blocco viene memorizzata una sola tupla

Per block = tuple

M mod 50 = 0 - 49 (blocchi)

Key M	Hash 50 M mod 50
60600	0
66301	1
205751	1
205802	2
200902	2
116202	2
200604	4
66005	5
116455	5
200205	5
201159	9
205610	10
201260	10
102360	10
205460	10
205912	12
205762	12
200464	14
205617	17
205667	17

M	M mod 50
200268	18
205619	19
210522	22
205724	24
205977	27
205478	28
200430	30
210533	33
205887	37
200138	38
102338	38
102690	40
115541	41
206092	42
205693	43
205845	45
200296	46
205796	46
200498	48
206049	49

Figure 94: An example

• About hashing

- Le *collisioni (overflow)* possono essere gestite utilizzando il prossimo *blocco disponibile* o con blocchi collegati in un'area chiamata *file di overflow*
- **Inefficiente** per quanto riguarda l'accesso basato su predicati di intervallo o senza chiave
- I file hash *"degenerano"* se lo spazio extra è troppo piccolo (dovrebbe essere come minimo il 120% dello spazio richiesto) e se la dimensione del file cambia molto nel tempo

- **Tree structures**

- Il più utilizzato nei DBMS relazionali
 - * Gli indici SQL sono implementati in questo modo
- Fornisce l'accesso in base al valore di una chiave
 - * nessun vincolo sulla posizione fisica delle tuple
- La *chiave primaria* del modello relazione e la *chiave del hash base e della three structures* sono concetti differenti

- **Index file**

- *Index:*
 - * una struttura ausiliaria per l'accesso efficiente ai record di un file basato sui valori di un dato campo o record di campi, chiamato chiave di *index key*.
- *Il concetto di indice:* indice di un libro, visto come un elenco
 - *Index key non è una chiave*

- **Types of indexes**

- *Primary index:*
 - * Basato sulla chiave primaria
- *Secondary index:*
 - * Basato su altri attributi (inclusa la chiave secondaria)
- *Clustered index:*
 - * I record del file fisico sono ordinati fisicamente in base alla chiave dell'indice
- *Dense index:*
 - * Uno che abbia una voce di indice per ogni record del file
- *Sparse index:*
 - * Avere meno voci di indice rispetto al numero di registrazioni del file

- **Tree structures**

- Ogni albero ha
 - * Un *root node*

- * Alcuni *intermediate nodes*
- * Alcune *leaf nodes*
- Ogni nodo corrisponde a un **blocco**
- I collegamenti tra i nodi sono stabiliti da puntatori alla memoria di massa
- In generale, ogni nodo ha un numero elevato di discendenti (*fan out*)
- In un **balanced tree**, le lunghezze dei cammini dal nodo radice ai nodi foglia sono tutti uguali.
- * Gli alberi bilanciati offrono prestazioni ottimali.

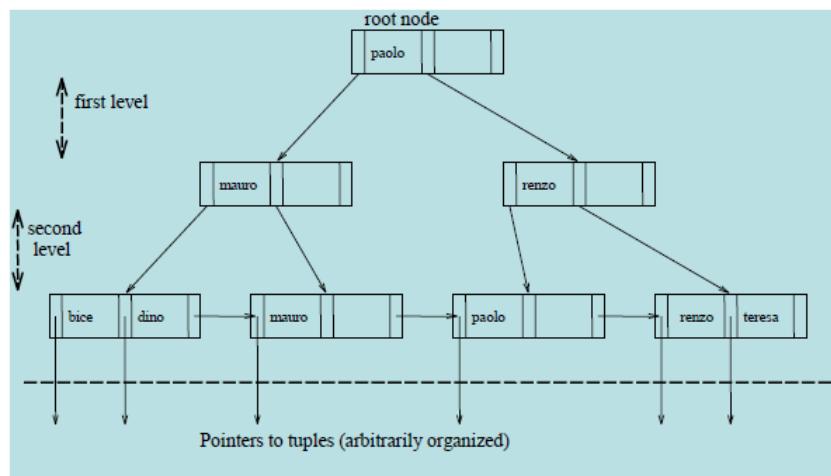


Figure 95: An example

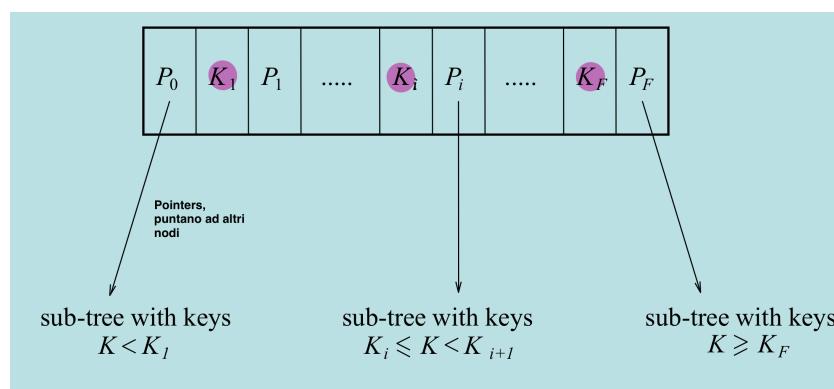


Figure 96: Structure of the tree nodes

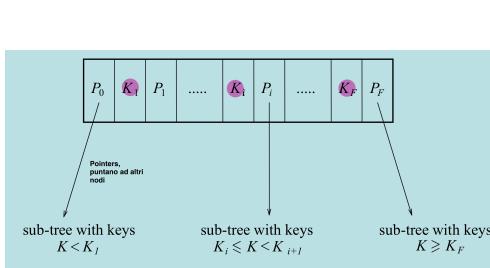
- **B and B+ trees**

- ***B+ trees***

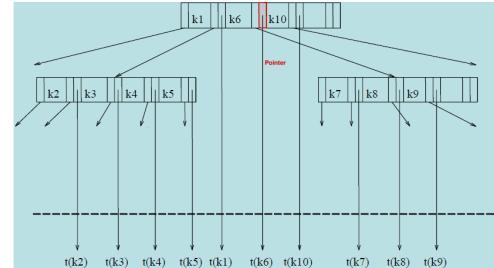
- * I nodi foglia sono collegati in una catena nell'ordine imposto dalla chiave.
 - * Supporta le query a intervalli in modo efficiente
 - * Il più utilizzato dai DBMS relazionali

- ***B trees***

- * Nessuna collegamento sequenziale tra i nodi foglia
 - * Supporta le query a intervalli in modo efficiente
 - * Il più utilizzato dai DBMS relazionali
 - * I nodi intermedi usano due puntatori per ogni chiave K_i
 - Un puntatore punta direttamente al blocco che contiene la tupla correttamente a K_i
 - Mentre l'altro puntatore punto a un **sottoalbero** con chiavi $K_i < x < K_{i+1}$



(a) B+ tree



(b) B tree

- **Search technique**

- Alla ricerca di una tupla con **valore chiave** V , per ogni nodo intermedio:
 - * Se $V < K_1$ segui P_0
 - * Se $V \geq K_F$ segui P_F
 - * Altrimenti, segui P_j dove $K_j \leq V < K_{j+1}$
 - I nodi foglia possono essere organizzati in 2 modi
 - * **In key-sequenced:** le tuple sono contenute nelle foglie
 - * **In indirect trees:** i nodi foglia contengono puntatori alle tuple, allocato con qualsiasi altro meccanismo (entrysequenced, hash, key-sequenced, ...)

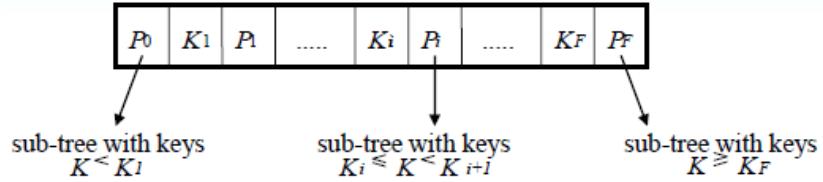


Figure 97: Search technique

- Split and Merge operations

 - **SPLIT:**

 - * richiesto quando l'inserimento di una nuova tupla non può essere eseguita localmente su un nodo
 - * Provoca un aumento dei puntatori nel nodo superiore
 - * potrebbe causare ricorsivamente un'altro **split**

 - **MERGE:**

 - * richiesto richiesto quando due nodi "vicini" hanno voci che potrebbe essere condensato in un unico nodo.
 - * Fatto per mantenere un riempimento elevato del nodo
 - * Generare percorsi minimi dalla radice alle foglie.
 - * Provoca una diminuzione dei puntatori nel nodo superiore
 - * potrebbe causare ricorsivamente un'altro **merge**

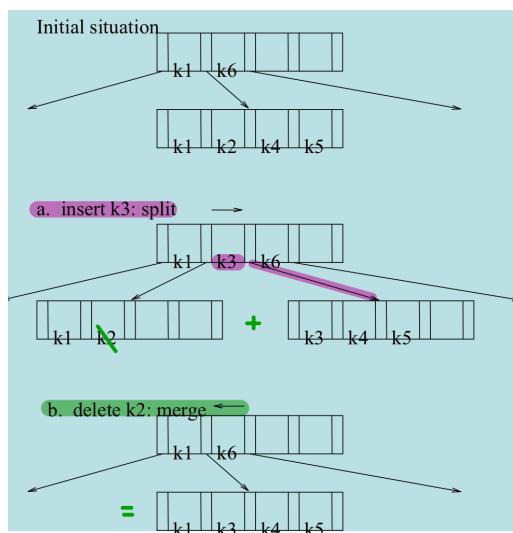


Figure 98: Split and merge

- **Index usage**

- Ogni tabella dovrebbe avere
 - * **A primary index:** unico, sulla chiave primaria
 - * **Several secondary indexes:** sia unico che non, solitamente su attributi maggiormente utilizzati per selezioni e join
- Si aggiungono progressivamente verificando che il sistema li usa davvero, e senza eccessi

- **Query optimization**

- **Optimizer:** un modulo importante nell'architettura di un DBMS
- Riceve una query scritta in SQL e produce un programma di accesso in formato "oggetto" o "interno", che utilizza i metodi di accesso ai dati.
- I vari passi
 - * Analisi lessicale, sintattica e semantica
 - * Traduzione in una rappresentazione interna
 - * Ottimizzazione algebrica
 - * Ottimizzazione basata sui costi
 - * Generazione del codice

- **Internal representation of queries**

- Una rappresentazione ad albero, simile all'algebra relazionale:
 - * I nodi foglia corrispondono *strutture di dati fisici* (tabelle, indici, file).
 - * I nodi intermedi rappresentano l'accesso alle *strutture di dati fisici* che sono supportate dai metodi di accesso
- Tipicamente le operazioni effettuate includono
 - * scansioni sequenziali
 - * ordinamenti
 - * accessi indicizzati
 - * metodi di valutazione di join
 - * ..

- Input: query in SQL
 $\text{SELECT } A \text{ FROM } R, S, T \text{ WHERE } R.A=S.A \text{ AND } R.B=T.B$
- Output: execution plan

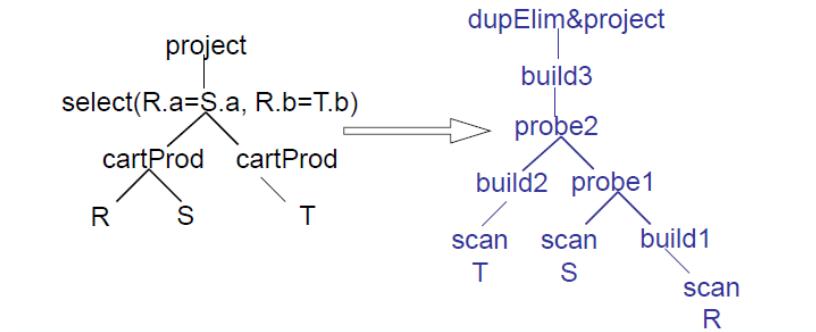


Figure 99: Query optimization input-output

- Approaches to query execution
 - *Compile and store:*
 - * la query viene compilata una volta ed eseguita molte volte
 - * Il codice interno è memorizzato nel DBMS, insieme alle indicazioni delle dipendenze del codice
 - * e la particolare versione utilizzata nel momento della compilazione
 - * Su modifiche rilevanti del **catalogo**, la compilazione della query viene invalidata e ripetuta
 - *Compile and go:*
 - * esecuzione immediata, nessuna memorizzazione
 - * Anche se non memorizzato, il codice potrebbe rimanere per un po' di tempo nel DBMS ed essere disponibile per altre esecuzioni
- Relation profiles
 - I profili contengono informazioni quantitative su tabelle e sono memorizzati nel *data dictionary*
 - * La cardinalità (numero di tuple) per ogni tabella T
 - * La dimensione in byte per ogni attributo A_j in T
 - * Il numero di valori distinti di ciascun attributo A_j in T

- * il valore minimo e massimo di ogni attributo A_j in T
- Calcolato periodicamente attivando appropriate *system primitives*
- Utilizzato nell'ottimizzazione basata sui costi per stimare le dimensioni dei risultati intermedi delle query in esecuzione

- **Sequential scan**

- Esegue un accesso sequenziale a tutte le tuple di una tabella o di un risultato intermedio, eseguendo contemporaneamente varie operazioni, come
 - * Proiezioni su un set di attributi
 - * Selezioni su predicati
 - * Sort (ordinamenti)
 - * Inserimenti, cancellazioni, modifiche alle tuple attualmente accessibili durante la scansione

- **Sort**

- Questa operazione viene utilizzata per ordinare i dati secondo il valore di uno o più attributi
 - * Ordinamenti in memoria centrale utilizzando algoritmi ad-hoc
 - * Ordinamenti su grandi file, questi non possono essere trasferiti in memoria centrale, viene eseguita unendo piccole parti a parti già ordinate

- **Indexed access**

- Gli indici vengono usati quando le query includono
 - * simple predicates ($A_i = v$)
 - * interval predicates ($v_1 < A_j < v_2$)
- Con predicati di congiunzione supportati, (*AND*) il DBMS sceglie il predicato supportato più selettivo per il primary access e valuta gli altri predicati nella memoria principale
- Con predicati di disgiunzione (*OR*)
 - * Se nessuno di loro è supportato, è necessaria una scansione
 - * se tutti sono supportati, gli indici possono essere utilizzati solo con l'eliminazione dei duplicati

- **Join Methods**

- I join sono le operazioni più frequenti (e costose) nei DBMS
- Esistono diversi metodi per la valutazione del join
 - * nested-loop
 - * merge-scan
 - * hashed
- Questi tre metodi si basano su scansione, hashing, e ordinamenti.

- **Cost-based optimization**

- Un problema di ottimizzazione, le cui decisioni sono
 - * Le operazioni di accesso ai dati da eseguire
 - * L'ordine delle operazioni
 - * Parallelismo e pipelining possono migliorare le prestazioni
- Ulteriori opzioni vengono visualizzate nella selezione di un piano all'interno di a contesto distribuito

- **Approach to query optimization**

- Approccio di ottimizzazione:
 - * Utilizzo di profili e di formule approssimative sui costi
 - * Costruisci un albero decisionale, in cui ogni nodo corrisponde a una scelta
 - * ogni nodo foglia corrisponde ad uno specifico piano di esecuzione.
 - * Assegna ad ogni piano un costo
 - * Scegli il piano con il costo più basso
- L'optimezer dovrebbe ottenere buone soluzioni in un tempo breve

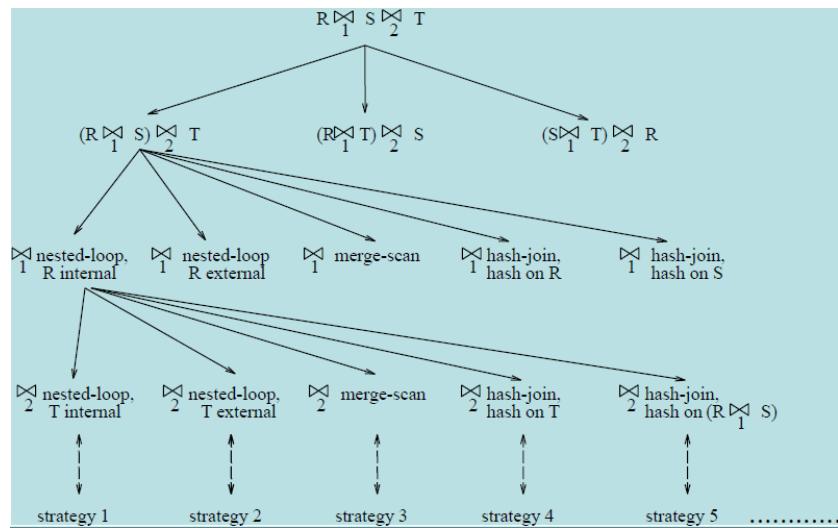
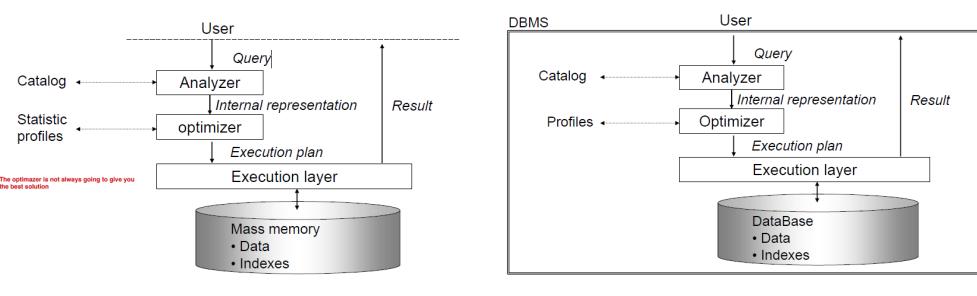


Figure 100: An example of decision tree



(a) Query processing components

(b) Centralized architecture (DBMS)

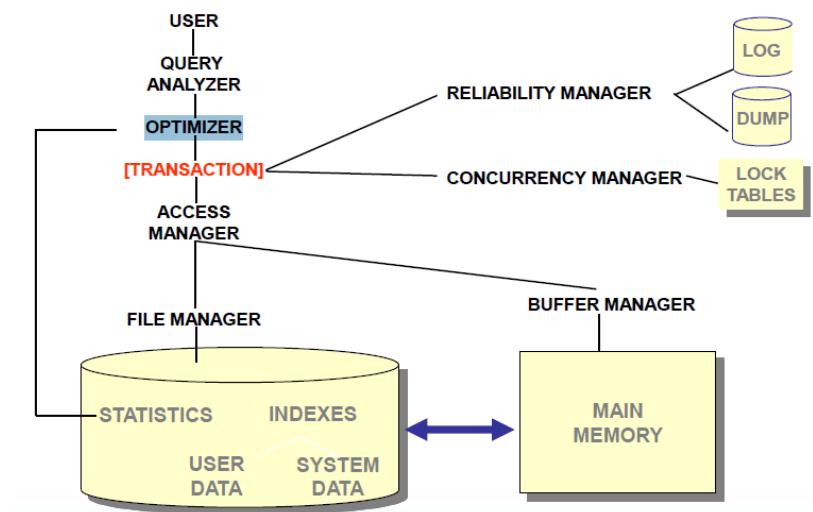


Figure 101: Overall view: components of a DBMS