

# **Advanced Databases**

**AA 2018/19**

**Luca Corbucci**

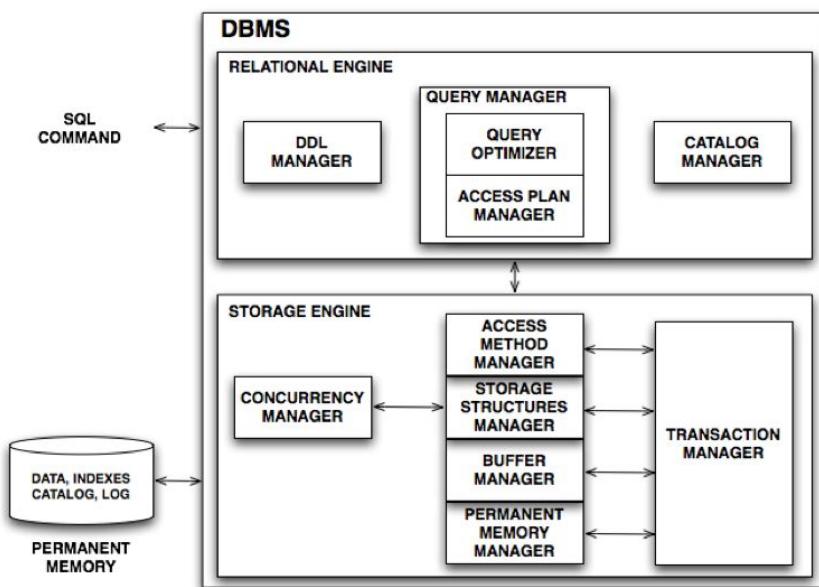
# Sommario

<b>Lezione 1</b>	<b>2</b>
<b>Lezione 2</b>	<b>9</b>
<b>Lezione 3</b>	<b>15</b>
<b>Lezione 4</b>	<b>22</b>
<b>Lezione 5</b>	<b>29</b>
<b>Lezione 6</b>	<b>46</b>
<b>Lezione 7</b>	<b>51</b>
<b>Lezione 8</b>	<b>84</b>
<b>Lezione 9</b>	<b>91</b>
<b>Lezione 10</b>	<b>109</b>
<b>Lezione 11</b>	<b>134</b>
<b>Lezione 12: Decision Support Systems</b>	<b>150</b>
<b>Lezione 13: Column Databases</b>	<b>166</b>
<b>Lezione 14: Parallel and Distributed Systems</b>	<b>185</b>
<b>Lezione 15: BigData &amp; NoSQL</b>	<b>210</b>
	<b>221</b>

# Lezione 1: Permanent Memory and Buffer Management

## Architettura dei DBMS

Un database è una collezione di dati omogenei che contengono delle relazioni e sono utilizzati per mezzo di un DBMS (Data Base Management System). Vediamo ora la struttura di un DBMS basato sul modello relazionale ovvero su un modello logico di rappresentazione dei dati strutturato attorno al concetto di relazione (tabella).



L'architettura dei DBMS può essere suddivisa in due parti principali:

- Storage Engine, a sua volta formato da:
  - Permanent Memory Manager: gestisce l'allocazione e la deallocazione delle pagine sul disco. Ci fornisce una visione della memoria come un insieme di database ognuno formato da un set di files e quindi da un set di **pagine** di dimensioni differenti. I dati vengono trasferiti dal disco alla memoria temporanea sotto forma di pagine, in particolare possiamo

dire che le performance di ricerche e delle altre operazioni sul database dipendono da quante pagine vengono spostate dal disco alla memoria temporanea.

- Buffer Manager: gestisce il trasferimento delle pagine dal disco alla memoria principale. Il buffer è una risorsa limitata che quindi va gestita nel modo più intelligente possibile.
- Storage Structures Manager: gestisce le strutture dati per ottenere e memorizzare velocemente informazioni
- Access Methods Manager: fornisce le operazioni per interagire con il database
- Transaction Manager: garantisce la consistenza del database
- Concurrency Manager: garantisce l'accesso concorrente al database senza andare a compromettere la consistenza
- The Relational Engine che comprende:
  - Data Definition Language: processa i comandi inviati dall'utente
  - Query Manager: ottimizza le query
  - Catalog Manager: gestisce i metadati

Una implementazione di DBMS relazionali è JRS, scritto in Java che ci permette di scrivere ed eseguire query SQL andando poi a capire come le query vengono ottimizzate.

## Memoria Permanente e Buffer Management

La memoria utilizzata da un DBMS è organizzata in una gerarchia con due livelli, abbiamo una memoria principale e una memoria permanente:

- La memoria principale ha una dimensione non troppo grande, è volatile e la accedo in pochissimo tempo (nanosecondi)
- La memoria permanente (quindi non volatile) invece può essere o con dischi magnetici che hanno una grande capienza ma sono molto lenti nell'accesso ai dati oppure con Flash Memory che

hanno una dimensione minore ma riducono i tempi di accesso ai dati

Per le Flash Memory c'è però da considerare il fatto che abbiamo una velocità maggiore se vogliamo scrivere o leggere i dati mentre se vogliamo sovrascrivere i dati sarà più problematico. Va inoltre considerato che le SSD garantiscono una quantità di cancellazioni/scritture su disco pari a circa 100k, quindi potrebbero non essere troppo adatte in questo caso.

Utilizzando un disco meccanico abbiamo dei tempi di accesso ai dati che sono più alti rispetto alla memoria temporanea, questo perchè la struttura del disco è formata da:

- Vari piatti di dimensione 2.5 o 3.5
- Su ogni piatto abbiamo una serie di track che accediamo senza dover muovere il braccio
- Ogni track è diviso in settori in cui ci sono i dati
- Un insieme di track forma un blocco di dati

Il costo che abbiamo in termini di tempo per leggere un dato presente nel disco dipende da:

- Tempo per raggiungere quel determinato track (ts)
- Tempo per raggiungere il blocco da cui vogliamo leggere i dati (tr)
- Tempo per trasferire i dati in memoria principale (tb)

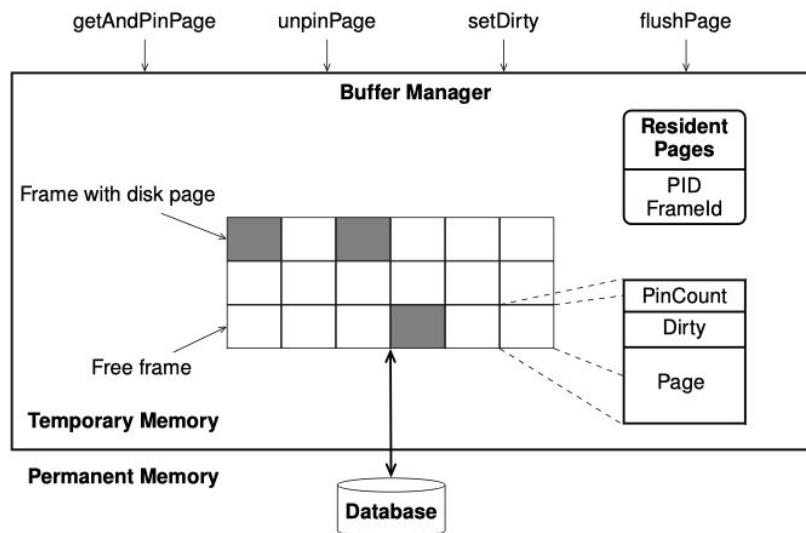
Quindi complessivamente abbiamo  $(ts+tr+tb)$ , se leggiamo k file allora diventa  $(ts+tr+tb*k)$ .

Quando devo aggiungere o eliminare dei dati presenti all'interno del database abbiamo a disposizione il **Permanent Memory Manager** che ci permette di gestire l'allocazione e la deallocazione delle pagine dal disco. Il PMM ci offre anche un'astrazione della memoria permanente in termini di database formati da file che a loro volta sono formati da pagine di dimensione fissata. Queste pagine sono spostate dalla memoria permanente a quella temporanea e sono numerate partendo da 0. (Le pagine nel disco sono chiamate pagine fisiche,

quando le spostiamo nella memoria temporanea le chiamiamo semplicemente pagine).

Data una transazione, il **buffer Manager** è il componente del DBMS che **si occupa di prendere dalla memoria permanente le pagine che ci servono cercando di minimizzare gli accessi al disco** utilizzando un buffer con una strategia di replacement delle pagine.

Aumentando la dimensione del buffer è probabile che si riesca anche a ridurre il costo delle operazioni da effettuare perché abbiamo nella memoria temporanea un numero maggiore di pagine che potremmo riutilizzare senza doverle andare a prendere in memoria.



Il componente principale del Buffer Manager è un buffer pool che è un array di frame di dimensione fissata, ogni frame contiene una pagina di memoria ed è identificabile mediante un FrameID.

Per ogni frame abbiamo anche due variabili aggiuntive:

- PinCount: conta quante sono le richieste di quella specifica pagina da parte delle varie transazioni.
- Dirty: variabile booleana che mi indica se quella pagina è stata modificata o meno.

Abbiamo a disposizione anche una tabella chiamata “Resident Pages” che contiene un identificativo della pagina che stiamo cercando e il frame corrispondente in cui è stata memorizzata (se il buffer pool contiene quella pagina).

Per gestire il buffer pool abbiamo a disposizione delle primitive:

- GetAndPinPages: è la funzione che ci permette di controllare se la pagina che stiamo cercando è già nell'array dei frame, nel caso in cui sia già presente allora dobbiamo andare ad aumentare il valore del pin count e poi viene restituito l'identificatore della pagina.

Se invece la pagina non è nel buffer:

- Viene selezionato un frame libero, se non c'è uno libero dobbiamo utilizzare una politica di replacement per capire dove posizionare la nuova pagina. Solitamente la politica che viene utilizzata per rimpiazzare le pagine è LRU, abbiamo una coda e viene cancellata di volta in volta la pagina che abbiamo usato meno recentemente sostituendola con una nuova.

Se il frame che abbiamo selezionato per la rimozione ha il dirty bit fissato a true allora prima di cancellare il frame devo salvare la modifica anche nel disco.

- Una volta scelta la pagina vado nel disco e leggo la pagina e la sposto nel pool con valore del pin count fissato a 1.
- Aggiorno anche la Resident pages table in modo da linkare la posizione del frame nel pool alla nuova pagina che ho appena inserito.
- SetDirty() mi permette di modificare il valore della variabile booleana Dirty
- UnpinPage(P) diminuisce il valore del Pin Count
- FlushPage(P) scrive nella memoria permanente le modifiche che sono state effettuate su quella pagina di memoria.

Una pagina può essere rimossa dal pool solamente se il pinCount = 0 e se il dirty bit è falso. Altrimenti devo aspettare di portare a 0 il pinCount,

devo scrivere le modifiche sul disco e poi a quel punto sarà possibile eliminare la pagina dal pool.

## Lezione 2: Data Organizations

### Struttura dei record

Lo storage structure manager implementa la memorizzazione dei record all'interno della memoria permanente. I record vengono memorizzati all'interno di file di pagine, possiamo dire che sopra al livello dello storage structure manager accediamo ai record mentre sotto a questo livello accediamo alle pagine.

I record vengono associati ad un RID ovvero ad un id univoco che poi viene utilizzato per recuperare il record dalla memoria.

Le pagine hanno dimensione fissata e quando vogliamo accedere ad un record il vero costo sta nel portare la pagina nella memoria principale e non nel modificare il record una volta che si trova nella memoria principale.

Ogni record è formato da vari attributi e da un record header che viene sfruttato per gestire meglio i record (può contenere ad esempio informazioni sulla dimensione del record o sul numero di attributi). Possiamo assumere che un record non è più grande di una pagina.

### Struttura delle pagine

Ad ogni record che viene memorizzato in un database viene assegnato un RID ovvero un record identifier, questo record identifier ci consente anche di andare a recuperare il record all'interno della pagina in cui si trova.

Una possibile soluzione consiste nell'utilizzare un RID che mantiene due informazioni (pagina, posizione nella pagina).

Questo però comporta che, in caso di modifica di un record, andrebbero modificati i RID dei record che si trovano in quella pagina e quindi verrebbero anche modificati tutti i riferimenti di altri record a quei RID. Quindi la soluzione che si adotta consiste nell'utilizzo all'interno delle pagine di uno slot array capace di mantenere in ogni posizione l'indirizzo del record all'interno della pagina. Quindi il RID mi dirà il numero della

pagina in cui devo cercare e poi la posizione dello Slot Array in cui devo andare a cercare l'indirizzo del record.

In questo modo se modifICO il contenuto di un record, mi basta modificare l'indirizzo nello slot array e di nuovo posso accedere al record senza fare ulteriori cambiamenti. Se invece ad esempio ho un record che non entra tutto in una sola pagina, viene utilizzato un puntatore dalla prima pagina alla seconda (in pratica un secondo RID) e in questo modo all'esterno della pagina non ci sono modifiche da fare.

## Gestione delle pagine

Quando aggiungo un nuovo record vengono effettuati i seguenti passaggi:

- Cerco e trovo una pagina che contiene abbastanza spazio libero per contenere il nuovo record. Prendo l'indirizzo P della pagina
- Memorizzo il record nella pagina e mi prendo la posizione j dello slot array che contiene l'indirizzo del record nella pagina
- Assegno al record il RID formato da (P,j)

L'inserzione di un record deve essere implementata in modo efficiente, possiamo utilizzare due modelli differenti:

- Consideriamo la prima pagina del file che è chiamata Header Page, da questa partono due linked list, in una linked list inseriamo tutte le pagine che sono completamente piene e in un'altra quelle che invece hanno dello spazio libero al loro interno.  
Quando una pagina viene svuotata o quando viene riempita allora sposto la pagina da una lista all'altra
- In alternativa possiamo memorizzare una directory di pagine e per ognuna ci salviamo il puntatore alla pagina e la quantità di memoria disponibile. Se la directory si esaurisce ne creiamo una nuova con una linked list. Se dobbiamo fare un inserimento prima cerchiamo una pagina che abbia abbastanza posto

Per motivi di efficienza non andiamo a unire le pagine che hanno dello spazio vuoto, quindi potremmo trovarci nella situazione in cui abbiamo varie pagine vuote che però non hanno abbastanza spazio per tutti i dati che dobbiamo inserire. In questo caso va riorganizzato il database.

## Costi

Le operazioni che effettuiamo sui record che sono in memoria temporanea non hanno un impatto enorme sui costi, quello che conta sono gli accessi in memoria.

Per il calcolo dei costi delle varie operazioni si considera il numero di record che abbiamo ( $N_{rec}$ ), poi la lunghezza dei record ( $L_r$ ) poi il numero delle pagine ( $N_{pag}$ ) e la dimensione delle pagine ( $D_{pag}$ ).

Le operazioni classiche sono la ricerca (è importante perchè la facciamo sempre) che può essere una equality search o una range search, poi abbiamo inserimento, aggiornamento ed eliminazione di un record del database.

## Heap e Sequential Organization

---

### Altro libro:

I record vengono mantenuti in un file che supporta anche una operazione di scansione che ci consente di scorrere tutto il file.

Quando un file è organizzato in modo che l'ordinamento dei dati è simile all'ordinamento dei dati in base a qualche indice diciamo che l'indice è clusterizzato, altrimenti è non clusterizzato.

Un indice è la struttura dati che organizza i record nel disco in modo da velocizzare e ottimizzare l'esecuzione di alcuni tipi di operazioni. Un

indice ad esempio ci permette di ottenere i record che soddisfano una certa condizione su una chiave che utilizziamo per fare la ricerca.

Abbiamo due tipi di indici:

- Indice primario: si tratta dell'indice che include la primary key. In questo caso i record memorizzati nell'indice hanno associata una chiave di ricerca  $k^*$  che ci permette di localizzare il record che identifichiamo con la chiave  $k$ . Con l'indice primario una entry dell'indice la identifichiamo solamente con  $k^*$
  - Indice secondario: è l'indice che non include la primary key. Una entry dell'indice in questo caso è rappresentata come una coppia  $\langle k, \text{rid} \rangle$  oppure con  $\langle k, \text{rid-list} \rangle$ .
- 

Ci sono vari metodi per gestire l'inserimento dei dati all'interno delle pagine, i più semplici sono:

- **Heap Organization:** quando dobbiamo inserire un nuovo record all'interno del file, prendo l'ultima pagina e lo inserisco qua. Quindi seguiamo l'ordine di inserimento e questo è un bene perchè il costo dell'inserimento di un record è 2 visto che spostiamo una volta la pagina dalla memoria permanente alla temporanea e poi una volta la scriviamo sul disco con il nuovo dato.  
Questo metodo viene utilizzato quando abbiamo pochi dati e quando vengono effettuate poche ricerche.  
Lo spazio occupato è  $N_{\text{pag}}$  ovvero dipende dal numero di pagine che uso, la ricerca sequenziale mi costa al caso pessimo  $N_{\text{pag}}$  perchè non trovo quel record nelle pagine, se invece il record c'è in media ho un costo di  $N_{\text{pag}}/2$ .  
Se voglio fare una ricerca all'interno di un certo range invece dovrò scorrere tutte le pagine perchè non sappiamo dove si possono trovare gli elementi che entrano nel range.

La cancellazione infine costa una ricerca + 1. Il + 1 è dovuto al fatto che scrivo su disco.

- **Sequential Organization:** In questo caso quando inseriamo un nuovo record dobbiamo mantenere un ordinamento su un certo attributo. Quindi questo può comportare un inserimento a metà di una delle pagine e non sempre inseriremo alla fine, questo comporta alti costi per mantenere ordinati i record nelle varie pagine.

La memoria necessaria è la stessa che abbiamo nell'Heap Organization. La ricerca è migliore in questo caso rispetto allo Heap perchè ora abbiamo la possibilità di fare una ricerca binaria nel caso in cui i dati siano salvati in modo sequenziale e questo ci costerebbe  $\log(N_{\text{Pag}})$ , se non sono salvati vicini abbiamo un costo pari al caso con lo Heap.

Per considerare il range search dobbiamo considerare il range massimo di valori che abbiamo quindi  $k_{\max}$  e  $k_{\min}$ . Calcoliamo quindi un selectivity factor **sf** che è pari a  $(k_2 - k_1) / (k_{\max} - k_{\min})$  e mi indica una stima del numero di pagine che saranno occupate dagli elementi nel range.

Quindi alla fine il range search ha un costo pari a **C<sub>s</sub>** =  $\log(N_{\text{pag}}) + (\text{sf} * N_{\text{Pag}}) - 1$ .

L'inserimento invece è decisamente peggiore, se inseriamo in una pagina che ha spazio il costo è  $C_s + 1$ , se invece inserisco nel mezzo di una pagina allora devo modificare tutte le pagine successive, quindi il costo in termini di pagine lette e scritte diventa  $C_s + N_{\text{pag}} + 1$ .

## External Sorting

L'operazione di ordinamento nei database è essenziale ed è bene farla in modo che sia efficiente. Consideriamo l'ordinamento dei vari record all'interno delle pagina salvate su disco.

Si tratta di un "External Sorting" che possiamo dividere in una prima fase in cui prendiamo blocchi di dati grandi quanto il buffer e creiamo dei run ordinati che vengono scritti sul disco, poi eseguiamo il merge e il numero

di run che vengono ordinati ogni volta dipende tutto dalla dimensione del buffer.

Questo algoritmo è simile al classico Merge Sort (che funziona in  $n \log n$ ) con la differenza che la base del logaritmo non è 2 ma è un valore molto più alto quindi vado a mergiare più run contemporaneamente e questo comporta che alla fine il costo è praticamente una costante.

Abbiamo in tutto un costo che è  $2 * N_{\text{pag}} + 2 * N_{\text{pag}} * \text{Numero Merge Phases}$ . Il primo termine è dovuto al fatto che ordiniamo i run, il secondo al fatto che dobbiamo leggere i run ordinati e poi eseguire il merge.

Alla fine il costo è circa  $2(N_{\text{Pag}} * \log_2(N_{\text{Pag}}))$ . (Importante: vuole che si dica così la formula perchè si ragiona in termini di pagine e non va bene  $n \log n$ ).

## Lezione 3: Hashing Organization

Si utilizza l'hashing per fare in modo che un record venga recuperato velocemente e con pochi accessi. In questo caso infatti ad ogni record viene assegnata una certa chiave che poi verrà usata per la ricerca. Il mapping <chiave, records> lo possiamo implementare in due modi:

- Primary Organization: l'organizzazione delle tabelle è "primary" se determina il modo in cui i record sono fisicamente memorizzati. Il mapping dalla chiave al record viene implementato tramite una funzione hash abbiamo  $k \rightarrow \text{record}$ .  
Una primary organization è statica se partiamo con una dimensione fissata della tabella e poi aggiungendo nuovi dati abbiamo un degrado delle prestazioni, è dinamica se invece la dimensione della tabella evolve gradualmente con gli inserimenti. Il termine Primary organization viene anche utilizzato per indicare l'organizzazione per la chiave primaria.
- Secondary Organization: in questo caso utilizziamo una tabella che viene utilizzata come un indice e mi elenca input e output. Questa secondary organization aiuta nel momento in cui dobbiamo rispondere ad una query ma non va a modificare il modo in cui i dati vengono memorizzati.  
L'indice è una tabella  $I(K, RID)$  ordinata sulla base del valore di  $K$ , ogni elemento dell'indice è una coppia, abbiamo  $K$  che è la chiave e il RID del record corrispondente.  
Il termine Secondary organization viene anche utilizzato per indicare l'organizzazione delle chiavi secondarie.

### Static Hashing Organization

Abbiamo bisogno di:

- Una funzione hash
- Una tecnica per gestire gli overflow
- Il load factor

- La dimensione della pagina

Ammettiamo di avere dei record che hanno tutti la stessa dimensione, abbiamo una quantità  $M$  di pagine e ogni pagina ha dimensione  $C$ . Dati questi dati possiamo calcolare il loading factor che è dato da  $d=N/(M*c)$ . Il load factor mi indica quanti record abbiamo nelle varie pagine e quindi mi indica anche la probabilità di avere un overflow. In genere è preferibile avere una quantità di pagine minore ma con una maggiore dimensione e un load factor minore che avere invece tante pagine ma di dimensioni ridotte.

L'overflow avviene quando abbiamo una collisione  $H(K1)=H(K2)$  e ci troviamo a dover inserire all'interno delle pagine nuovi record senza però avere lo spazio.

In questo caso abbiamo due possibilità:

- Open Overflow: Possiamo prendere la prima posizione libera che troviamo facendo una scansione lineare e poi inseriamo lì il record
- Chained Overflow: Usiamo una overflow area in cui inserire il record.

La pagina in cui va inserito il record dipende dal risultato della funzione hash che calcoliamo, una funzione hash classica che viene usata è  $H(x) = k \text{ mod } M$ .

Le performance di una organizzazione statica sono ottime fino a quando non bisogna iniziare a gestire gli overflow, in quel momento le prestazioni degradano e ad un certo punto sarà anche necessario ricostruire l'intero file.

La fase di riorganizzazione:

Per ogni pagina viene chiamata la routine di inserimento e ogni record viene posizionato in una nuova pagina.

Questa riorganizzazione mi porta a dover leggere per una volta tutte le pagine, poi per ogni record di una pagina, questo viene scritto in un'altra pagina, quindi abbiamo un costo di  $N\text{Page} + 2(N.\text{record})$ .

In totale abbiamo  $2N\text{Page} + 2(N.\text{record})$ .

Possiamo migliorare le performance andando ad ordinare i record in

base all'hash che abbiamo calcolato, quindi dati gli hash ordinati abbiamo un costo che viene ridotto a  $4^*N\text{Page}$ .

Utilizzando l'hash perdiamo anche la possibilità di svolgere il range search velocemente perchè bisogna leggere l'intero file, dal punto di vista del numero di pagine che accediamo siamo messi anche peggio del metodo sequenziale perchè in questo caso le pagine non sono piene.

Le performance dell'equality search sono ottime e abbiamo solamente una pagina da prendere dal disco quando cerchiamo un certo record.

## Dynamic Hashing Organization

Possiamo considerare due metodi per il dynamic hashing, un primo metodo che usa uno spazio primario per inserire i record basandoci sull'hash e un secondo che usa uno spazio primario e poi una struttura dati ausiliaria.

Abbiamo la necessità di utilizzare una funzione hash che possa essere modificata e che quindi possa adattarsi alla dimensione del primary space.

## Virtual Hashing

Nel caso del virtual hashing abbiamo uno spazio primario che contiene  $M$  pagine, ognuna di queste  $M$  pagine ha una dimensione  $c$ .

Abbiamo poi un bit vector in cui segniamo con 1 le pagine in cui è presente almeno un record e con 0 le pagine in cui non ci sono record. Inizialmente partiamo con la dimensione  $M$  e quindi abbiamo una funzione hash che mappa i record nelle  $M$  pagine, poi quando si verifica un overflow andiamo a raddoppiare la dimensione  $M$  del primary space. Quando raddoppiamo questo spazio è necessario modificare la funzione hash e calcolare l'hash per il record che ha causato overflow e per quelli che stavano nella pagina in cui abbiamo avuto overflow.

La regola mi dice che quando ricalcolo l'hash posso mandare quel record nella stessa posizione in cui si trovava prima oppure in una nuova posizione che sarà distante  $(2^j)^*M$  dalla posizione in cui stava prima,

dove j è il numero di volte che ho raddoppiato la dimensione del primary space.

a)

							$H_0(3820)$
0	1	2	3	4	5	6	
112	519			6647	2385		
1176	3277		723	1075	2665		
	848			7830	2840		
					286		
1	1	1	1	1	1	1	1

b)

														$H_1$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	
112	519			6647	2385									
1176	3277		723	1075	2665									
	848			7830	2840									
					286									
1	1	1	1	1	1	1	1	0	0	0	0	0	1	0

Per quanto riguarda i costi:

- La memoria non viene gestita bene e il load factor è in media 0.67.  
La cosa positiva è che quando vogliamo recuperare un certo record il costo per accedere a quel record è di una sola pagina.
- Assumiamo che il bit vector entra tutto nel buffer, in questo caso un record viene recuperato con un costo di 1 solo accesso in memoria. Questo perchè prima facciamo il calcolo dell'hash che mi costa 0, poi guardiamo il bit vector (se è in memoria costa 0) poi leggiamo la pagina e quindi costa 1.  
Se il bit vector non entra in memoria temporanea non ha senso usare questo metodo perchè avrei 1 accesso al disco solamente per leggere il bit vector.
- Il costo del lookup è di 1 pagina con virtual hash.
- Può essere un problema il continuo raddoppio della memoria necessaria. Questo perchè andando avanti a duplicare avremo un virtual space che sarà molto sparso. Questo può essere un problema se quello che troviamo nel virtual space non è l'effettivo posizionamento sul disco dei vari record, in questo caso infatti abbiamo anche la necessità di una relocation table e quindi potrebbe essere un problema la memorizzazione di questa tabella.

- Se la struttura del virtual hash non è mappata direttamente nel disco può essere un problema e dovrei creare una remap table che quindi mi occuperebbe anche altro spazio.

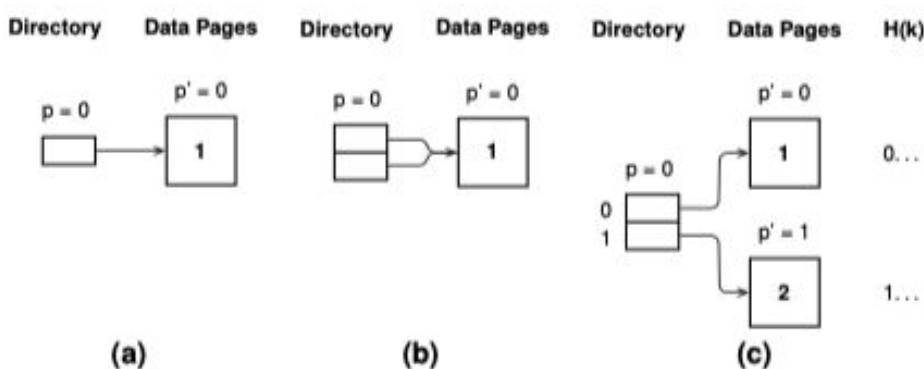
## Extensible Hashing

In questo caso oltre alle pagine che contengono i record, abbiamo anche una struttura dati ausiliaria che è chiamata directory (B) e che contiene i puntatori alle pagine. Le pagine vengono allocate quando è necessario e in questo modo si diminuisce lo sparse problem del virtual hash.

Dato un record  $r$ , calcoliamo l'hash di  $b$  bit e consideriamo i primi  $p$  bit che vengono usati come offset all'interno della directory. In B abbiamo sempre un numero di puntatori che è una potenza di 2, in particolare è  $2^p$ .

Poi abbiamo un secondo livello di organizzazione dei record, ogni puntatore della directory punta infatti ad una pagina dove troviamo quei record che hanno un hash con almeno  $p'$  bit in comune, con  $p' < p$ .

Quando non abbiamo abbastanza spazio per inserire all'interno di una pagina tutti i record, dobbiamo andare a raddoppiare la dimensione di B, le nuove posizioni che inseriamo all'interno di B comportano anche un aumento del valore di  $p$ , inizialmente avremo che i nuovi puntatori punteranno alla pagina già esistente, poi, dato che ora  $p' < p$  verrà creata una nuova pagina e i record che stavano nella prima verranno suddivisi tra le due pagine.



Risolviamo il problema di una hash table sparsa ma allo stesso tempo ora abbiamo un problema perchè si crea una tabella hash che diventa sempre più grande.

## Linear Hash

Il linear Hash cerca di evitare l'uso delle strutture dati in modo da evitare che si abbiano problemi di memoria come avviene nel caso del virtual hash con la bitmap o nell'extensive hash con l'hash table.

In questo caso partiamo con M pagine e un puntatore P che punta alla prima di queste pagine, quando si verifica un overflow, indipendentemente dalla pagina in cui si verifica, viene aggiunta una nuova pagina in cui vengono messi una parte dei record che stavano nella pagina puntata da P. Il record che crea l'overflow invece viene memorizzato tramite una overflow chain. Quando si verifica un overflow modifichiamo anche la funzione hash raddoppiando il valore della M che usiamo nel modulo, ad esempio  $H_0 = k \bmod M$ ,  $H_1 = k \bmod 2M$ .

Questo raddoppio fa sì che quando abbiamo un record che finisce in una certa pagina P con la funzione  $H_0$ , poi usando la funzione  $H_1$  avremo un record che finirà nella pagina  $P+M$ .

I record non sono sparsi in questo caso perchè la crescita della tabella è una pagina alla volta.

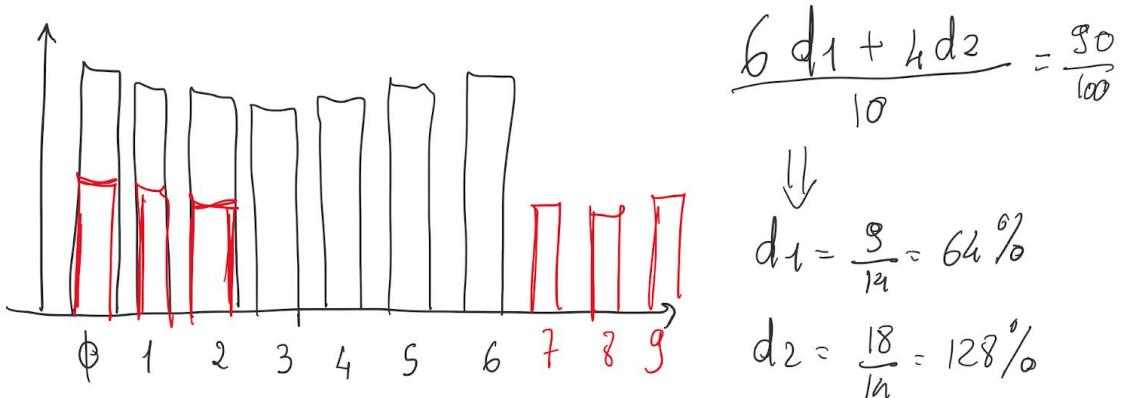
Il costo della equality search però non è buono come nel caso del virtual hash, qua infatti può crearsi una chain di record, quindi il numero di accessi in memoria può essere maggiore di 1.

Altro problema che si presenta:

- Ammettiamo di avere delle pagine piene al 90%, ci si presenta l'overflow e dobbiamo suddividere le pagine e creare pagine più piccole quando ho l'inserimento di un nuovo record.
- Il problema sta nel fatto che in questo modo avremo 6 pagine che hanno una certa densità  $d_2$  e 4 pagine che hanno densità  $d_1$ .
- Considerando che vogliamo pagine piene al 90% calcoliamo l'equazione e vediamo che in alcune pagine abbiamo una densità

molto bassa e in altre una densità alta, in alcuni casi abbiamo overflow e in altri casi abbiamo pagine sparse.

L'esempio:



In questo caso la metà delle pagine sono troppo piccole e la metà delle pagine sono troppo grandi, i record non sono distribuiti in modo uniforme ma basandosi su una funzione esponenziale.

### Conclusione:

L'hash organization è semplice da implementare **ma non supporta il range search**.

Nel caso della versione statica vorremmo mantenere le pagine occupate all'80% ma è molto difficile, esiste la versione dinamica che però è più complessa da gestire.

## Lezione 4: Tree Primary Organizations

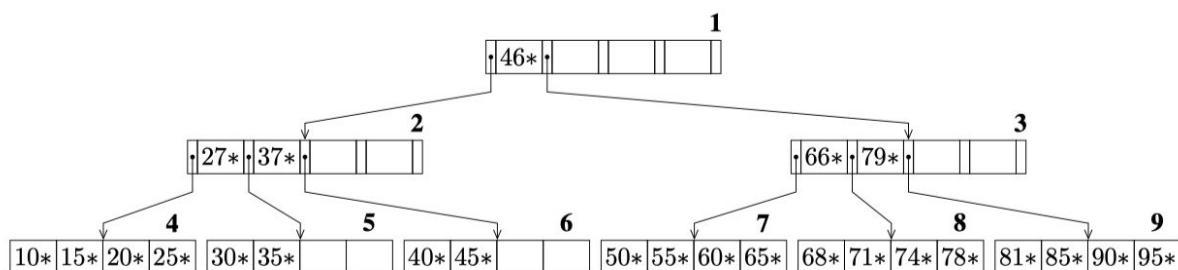
### Tree Organizations

Vogliamo provare a risolvere il problema della memorizzazione e richiesta di record utilizzando una organizzazione ad albero.

Potremmo utilizzare un classico Binary Tree, questo comporta dei problemi se non facciamo attenzione al modo in cui i dati vengono suddivisi nelle varie pagine. Abbiamo poi problemi quando dobbiamo mantenere bilanciato l'albero e quando dobbiamo mantenere le pagine abbastanza piene.

### B-Tree

La soluzione a questi problemi che mi consente di memorizzare e riprendere record identificabili tramite una chiave è il B-Tree.



Il B-tree funziona in questo modo:

- Abbiamo un ordine  $m$  ovvero l'ordine mi indica quanti sono i puntatori che escono da ogni nodo
- Ogni nodo contiene almeno  $(m/2)-1$  chiavi e al più ne contiene  $m-1$
- Per ogni nodo abbiamo in uscita un numero di puntatori pari al numero di figli
- Le foglie sono tutte allo stesso livello
- L'albero viene mantenuto bilanciato e abbiamo che l'altezza è pari al percorso root-foglia

Abbiamo quindi degli alberi che sono poco profondi ma che allo stesso tempo sono molto larghi.

Nel B-Tree scegliamo l'attributo che vogliamo supportare e organizziamo l'albero sulla base di questo attributo.

Dato il BTree c'è un'importante relazione tra il numero di chiavi N, l'ordine m e l'altezza h, abbiamo infatti:

$$\log_m(N + 1) \leq h \leq 1 + \log_{\lceil m/2 \rceil} \left( \frac{N + 1}{2} \right)$$

La base del logaritmo, che in un caso è m e nell'altro è m/2 è il numero medio di record che troviamo in una pagina.

### Operazioni con B-Tree:

- Equality Search: data la chiave k che vogliamo cercare, partiamo dal nodo root e troviamo il puntatore tale che  $k_i < k < k_{i+1}$ . Seguiamo questo puntatore e andiamo avanti fino a che non arriviamo ad una foglia  
La ricerca quindi mi costa al massimo h letture di pagine nel caso in cui non trovo quello che cerco, altrimenti il costo è minore di h.
- L'inserimento deve avvenire tenendo conto del fatto che può capitare un overflow nel caso in cui abbiamo già m-1 chiavi nella pagina.  
Prima di tutto faccio una ricerca e capisco dove deve finire la chiave.  
L'inserimento avviene sempre in una foglia, dato che stiamo organizzando l'albero in base ad una primary key, quando inseriamo un nuovo record non ne troviamo un altro con la stessa chiave.  
Se ho una foglia che ha già m-1 chiavi devo spartire la foglia dove andrebbe posizionata la nuova chiave. Spartendo la foglia prendo il suo elemento medio e lo porto al nodo superiore, questo diventa un elemento del nodo superiore e ci permette di avere un nuovo

puntatore. Agendo in questo modo manteniamo comunque la proprietà secondo cui l'albero deve rimanere bilanciato.

Nel caso ottimo l'inserimento costa  $h$  letture e 1 scrittura.

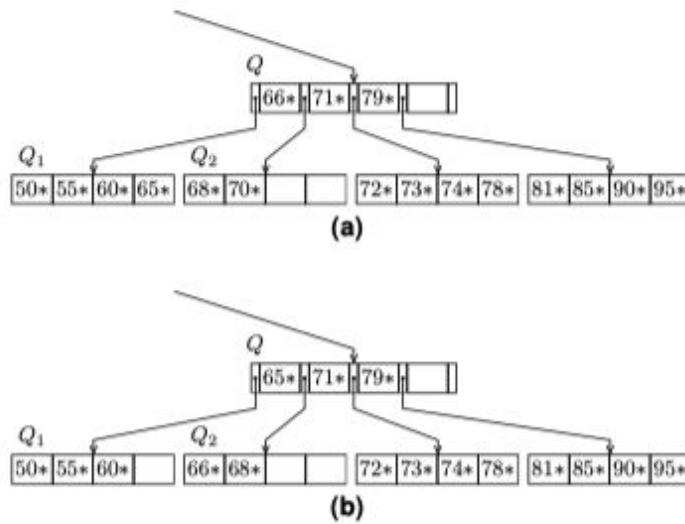
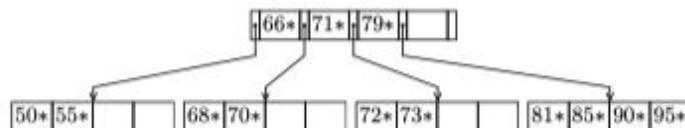
Nel caso pessimo l'inserimento mi costa  $h$  letture e  $2h+1$  scritture.

- Eliminazione di una chiave: si tratta del processo inverso rispetto all'inserimento. Può capitare che quando elimino la chiave non viene più rispettata la proprietà che il numero di chiavi nella pagina deve essere maggiore di  $(m/2)-1$  in questo caso viene fatto il merge di due pagine che si trovano vicine andando però a spostare nella foglia la chiave che si trova al livello superiore e che punta alla foglia.

In alternativa possiamo fare una rotazione prendendo l'elemento a sinistra della foglia e portandolo nel livello superiore e portando poi quello del livello superiore nella foglia.

Al caso ottimo abbiamo la chiave in una foglia e in questo caso abbiamo  $h$  letture e 1 scrittura, ammesso che non dobbiamo fare rotazioni. Poi c'è il caso in cui eliminiamo un nodo interno e in questo caso abbiamo  $h$  letture e 2 scritture.

Il caso peggiore l'abbiamo quando è richiesto il merge, perché abbiamo  $2h-1$  letture e  $h+1$  scritture.

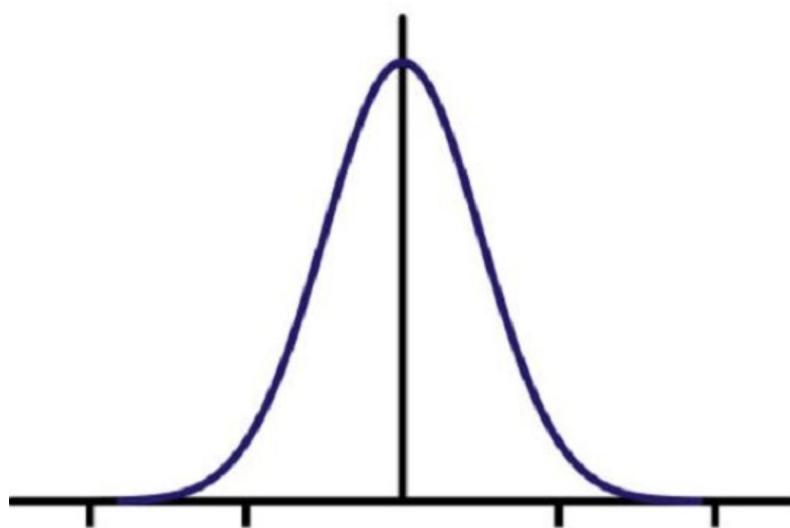
**Figure 5.5:** A rotation example

- Range Search: dobbiamo trovare gli elementi all'interno di un certo range di chiavi, calcoliamo il selectivity factor che è pari a  $sf(p) = (v_2 - v_1)/(k_{max} - k_{min})$ .  
Il numero di record da restituire è pari a  $E_{reg} = sf(p) * N$ .  
Alla fine il costo della range query è pari a  $C = sf * N_{nodes}$ .  
In questo caso abbiamo che la range search è migliore rispetto alla versione con hash ma la equality search è peggiore.

Il problema è che la formula per il calcolo del selectivity factor non funziona bene e funzionerebbe bene solamente se avessi una distribuzione uniforme dei dati, negli altri casi understima o overstima.

Ad esempio se abbiamo una distribuzione gaussiana dei dati, se i valori  $v_2$  e  $v_1$  sono ai lati della parte più alta della curva abbiamo una sotto stima del valore reale del selectivity factor, se invece i

punti sono nella parte più bassa allora abbiamo una sovra stima.



## B+ Tree

Il B+Tree è un albero simile al BTree, in questo caso però l'obiettivo è avere un albero che sia meno profondo ma più largo, in pratica vogliamo aumentare anche il valore dei puntatori che escono da ogni nodo.

L'idea in questo caso è di mantenere i record solamente nelle foglie e non nei nodi superiori. Nei nodi superiori viene messo il valore più alto del nodo inferiore ma questo non rappresenta un record e viene usato solamente per muoversi all'interno dell'albero.

I record che sono nelle foglie sono quindi organizzati in file sequenziali e sono ordinati, per questo sono chiamati anche index sequential.

Le foglie sono organizzate con una lista linkata in entrambe le direzioni e quindi è semplice eseguire operazioni come ad esempio il range search perchè arrivo alla prima foglia e poi vado avanti usando i puntatori da una foglia all'altra.

## Operazioni:

- Equality Search: la ricerca di una chiave richiede sempre un numero di accessi che è pari all'altezza  $h$  dell'albero

- Range Search: dobbiamo calcolare prima il selectivity factor che è  $sf = (v2-v1)/(kmax - kmin)$  e poi possiamo calcolare il costo del range search che è  $C = sf * NLeaf$ .
- Cancellazione del nodo: ho il costo della ricerca del nodo più devo modificare la foglia eliminando quella specifica chiave che ho trovato. Se è l'ultimo valore non ci sono problema e sopra lascio l'ultimo valore perchè tanto non è un record. Se eliminando faccio si che si vada in underflow allora devo applicare merge o rotazioni
- Inserimento: inserisco sempre nelle foglie, se creo un overflow allora devo fare la divisione in due e il valore nel mezzo lo porto al livello superiore.

## Index Organization

Un indice è una tabella con coppie  $\langle key, RID \rangle$  dove la key identifica un record e il RID è l'id del record e mi aiuta a ritrovare il record sul disco.

Possiamo utilizzare due tipi di index:

- Se abbiamo una heap organization, un indice è definito per ogni chiave e gli elementi dell'indice sono coppie  $\langle k, rid \rangle$
- Se la tabella è salvata con una primary organization dinamica usando la chiave primaria allora possiamo definire un indice sulle altre chiavi e abbiamo delle coppie  $\langle k, pki \rangle$  dove pki è la chiave primaria del record corrispondente.

Qua ho un vantaggio, ad esempio potrei avere una tabella di studenti con la matricola che è la chiave primaria, se trovo che nell'indice ho anche la matricola, non devo accedere alla tabella.

Per una tabella possiamo creare più di un indice ognuno con una chiave di ricerca differente.

Possiamo distinguere anche tra:

- Clustered Index: abbiamo un ordinamento sul valore K della chiave dei record che sono contenuti all'interno della tabella.

L'ordinamento è parziale perchè se aggiungo nuovi record questi non vengono posizionati in modo da mantenere l'ordine.

- Unclustered Index: i record sono posizionati in modo random nella tabella

Tipicamente l'indice viene organizzato con un B+Tree

Operazioni:

- Equality Search: il costo è  $C = Ci + Cd$  dove  $Ci$  è il costo per accedere alle foglie dell'indice e  $Cd$  è il costo per accedere alle pagine che contengono i dati.  
Per la equality search abbiamo un costo che è 1+1
- Per la range search bisogna fare una distinzione tra il caso in cui abbiamo un indice con clusterizzazione e senza:
  - Unclustered index: Per prima cosa bisogna calcolare il selectivity factor che è  $sf = (v2 - v1) / (kmax - kmin)$ .  
Poi bisogna considerare il costo per accedere alla foglia per prendere il RID e il costo per accedere alla pagina.  
Quindi abbiamo il costo per accedere alle foglie che è pari a  $CI = sf * NLeaf$  e il numero di record che soddisfano le condizioni invece è  $Erec = sf * Nrec$ .  
Quindi il costo totale è  $C = sf * NLeaf + sf * Nrec$ .  
Conviene o no usare l'index unclustered? Dipende, perchè se il costo  $C$  è minore del numero di pagine allora conviene, altrimenti conviene fare una scansione completa delle pagine. ( $sf * NLeaf + sf * Nrec < Npag$ ).
  - Clustered index: Anche in questo caso il costo viene stimato considerando il costo per accedere alle foglie e il costo per accedere alle pagine. Il numero di pagine da visitare per trovare i record che soddisfano le condizioni è  $Erec = sf * Npag$ .  
Il costo totale quindi diventa:  
 $C = sf * NLeaf + sf * Npag$ .

## Lezione 5: Non Key attribute organizations

Gli attributi Non Key sono quelli che non identificano univocamente un record. Quindi ad esempio se abbiamo 10 record, avremo 10 diversi valori per un certo attributo non key.

Dato un indice su un attributo non key è possibile svolgere le seguenti operazioni:

- Equality Search
- Range Search
- Boolean Search con AND e OR in cui usiamo le due operazioni precedenti

Avere un indice di questo genere serve perchè se considerassi solamente la primary organization non avremmo modo di risolvere velocemente le query sulle chiavi non primarie e dovrei fare una scansione completa dei dati.

I parametri che vengono usati con questo tipo di organizzazione:

- NKey è il numero di chiavi differenti che abbiamo nell'indice per un certo attributo
- Nleaf è il numero delle foglie che abbiamo nell'albero che equivale al numero di pagine dell'indice
- Nrec è il numero di record
- Npag è il numero di pagine differenti che servono per memorizzare tutti i record

### Inverted Index

Quando creiamo un indice su un attributo non ha senso andare a creare delle tabelle con  $\langle\text{valore}, \text{rid}\rangle$  perchè valore si ripete tante volte, quindi dobbiamo creare un inverted index ovvero una struttura in cui inseriamo  $\langle\text{valore}, \text{lista di rid}\rangle$ .

- Table

RID	Code	City	BY
1	100	MI	1972
2	101	PI	1970
3	102	PI	1971
4	104	FI	1970
5	106	MI	1970
6	107	PI	1972



- Indexes

City	n	RID-List
FI	1	4
MI	2	1 5
PI	3	2 3 6

Index on City

BY	n	RID-List
1970	3	2 4 5
1971	1	3
1972	2	1 6

Index on BY

La lista dei rid viene mantenuta ordinata, l'inverted index viene memorizzato in un B+Tree e quindi nelle foglie dell'albero avremo queste liste di RID che sono ordinate.

Lo spazio per memorizzare gli alberi che creiamo per gestire i vari indici è pari a Numero di indici per il numero di record per la lunghezza dei vari rid.

Spiegazione della Cardenas formula:

La Cardena Approximation mi indica una approssimazione del numero di pagine che dobbiamo leggere quando i RID sono ordinati.

La formula vera sarebbe:

$$\Phi(k, n) = n \left( 1 - \left( 1 - \frac{1}{n} \right)^k \right)$$

Dove abbiamo che  $1/n$  è la probabilità che una pagina contiene uno dei record,  $(1-1/n)^k$  è la probabilità che non ne contiene nessuno e  $(1 - ((1-1/n)^k))$  è la probabilità che la pagina contiene almeno un record.

La  $n$  davanti serve perchè così stimiamo il numero delle pagine che contengono almeno uno dei  $k$  record.

La cardenas formula la utilizziamo solamente se i rid sono ordinati.

In generale per le altre operazioni il costo complessivo in termini di pagine che vengono portate in memoria dipende dal costo per accedere alle foglie dell'indice ( $C_i$ ) e il costo per accedere alle pagine con i dati ( $C_d$ ).

## Operazioni:

Consideriamo le operazioni che vengono effettuate con un indice su un solo attributo

- **Equality Search:** assumendo di avere una distribuzione uniforme dei valori che abbiamo nell'indice dobbiamo considerare il costo come  $C_s = C_i + C_d$ .

Per prima cosa si deve valutare il selectivity factor, dato che abbiamo un certo numero di chiavi  $N_{key}$  differenti per l'attributo che stiamo considerando, avremo un selectivity factor che è pari a  $1/N_{key}$  e questo mi indica quante rid list dobbiamo leggere.

Il costo per accedere alle foglie dell'index è  $C_i = sf * N_{leaf}$  ovvero  $C_i = N_{Leaf}(I)/N_{Key}(I)$ .

Ora consideriamo due casi

- Se l'indice non è ordinato (**UNCLUSTERED**) dobbiamo stimare il numero di record che soddisfano la nostra condizione ovvero dobbiamo capire quanti RID ci sono nella rid list che vogliamo leggere, quindi calcoliamo  $E_{rec} = (N_{rec}/N_{Key})$ .

Per calcolare  $C_d$  dobbiamo poi considerare la Cardenas formula:  $C_d = Cardena(E_{rec}, N_{pag})$  ovvero dobbiamo prendere il minimo tra la lunghezza dei rid e la dimensione delle pagine. Non essendo i rid ordinati in questo caso potrei dover saltare da un punto all'altro e quindi potrei dover leggere più pagine di quante ne leggerei facendo una scansione.

- Se l'indice è ordinato (**CLUSTERED**) abbiamo che il costo di  $C_d = sf * N_{pag} = N_{pag}/N_{key}$ .

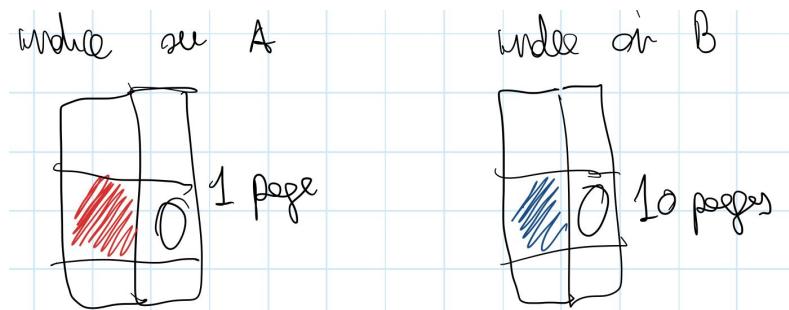
Ovvero consideriamo tutte le pagine che abbiamo, noi vogliamo leggere solamente una RID list, quindi dividiamo il numero delle pagine per il numero delle chiavi che dobbiamo e in questo modo riusciamo a capire quante pagine sono occupate in media da ognuna delle RID list.

- **Range Search:** in questo caso la condizione mi dice di cercare i valori entro un certo range. Il costo è sempre  $C_s = C_i + C_d$ . Il costo per l'accesso alle leaf dell'index è sempre  $C_i = s_f * N_{leaf}$ . Quello che cambia però è il modo in cui calcolo  $s_f$  ovvero la quantità di record che soddisfano la condizione.  
 $S_f = (v_2 - v_1) / (\max - \min)$ .  
 $C_d = \text{Numero di rid list da leggere} * \text{Numero di pagine a cui accediamo per vedere la rid list}$ .  
Poi consideriamo il numero delle liste di rid che devono essere lette ovvero calcoliamo  $s_f * N_{key}$ . Anche qui abbiamo poi una distinzione:
  - Se i dati sono unclustered allora dobbiamo usare la cardenas formula. Quindi il CD completo diventa  
 $C_d = s_f * N_{key} * \text{Cardena}((N_{rec}/N_{key}), N_{pag})$ . Nella formula della cardena abbiamo  $N_{rec}/N_{key}$  perchè i nodi non sono ordinati e quindi salto
  - Caso clustered: ora i dati sono ordinati quindi il numero delle pagine lo divido per il numero delle differenti chiavi e poi lo moltiplico per la quantità di liste che devo seguire.  
 $C_d = s_f * N_{key} * (1/N_{key} * N_{pag}) = s_f * N_{pag}$
  - Caso unclustered con rid non ordinati: in questo caso non si usa la cardenas formula.  
Quindi  $C_d = (s_f * N_{key}) * (N_{rec}/N_{key})$

## Indice su più di un attributo (Indice su A e su B)

Abbiamo due indici, uno su A e uno su B, succede che prima verifichiamo la condizione su A e otteniamo delle rid list e poi verifichiamo la condizione sull'indice B e ne otteniamo altre.

Dobbiamo intersecare le rid list per vedere quali sono i rid in comune che rispettano le due condizioni. Quindi, prendiamo i rid che rispettano la condizione A e li portiamo in memoria, creiamo una hash table e poi prendiamo le rid list di B e controlliamo quali sono nella hash table. Quindi, mi serve un buffer che mi contenga la pagina con i rid di A e poi una pagina in cui metto di volta in volta i rid di B per fare l'intersezione.



Il numero di rid che sono nell'intersezione è dato da  $Erec = Nrec * sf1 * sf2$ .

Ora consideriamo le operazioni da effettuare considerando che l'indice non è su un solo attributo ma è su più di un attributo (INDICE SU A+B). Consideriamo ad esempio un AND tra due condizioni, abbiamo due indici uno su un attributo A e uno su un altro attributo B.

Calcoliamo con i due indici due liste di RID che rispettano le condizioni, portiamo in memoria la più piccola e creiamo una hash table con i dati della lista, poi prendiamo la lista più lunga e controlliamo se i rid stanno nell'hash table. Quanto ci costa questo metodo?

Dobbiamo considerare  $Cs = Ci1 + Ci2 + \text{costo per accedere ai dati}$ .

Come si calcola C1 ovvero il costo per accedere al primo indice? La formula base è  $sf * Nleaf$ . In questo caso abbiamo due indici quindi avremo che  $Ci = sf(I1) * Nleaf(I1) + sf(I2) * Nleaf(I2)$ .

Il selectivity factor potrà essere molto differente tra i due attributi, invece il numero delle foglie sarà molto simile.

Per accedere ai dati si usa la Cardenas Formula ovvero  $\text{Cardenas}(Erec, Npage)$  dove Erec è il numero di record che mi aspetto. Uso la Cardenas Formula, i rid sono ordinati e non leggo mai una pagina due volte.

Come si calcola Erec?

La formula generale è  $Sf^*Nrec$  ma in questo caso dobbiamo considerare più selectivity factor.

Se assumiamo l'indipendenza delle due condizioni il selectivity factor è il prodotto dei due selectivity factor.

Va bene assumere l'indipendenza?

Dipende dagli attributi, se c'è un'alta correlazione non posso avere un calcolo corretto del sf e l'errore può essere grande, allo stesso tempo però non possiamo fare di meglio quindi prendiamo comunque il selectivity factor.

Quindi  $Erec = sf1 * sf2 * Nrec$ .

Il costo totale è  $Cs = Ci1 + Ci2 + Cardenas(Erec, Npage)$  ammesso che i rid sono ordinati.

Se invece i rid non sono ordinati non uso la cardenas formula e il costo diventa  $sf1 * sf2 * Nrec$  e quindi questo può essere un problema, quindi si ordinano sempre i RID quando c'è la possibilità.

Quindi le formule in questo caso sono:

$$C_I = \sum_{j=1}^k [s_f(\psi_j) \times N_{leaf}(I_j)]$$

Con  $Sf$  uguale a  $(1/Nkey)$  se faccio equality search e  $v2-v1/kmax-kmin$  se faccio range search.

$$E_{rec} = \left\lceil N_{rec}(R) \prod_{j=1}^k s_f(\psi_j) \right\rceil$$

Il numero di pagine che vengono accedute è:

$$C_D = \lceil \Phi(E_{rec}, N_{pag}(R)) \rceil$$

## Combined Index

Qui la situazione è diversa perchè abbiamo una sola tabella in cui però abbiamo un indice su due differenti attributi.

Ad esempio potrei avere un unico indice su due attributi come ad esempio Age e Salary. Per alcune operazioni sarà meglio avere un unico indice combinato su age e salary che avere invece due indici separati.

Se consideriamo una equality search:

- Il costo Ci per l'accesso all'index è il prodotto di Selectivity factor del primo indice, per il secondo selectivity factor e per il numero di foglie.  $C_i = sf_1 * sf_2 * N_{leaf}$ .

Se entrambi gli attributi non hanno un buon selectivity factor, devo considerare che prendendo gli indici separatamente avrei un costo più alto rispetto al prendere un unico indice combinato.

Se siamo interessati solamente ad eseguire l'equality search non siamo interessati all'ordine di age e salary nell'indice, chi c'è prima non ci interessa.

Se invece vogliamo supportare altre operazioni come ad esempio equality sull'attributo A e range sull'attributo B, allora in questo caso diventa importante che l'attributo A sia messo prima di B.

In questo caso abbiamo sempre lo stesso costo per accedere all'indice che sarà  $C_i = sf_1 * sf_2 * N_{leaf}$ .

Se vogliamo supportare anche il range search su A e su B, il risultato non sarà buono allo stesso modo, il motivo è legato al fatto che iniziamo a saltare da una pagina all'altra e da ognuna delle pagine leggiamo solamente pochi rid, quindi in questo caso il combined index non è buono da usare.

Un'altra proprietà interessante del multi index è che se noi mettiamo come primo attributo Age, avremo una prestazione quasi uguale al solo indice su Age nel momento in cui svolgiamo delle equality search su Age.

L'ordine in questo caso è importante perchè se age è primo posso fare query su age ma non avrò le stesse prestazioni su salary perchè age sarà ordinato ma salary no.

Le prestazioni saranno quasi uguali a quelle dell'indice singolo perché nell'indice singolo troviamo i vari RID ordinati nell'inverted index mentre in questo caso non abbiamo i rid ordinati e quindi vanno ordinati, questo però non è un gran problema se prima li portiamo in memoria.

Operazioni:

- Equality Search su entrambi gli attributi:

Per fare l'equality search con il combined index consideriamo sempre la formula  $C_s = C_i + C_d$ .

Il costo per l'accesso all'index, ovvero  $C_i$  è pari a  $N_{leaf} * S_f$ .

Ammesso di avere due attributi quindi  $C_i = N_{Leaf} * S_f(A) * S_f(B)$ .  
 $S_f$  in questo caso è  $1/N_{key}$ .

Il costo per l'accesso ai dati invece è la cardenas Formula in cui andiamo a considerare l'accesso a  $N_{rec}/N_{key}(A)*N_{key}(B)$  pagine oppure a  $N_{pag}$  pagine.

- Range Search su entrambi gli attributi:

In questo caso il discorso è più complesso, non possiamo usare la formula  $C_i = N_{Leaf} * S_f(A) * S_f(B)$  perchè non possiamo fare assunzioni sull'ordinamento degli indici. Di norma gli indici quando sono presi da soli sono ordinati, qua invece l'indice è combinato e quindi gli indici sono spartiti e non ordinati, i rids non sono davvero consecutivi.

Per prima cosa dobbiamo andare a calcolare il numero delle pagine che ci servono per leggere i blocchi in cui abbiamo i record che rispettano la condizione sull'attributo A.

Questo numero di aree gialle è pari a  $S_f * N_{key}$ , abbiamo in tutto  $N_{key}$  ma considero solamente quelle interessanti, questo calcolo va bene perchè noi creiamo dei blocchi che sono dei gruppi consecutivi di dati che starebbero nelle stesse rid list. Quindi  $S_f * N_{key}$  equivale anche al numero delle rid list.

Ora dobbiamo considerare che in ognuno di questi blocchi che leggo quante pagine mi servono per leggere tutti i rids?

Abbiamo  $N_{leaf} * S_f(B) * (1/N_{key}(A))$ . Quindi consideriamo il numero

di pagine dell'indice, poi consideriamo il selectivity factor su B per la range search su B e su A invece consideriamo il selectivity factor come se stessimo svolgendo una equality search.

Quindi equality su A e range su B.

Complessivamente abbiamo:

$$Ci = Sf(A) * Nkey * Nleaf * Sf(B) * (1/Nkey(A)).$$

Ora dobbiamo considerare anche il costo per l'accesso ai dati.

I modi più estremi per accedere ai dati sono due:

O leggiamo tutte le rid list, la portiamo ognuna in memoria e per ogni rid list scansioniamo i dati oppure portiamo tutto in memoria, ordiniamo e creiamo una sola rid list. Non vengono fatte intersezione perchè tutte le rid list vengono considerate e non abbiamo intersezione.

Se portiamo in memoria, ordiniamo e abbiamo una sola rid list ordinata allora possiamo sfruttare la Cardenas formula che mi permette di limitare gli accessi alle pagine perchè nella lista unica avremo vicini tra loro dei record che accediamo con un solo accesso ad una pagina, se invece accediamo di rid list in rid list, per accedere a record della stessa pagina potremmo fare più di un accesso.

Quindi per il caso degli accessi alle singole rid list consideriamo che dobbiamo accedere un numero di rid list che è pari a:

$Nkey(A) * Sf(A) * Nkey(B) * Sf(B)$ . Per ognuna delle rid list poi dobbiamo leggere una quantità di record che è pari alla cardenas Formula di  $(Nrec/Nkey(A) * Nkey(B), Npag)$ .

Se invece abbiamo una sola rid list lunga allora abbiamo una sola lista da leggere e nella cardenas formula consideriamo  $Nrec * Sf(A) * Sf(B)$  e  $Npag$ .

## Bitmap Index

Partendo da una tabella possiamo creare un indice su un solo attributo della tabella e questo mi porta ad avere un inverted index in cui inseriamo il valore e poi la lista dei rid corrispondenti.

Possiamo convertire questo inverted index in una bitmap index ovvero un indice in cui le varie rid list hanno tutte la stessa lunghezza e mettiamo 1 se quel rid è presente nella rid list 0 altrimenti.

RID	StudCode	City	BirthYear
1	100	MI	1972
2	101	PI	1970
3	102	PI	1971
4	104	FI	1970
5	106	MI	1970
6	107	PI	1972

City	n	RID Lists		
FI	1	4		
MI	2	1	5	
PI	3	2	3	6

Index on City

City	Bitmaps					
FI	0	0	0	1	0	0
MI	1	0	0	0	1	0
PI	0	1	1	0	0	1

Questo encoding ha vari vantaggi e in alcune situazioni utilizzare la bitmap può comportare anche dei miglioramenti in termini di performance.

Ad esempio se abbiamo le rid list in memoria e vogliamo fare una intersezione tra le varie rid list, è possibile farlo con le bitmap in modo rapido perché viene sfruttato l'hardware. Questo è valido per i database che stanno completamente in memoria, in particolare questo metodo viene utilizzato per l'analytics.

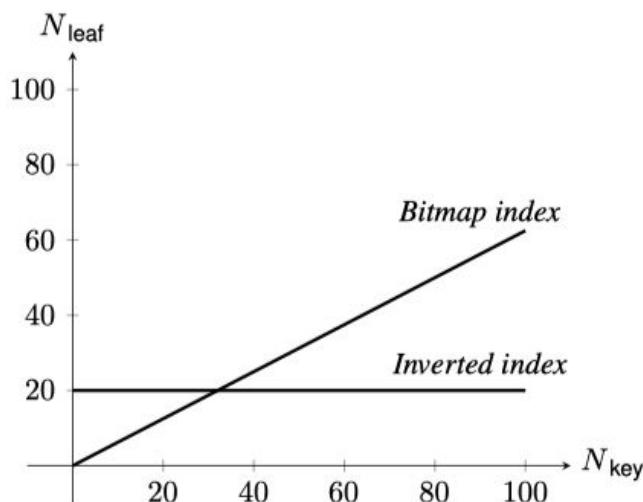
Nei database tradizionali le bitmap non sono molto usate, vengono

utilizzate specialmente quando dobbiamo combinare delle condizioni in AND e quando i dati sono statici.

Confronto tra la memoria necessaria per l'inverted index e per la bitmap:

- Nel caso dell'inverted index abbiamo che il numero delle foglie dipende da:  $N_{leaf} = (N_{key} \cdot L_k + N_{rec} \cdot L_{rid}) / D_{pag} = (N_{rec} \cdot L_{rid}) / N_{pag}$ . Ovvero il numero delle foglie dipende solamente dalla lunghezza dei rid e dalla dimensione delle pagine ma non dal numero di chiavi che abbiamo.
- Nel caso della bitmap invece il numero delle foglie è  $N_{leaf} = (N_{Key} \cdot L_k + N_{key} \cdot N_{rec} / 8) / D_{page} = N_{rec} \cdot N_{key} / (D_{pag} \cdot 8)$ .

Quindi l'inverted index non dipende dal numero delle chiavi mentre invece la bitmap sì. I due casi hanno lo stesso consumo di memoria se  $N_{key} = 8 \cdot L_{rid}$ .



## Multidimensional Data Organization

Possiamo trovarci con un database in cui ogni record è un punto su una mappa, ogni record ha due attributi, la latitudine e la longitudine.

Su un database di questo genere vorremmo poter fare delle ricerche range-range su entrambi gli attributi. Ad esempio vorrei trovare i ristoranti che sono nel raggio di 10km.

Se salviamo in modo banale i punti in un Btree, possiamo suddividere in punti tracciando delle linee verticali e mettendo nelle stesse pagine i punti sulla stessa linea. In questo modo però i ristoranti non sono memorizzati nelle pagine come sono vicini nella realtà e quindi potrei trovare all'interno di una pagina dei ristoranti che sono in due punti molto diversi tra loro.

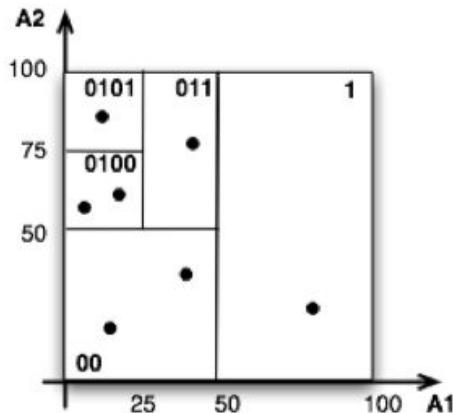
Quindi l'idea migliore è suddividere lo spazio in modo che ogni parte contenga lo stesso numero massimo di ristoranti e in modo che le pagine contengano ristoranti che sono effettivamente vicini tra loro.

Le zone in cui dividiamo il piano non sono di dimensioni uguali tra loro, suddividiamo sulla base della quantità di ristoranti che sono contenuti all'interno.

Prima dividiamo in verticale dividendo in due il piano, poi la parte in cui ci sono più di due punti viene a sua volta divisa in due parti in orizzontale.

Al termine di divisioni in verticale con divisioni in orizzontale e ogni volta assegno un valore 0 o 1 alle due parti che abbiamo creato.

In questo modo ogni sezione ha una sequenza di 0 e di 1 che la identifica.

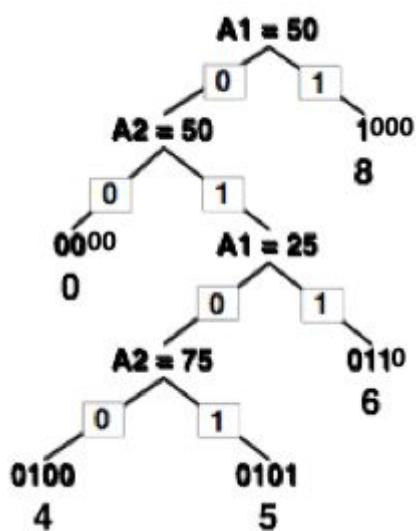


Partendo dalla divisione in varie sezioni labellate con 0 e 1, creiamo un partition tree che in realtà poi non viene memorizzato in memoria (conosciamo il numero dei livelli) e viene solamente utilizzato per assegnare un valore numerico alle varie sequenze di bit. Questo partition Tree può essere utilizzato anche per le operazioni di ricerca di un punto o di un range di punti.

Il Gtree invece viene memorizzato su disco.

Data la quantità M di split che vengono effettuati, la lunghezza dei vari encoding è M e nel caso in cui dovesse essere minore di M allora viene aggiunto del padding all'inizio dei vari encoding aggiungendo tanti 0 quanti bastano per arrivare alla lunghezza M.

Alla fine il partition Tree è il seguente:

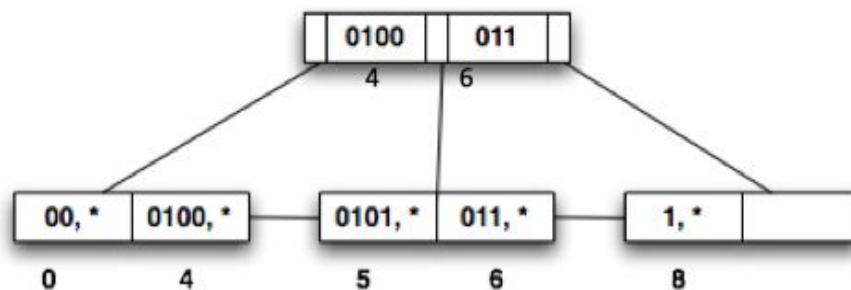


Ora possiamo creare il GTree corrispondente che è simile ad un Btree. I nodi interni del Gtree hanno la forma ( $p_0, s_1, p_1, s_2, p_2, \dots, s_m, p_m$ ) quindi abbiamo che  $p_0$  è il puntatore alla prima foglia che contiene i valori che sono minori di  $s_1$ ,  $p_1$  è il puntatore alla seconda foglia che ha i valori maggiori di  $s_1$  e minori di  $s_2$  e così via.

Se abbiamo le coordinate di un punto, ci muoviamo all'interno del partition tree per trovare la sequenza che codifica quel punto, a questa sequenza corrisponde un elemento in una delle foglie del GTree.

Se dobbiamo fare una **ricerca all'interno di un range**, ad esempio se dobbiamo considerare una zona di 10 km intorno a noi, avremo due punti che delimitano questa zona sulla X e sulla Y, troviamo i due punti all'interno del partition Tree e quindi abbiamo le due sequenze binarie che codificano i due punti.

Poi andiamo nel GTree e prendiamo quelle due sequenze, le foglie che vengono visitate sono quelle che contengono i due punti e quelle che sono nel mezzo tra i due.



Per esempio se abbiamo che i punti che cerchiamo sono in 0 e in 6 allora dovremo controllare 0,4,5,6.

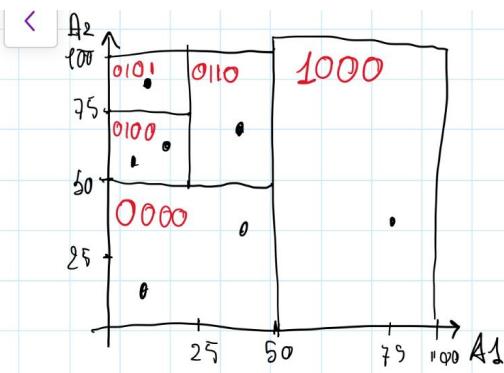
### Inserimento di un punto

L'inserimento di un nuovo punto viene effettuato seguendo questi passaggi:

- Per prima cosa si inserisce il punto nel grafico e si vede se nella zona in cui l'abbiamo inserito ci sono già abbastanza punti, se non superiamo il limite inseriamo senza problemi, altrimenti dobbiamo dividere la zona e quindi assegniamo un nuovo valore binario alle due aree.
- Il nuovo valore binario può superare la lunghezza del valore binario massimo che avevo in precedenza (M) oppure no. Se non lo supera non dobbiamo fare niente, altrimenti dobbiamo aumentare il valore di M e quindi a tutte le altre codifiche viene aggiunto uno 0 in fondo.

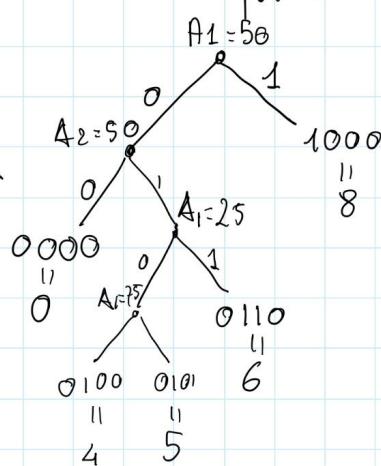
- Ora cerchiamo nel Gtree la foglia n cui andrebbe inserito il nuovo valore binario. Se l'inserimento avviene in una foglia e non superiamo la quantità massima di elementi all'interno di una foglia allora non ci sono problemi, altrimenti va spartita la foglia e quindi aggiungiamo un nuovo livello all'interno del Gtree.

Esempio di creazione del Gtree e di inserimento:



$$M = 4$$

Costruiamo il partition tree:



$$\emptyset = \emptyset \emptyset \emptyset \emptyset$$

$$4 : 0100$$

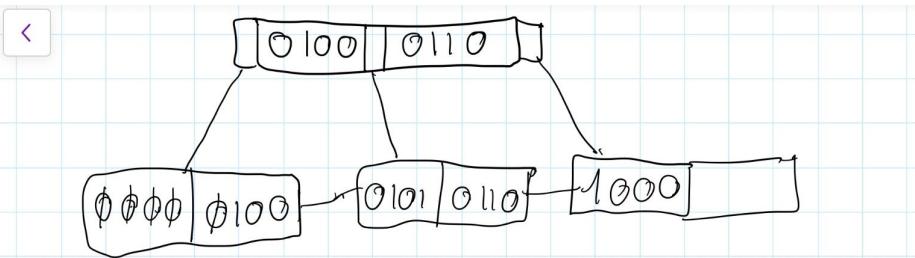
$$5 : 0101$$

$$6 : 0110$$

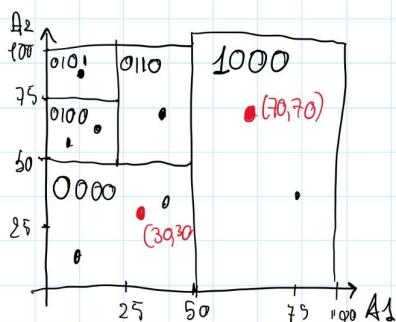
$$8 : 1000$$

Ora creiamo il Gtree  
Considerando che ogni  
pagina ha dim = 2, e ogni  
node ha la forma  
 $P_0 S_0 P_1 S_1 \dots S_m P_{m+1}$

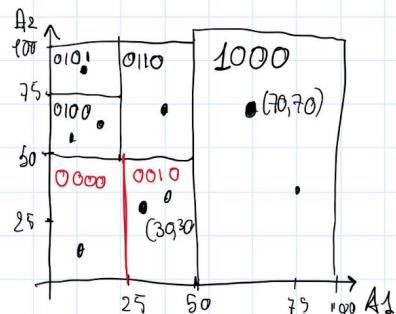
Po So  $P_1$  si - - Sm  $P_{m+1}$



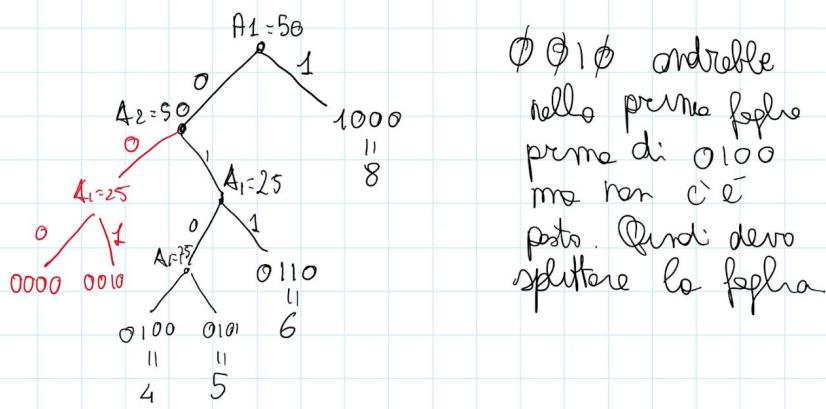
Se inseriamo il punto  $P_1 = (70, 70)$  non c'è problema perché ho solo 1 punto in quell'area



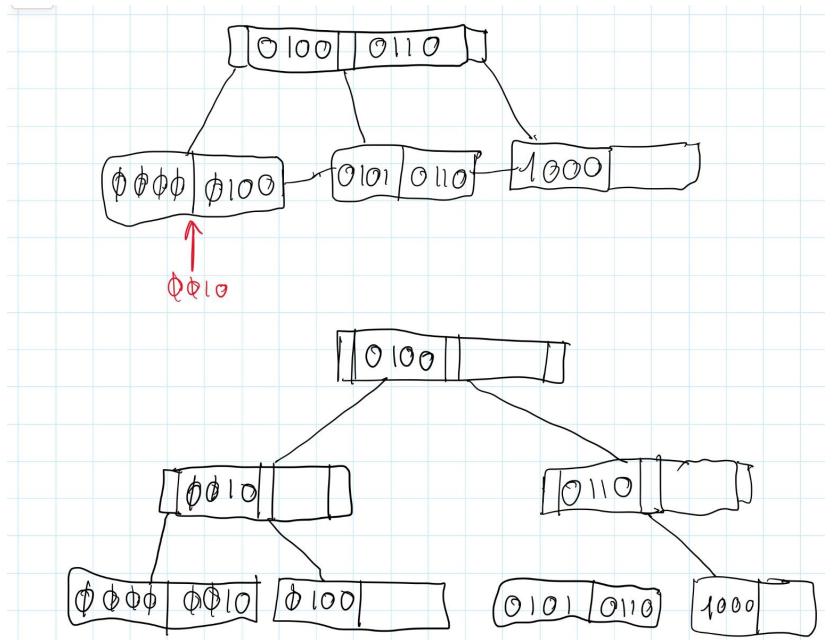
Se però aggiungo  $(30, 30) \rightarrow$  problema  
perché ho 3 punti. Quindi devo dividere



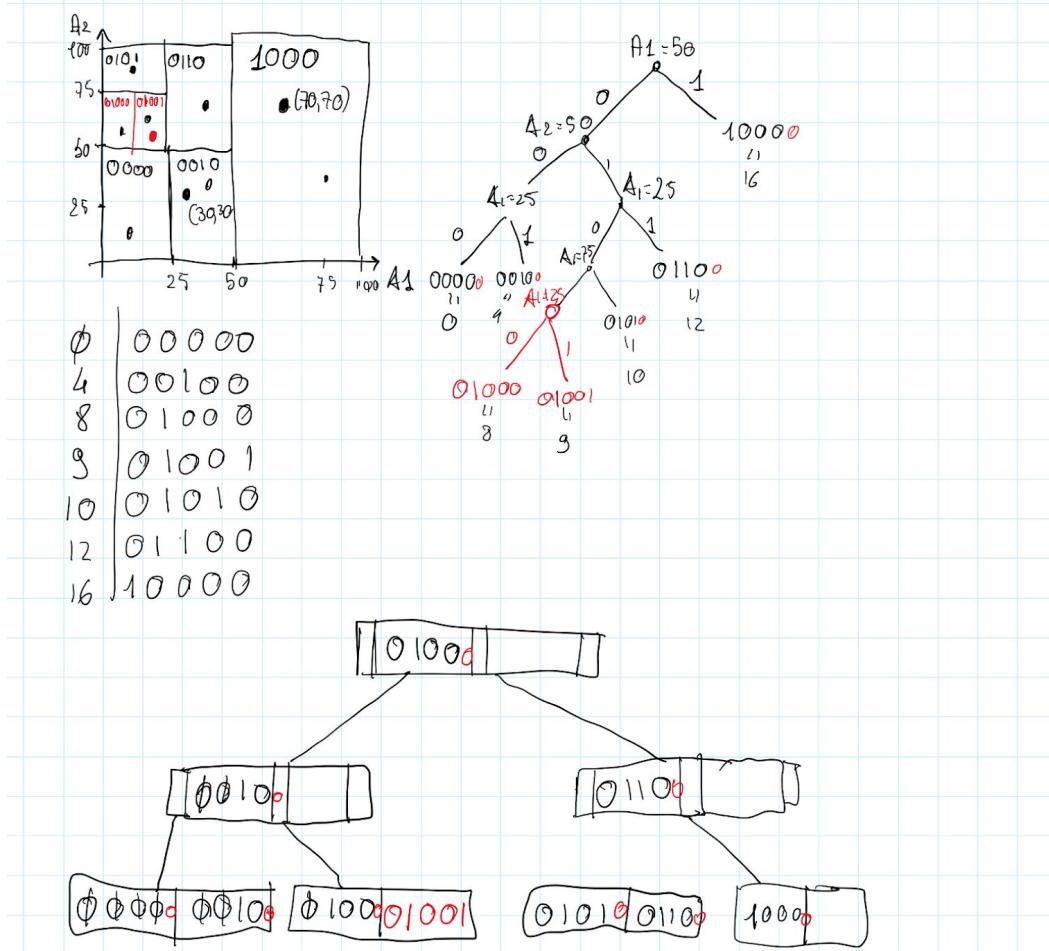
Questo divisione comporta una nuova foglia  
nel partition tree



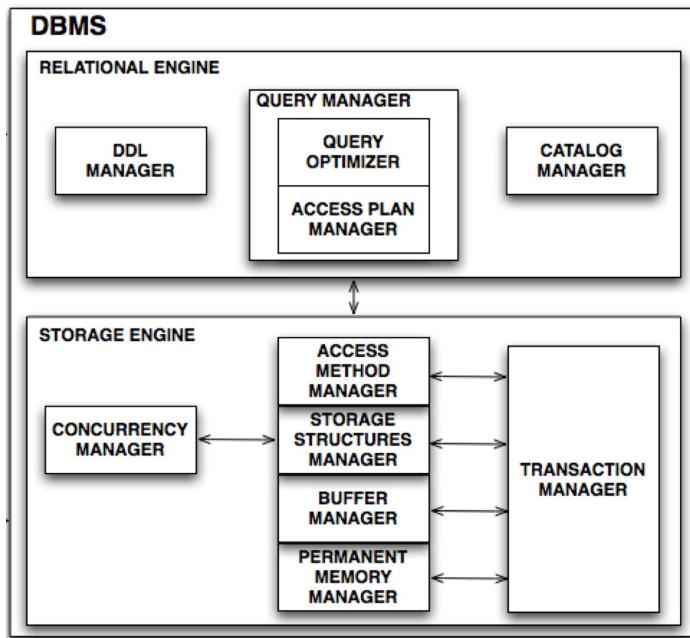
$\emptyset \emptyset 1 \emptyset$  andrebbe  
nella prima foglia  
prima di '0100'  
ma non c'è  
posto. Quindi devo  
splitsare la foglia



Alcuni inserimenti possono comportare anche un aumento dello  $H$ .



## Lezione 6: Access Methods Manager



L'architettura dei DBMS è divisa in due parti principali:

- Relational Engine: include moduli che supportano l'esecuzione di comandi SQL e interagisce con lo Storage Engine
- Storage Engine: include i moduli che permettono di eseguire operazioni sui dati che sono memorizzati.

Lo storage engine non è accessibile all'utente che utilizza il database che invece ha accesso al relational engine. Le operazioni che vengono svolte dallo storage engine dipendono dalla struttura dati che viene utilizzata per memorizzare i dati nella memoria permanente.

Lo storage engine ci permette di eseguire:

- Operazioni sullo heap, ad esempio l'apertura e la chiusura, la ricerca di un certo record dato il rid, l'eliminazione, l'aggiornamento e l'inserimento di un record. Per la ricerca tramite RID ho necessità di avere un indice che mi restituisca il RID che sto cercando che poi passo a questo metodo che mi restituisce il record vero e proprio
- Ci sono operazioni sugli indici

- Ci offre la possibilità di eseguire una scansione dello heap con un iteratore. Grazie a questo iteratore (è un pattern) possiamo avere a disposizione delle operazioni per muoverci al prossimo elemento dello heap, poi abbiamo una operazione che ci dice se abbiamo ancora dei record da vedere all'interno dello heap. Possiamo poi tornare all'inizio dello heap e ripetere la scansione e anche chiudere lo scanner e rilasciare le strutture dati utilizzate per la scansione.
- Sugli indici possiamo fare la stessa cosa che abbiamo fatto con lo heap, quindi abbiamo una scansione con uno scanner.  
Quando creiamo uno scanner per la ricerca all'interno dell'indice, passiamo due parametri, uno è la prima chiave e una è l'ultima. Se abbiamo l'indice sulla chiave primaria avremo un solo valore che verrà restituito, questo non vale se abbiamo uno scanner che lavora su un indice con chiavi non primarie. In questo caso abbiamo un range search. Non ha senso utilizzare la range search con un range molto grande perché ha senso solamente se il sf è minore di 1/capacity.

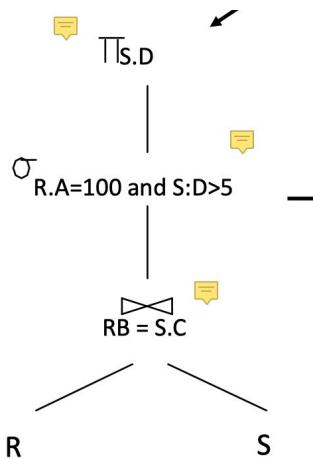
C'è poi una parte dello storage engine ovvero l'Access Method Manager, un livello che fornisce un'interfaccia allo storage engine.

Quindi il relational engine accede alle varie operazioni mediante questa interfaccia.

L'access Method Manager ci fornisce degli operatori per accedere a organizzazioni primarie e agli indici e si occupa anche del trasferimento dei dati dalla memoria permanente alla memoria principale.

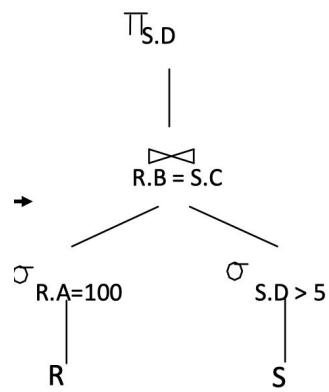
## Esecuzione dell'SQL

Una volta che abbiamo una query SQL, la prima cosa che facciamo è una traduzione nel linguaggio dell'algebra, abbiamo quindi i vari operatori per eseguire la join, la proiezione e la restrizione.



### Logical Plan

Una volta che abbiamo fatto questo primo passo, possiamo procedere con alcune manipolazioni logiche che portano ad una ottimizzazione della query.



### Logical transformation

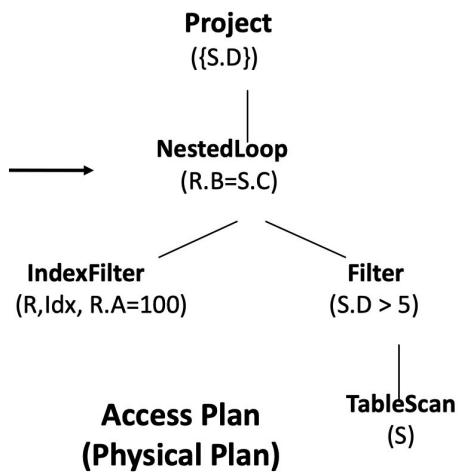
Abbiamo ottimizzazioni logiche che non dipendono dalle strutture dati che utilizziamo per memorizzare i record, nonostante questo possiamo comunque fare delle modifiche alla query perchè sappiamo che è meglio fare alcune operazioni prima di altre.

Le ottimizzazioni logiche sono molto poche e non fanno altro che muovere le varie operazioni tra i livelli dell'albero.

La maggior parte delle ottimizzazioni dipendono dal modo in cui i dati sono stati memorizzati sul disco.

La terza operazione che effettuiamo consiste nel trasformare l'albero dell'algebra in un physical tree ovvero in un albero simile a quello di

prima in cui però sostituiamo le operazioni con delle operazioni fisiche in cui indichiamo l'algoritmo che utilizziamo per svolgere l'operazione.



Un esempio è la join che può essere eseguita in vari modi, con un nested loop ad esempio.

La restriction possiamo eseguirla scansionando tutta la tabelle e usando un filtro oppure possiamo selezionare all'interno dell'indice usando un filtro.

La stessa operazione quindi ammette più di un operatore fisico e quindi vuol dire che abbiamo tanti differenti tipi di access plan che posso generare.

Possiamo generare tutti gli access plan, per ognuno calcoliamo i costi e poi alla fine si utilizza il migliore.

Questa generazione completa dei vari access plan si può fare solamente se l'access plan non è grande, se diventa troppo grande avremmo un costo esponenziale e quindi valutare il costo diventa troppo costoso.

Una volta che ho terminato la creazione dell'access plan devo eseguirlo. Per l'esecuzione abbiamo dei pattern differenti:

- Pipeline: generiamo un record dai due rami in basso, quando arriviamo al nested loop controllo se matchano e in caso porto in alto il record allo step successivo. Abbiamo un flow dei dati che va dal basso all'alto

- Materialization approach: in questo metodo tutti i dati vengono analizzati nei singoli step, quindi non abbiamo 1 record alla volta ma tanti record alla volta.

La maggior parte dei database utilizzano un approccio di tipo pipeline perchè abbiamo risultati intermedi e perchè utilizza meno spazio rispetto all'altro approccio. Infatti il materialization approach potrebbe dover memorizzare i dati intermedi anche sul disco se i dati sono tanti e non entrano tutti in main memory.

Utilizzando la pipeline abbiamo due possibili approcci per lavorare sui dati:

- Pull model: l'attività inizia dal root che chiede un record agli step che sono sotto, poi si va avanti a chiedere fino a che non si arriva alle foglie, è come una visita dell'albero.
- Push Model: lavoro dal basso all'alto

Tra i due il modello tradizionale è il pull model, il database chiede il prossimo record e quelli sotto lavorano.

## Lezione 7: Relational Operations

Vogliamo considerare l'implementazione delle operazioni fisiche. Consideriamo l'implementazione dell'operazione di selezione, sappiamo che il DBMS memorizza in un catalog delle informazioni riguardo alle tabelle del database e agli indici. In particolare viene memorizzato per ogni indice una stima del numero di chiavi che abbiamo.

Quindi, data l'operazione di selezione, una prima ottimizzazione la possiamo fare valutando il valore del selectivity factor.

Dato che il catalog mi stima il numero di chiavi di ogni indice possiamo capire se il selectivity factor è buono o no.

La stima che facciamo con il selectivity factor può anche essere sbagliata perché potremmo avere una distribuzione non uniforme.

Quindi in alcuni casi c'è il rischio di fare un numero di accessi che risulta maggiore rispetto al numero delle pagine che abbiamo.

Il selectivity factor per le operazioni base di uguaglianza, maggiore/minore e range search è il solito:

$$\begin{aligned}s_f(A = v) &= \frac{1}{N_{\text{key}}(A)} \\ s_f(A > v) &= \frac{\max(A) - v}{\max(A) - \min(A)} \\ s_f(A < v) &= \frac{v - \min(A)}{\max(A) - \min(A)} \\ s_f(v_1 < A < v_2) &= \frac{v_2 - v_1}{\max(A) - \min(A)}\end{aligned}$$

Invece per l'operatore di Join è differente, assumiamo di dover fare la join e di avere due attributi A e B. Per A abbiamo 10 valori differenti mentre per B ne abbiamo 100.

Ammettiamo che A e B siano ad esempio tabelle di cognomi, il selectivity factor va calcolato pensando alla probabilità.

Bisogna considerare la probabilità che A=B, dato che abbiamo da una parte 100 valori possibili e dall'altra solo 10, vuol dire che ci sono

$1/(100*10)$  possibilità di “successo” ovvero  $1/100$  possibilità che il family name di A sia uguale a quello che vediamo in B.  
 Quindi la formula è  $1/\max(N_{\text{keyA}}, N_{\text{keyB}})$  perchè dobbiamo considerare il numero più alto di chiavi perché è il più importante.

$$s_f(A = B) = \frac{1}{\max(N_{\text{key}}(A), N_{\text{key}}(B))}$$

Nel caso di un AND devo considerare il prodotto dei selectivity factor delle due condizioni, in questo caso il problema può essere la correlazione tra i due attributi e quindi il selectivity factor può risultare completamente sbagliato.

$$s_f(\psi_1 \wedge \psi_2) = s_f(\psi_1) \times s_f(\psi_2)$$

Se invece consideriamo l'or abbiamo che il selectivity factor viene calcolato sommando il selectivity factor di entrambi gli attributi e togliendo i casi in cui abbiamo “successo” in entrambi perchè altrimenti conterei questi casi due volte.

Dato che il selectivity factor viene utilizzato per capire se devo usare l'indice o no e dato che il valore che esce fuori deve essere basso, è difficile che sommando due selectivity factor si abbia un valore valido, quindi è difficile che poi utilizzi l'index dopo questo calcolo.

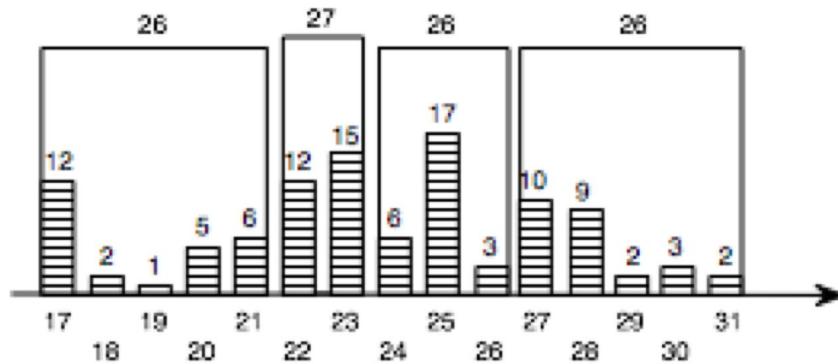
$$s_f(\psi_1 \vee \psi_2) = s_f(\psi_1) + s_f(\psi_2) - s_f(\psi_1) \times s_f(\psi_2)$$

Sappiamo che se i dati non sono distribuiti in modo uniforme, il selectivity factor ci restituisce un risultato negativo che non può essere preso in considerazione. Come possiamo risolvere questo problema?

Viene creato un istogramma in cui si rappresentano i valori dell'attributo, se ad esempio abbiamo l'attributo Voto relativo ad un esame abbiamo un istogramma in cui rappresento la distribuzione e ovviamente non sarà uniforme, ci saranno alcuni voti più comuni e altri che invece sono meno comuni.

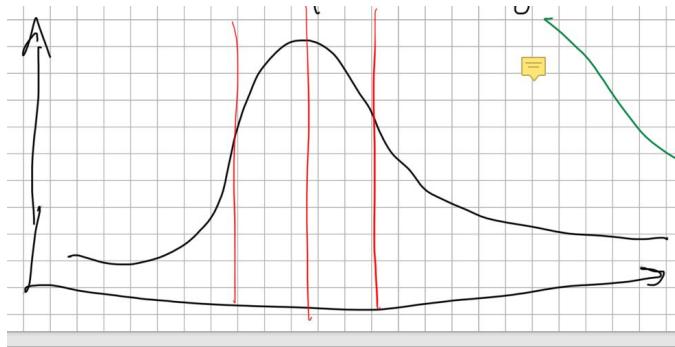
Quando voglio calcolare il selectivity factor di una certa query vado nell'istogramma e calcolo il valore reale del selectivity factor come numero di valori in uno stick / numero totale di record.

Ora se vogliamo rendere più uniforme l'istogramma abbiamo la possibilità di unire i vari stick in modo che ci forniscano una soluzione uniforme. Questa tecnica si chiama equi Height.



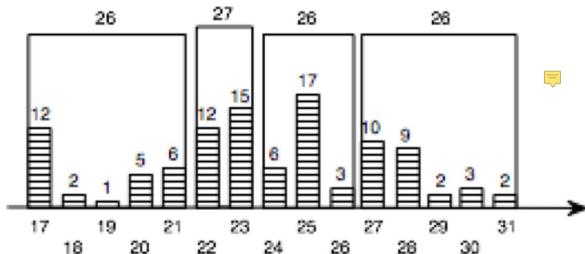
L'alternativa consiste nella tecnica Equi-Width che divide in parti uguali in larghezza, questa però non sempre ci fornisce una distribuzione che è più uniforme di quella di partenza.

Quindi alla fine si preferisce utilizzare la equi-Height che funziona bene se i valori sono sparsi e abbiamo una distribuzione a campana.



Se vogliamo calcolare il selectivity factor considerando l'utilizzo della tecnica equi-Height, data la query prendiamo lo stick in cui si trova il valore che vogliamo considerare e poi consideriamo la somma dei valori presenti all'interno del gruppo e la dividiamo per il numero di stick di quel gruppo.

Poi dividiamo per il numero complessivo dei record:



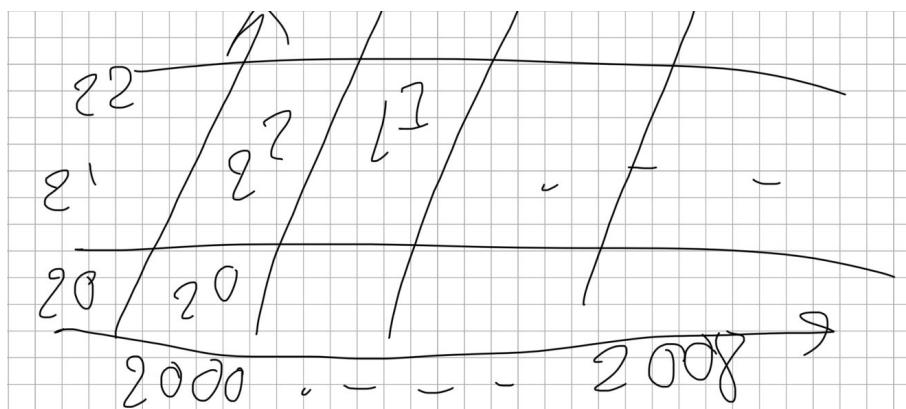
Ad esempio se chiediamo l'uguaglianza con 24, prendiamo il blocco in cui sta 24, ci sono 26 record, quindi calcoliamo  $(26/3)$  e poi questo lo dividiamo per 105 che è il numero complessivo dei record.

Se ho una range query invece devo cercare i blocchi in cui stanno gli stick agli estremi della range query e poi considero tutti quelli in mezzo. Se gli estremi sono nel mezzo del blocco, devo considerare solamente una parte dei dati presenti all'interno del blocco.

Se vogliamo avere una stima anche per quelle query in cui abbiamo una correlazione tra gli attributi, possiamo memorizzare un istogramma anche per la correlazione di due attributi. Assumiamo di avere gli attributi Birth Date e Grade che sono correlati.

Memorizziamo per ogni anno di nascita il numero di persone che hanno preso quello specifico voto.

Quindi possiamo memorizzare un istogramma della correlazione.



All'interno di questo istogramma con il numero di occorrenze dei vari voti abbiamo la possibilità di dividere i due assi in intervalli di una certa

dimensione. Gli intervalli non devono essere troppo piccoli perché altrimenti avrei aggiornamenti continui ma non devono neanche essere troppo grandi perché altrimenti mi servirebbe troppo spazio per memorizzare i dati. Quindi devo scegliere un intervallo capace di approssimare bene la funzione.

Il problema è che memorizzare un istogramma per la correlazione richiede molto spazio perché se abbiamo 20 intervalli per ogni attributo, alla fine avremo una tabella con 400 valori.

Istogrammi di questo tipo sono usati pochissimo.

L'aggiornamento dell'istogramma in generale non viene fatto ogni volta che abbiamo una singola modifica all'interno del database.

Si fa una operazione di aggiornamento o a mano o ogni tot tempo perché farlo ogni volta sarebbe un'operazione troppo costosa.

Quindi queste statistiche sono utili quando tra un aggiornamento all'altro non abbiamo modifiche troppo importanti che possano modificare la distribuzione dei dati.

## Physical Operator

Per l'implementazione delle operazioni relazionali assumiamo di avere a disposizione degli indici che possono rendere l'esecuzione delle operazioni più efficienti.

Distinguiamo in particolare due tipi di indici, quelli clusterizzati in cui abbiamo un ordinamento sulla chiave e quelli non clusterizzati in cui invece non abbiamo un ordine. In particolare quelli non clusterizzati possono comportare un accesso ad un numero di pagine che è maggiore del numero effettivo di pagine in cui sono memorizzate le foglie perché dato che i dati non sono ordinati, una pagina potrebbe essere visitata più di una volta.

Data una query, l'idea è di eseguire delle ottimizzazioni che possano portarci ad avere una esecuzione ottima.

La query viene rappresentata in un albero in cui i vari nodi sono operatori dell'algebra e l'obiettivo è modificare questo albero per ottimizzarlo.

Per stimare il costo C dell'operatore fisico, si considera il numero delle pagine che vengono lette o scritte nella memoria permanente.

Ad esempio se abbiamo un accesso con indice si utilizza la classica formula  $C_s = C_i + C_d$ , se facciamo una semplice scansione del file con i record consideriamo come costo il numero delle pagine.

Nell'algebra ognuno degli operatori che troviamo all'interno degli alberi ha una arietà ovvero per ogni operatore sappiamo il numero di input.

## Operazioni per Tabelle e Sort

Consideriamo i vari operatori per accedere alle tabelle:

- TableScan(R): vengono restituiti tutti i record che si trovano in R, l'arietà è 0 perchè non ha valori in input ma prende i dati direttamente dal disco. Il costo in questo caso è pari al numero totale delle pagine che vengono lette, quindi  $C=N_{page}$
- SortScan(R,Ai): abbiamo la tabella R e un attributo Ai, l'obiettivo è ordinare i dati della tabella in ordine decrescente considerando l'attributo Ai. L'ordinamento viene effettuato con l'algoritmo Merge Sort. Il costo dipende dal numero di pagine che sono occupate dalla tabella. Se abbiamo un numero di pagine che non supera  $B^2$  dove B è la dimensione della memoria principale, allora il costo è  $3*N_{page}(R)$ . Per l'ordinamento assumiamo di utilizzare il Merge Sort.
- IndexScan(R,Idx): dato R e l'indice I, IndexScan restituisce i record di R ordinati in modo crescente rispetto all'attributo Ai dell'indice. Il costo dipende dal tipo di indice, se è un indice clustered in cui però abbiamo un ordinamento delle chiavi su un attributo I dobbiamo comunque accedere alle foglie e a tutti i record quindi avremmo un costo che è  $C=N_{leaf}+N_{rec}$ . Quindi in pratica IndexScan non viene mai utilizzato

- IndexSequentialScan(R,I): I record di R sono memorizzati con una organizzazione primaria con indice sequenziale. IndexSequentialScan restituisce i record ordinati in base alla chiave primaria andando a scorrere solamente le foglie. Quindi il costo è  $C = N_{Leaf}$ .

In tutti questi casi la cardinalità del risultato è pari a  $E_{rec} = N_{rec}(R)$  perchè restituisco tutti i record di R.

## Operazioni per Proiezioni

Operatore che fa la proiezione senza però fare una eliminazione dei duplicati.

- Project(O,Ai): L'operatore di proiezione ha una arietà 1 perchè prende l'input da un altro operatore O, non aggiunge niente al costo perchè l'operazione viene eseguita in memoria principale e non elimina i duplicati.
- IndexOnlyScan(R,Idx,Ai): se ci troviamo nella situazione in cui abbiamo un indice sull'attributo Ai che vogliamo nella proiezione possiamo utilizzare l'indexOnlyScan. In questo caso non prendiamo l'input da un altro operatore ma andiamo direttamente sul disco a prendere l'indice e facciamo la scansione delle foglie. La scansione delle foglie costa  $C = N_{Leaf}$  e per ogni record che trovo restituisco l'attributo Ai che volevo e che trovo direttamente all'interno dell'indice senza dover accedere ai record.

Assumiamo di fare una scansione dell'indice sui family name, per ogni family name abbiamo vari RID. Questo indexOnlyScan può essere implementato in vari modi, se ad esempio ho  $\text{familyName} \rightarrow 4 \text{ rid}$ , posso restituire o 4 family name uguali oppure solamente un family name. Nel primo caso siamo nella situazione "Multiset Version", il secondo caso è la situazione "Set Version". Si usa praticamente sempre l'implementazione con MultiSet Version perchè si fa questa considerazione: se ho la multiset

version e quindi restituiscono più volte lo stesso family name ma voglio fare la set version, mi basta mettere nella pipeline l'operatore distinct per ottenere la set version (il distinct lo faccio in main memory quindi il costo è 0).

Se invece ho solamente l'implementazione della set version e devo fare una query in cui mi serve la multi set version non posso perchè ho perso alcune informazioni che avrei dovuto utilizzare. L'idea è di evitare l'implementazione di operatori che fanno due cose alla volta, nel multiset query facciamo solamente una operazione e poi il distinct è un altro operatore. Non faccio un operatore che fa entrambe le cose.

## Operazioni per cancellazione dei duplicati

- Distinct(O): operatore che ha arietà 1 perchè prende l'input da un altro operatore. Prende in input una serie di record ordinati in base ad una qualsiasi chiave poi le chiavi che compaiono più di una volta vengono eliminate.

Distinct prende un record e lo memorizza, poi chiede il prossimo all'operatore sotto e confronta, se sono uguali elimina i duplicati consecutivi.

L'importante è che l'input sia ordinato, dato che non ho un operatore che fa sia distinct che sort insieme allora ho la necessità di utilizzare un secondo operatore per l'ordinamento mettendolo in una pipeline con distinct. In questo modo possiamo ottimizzare la fase del sorting ottenendo un costo che è  $C = 2Npage$ .

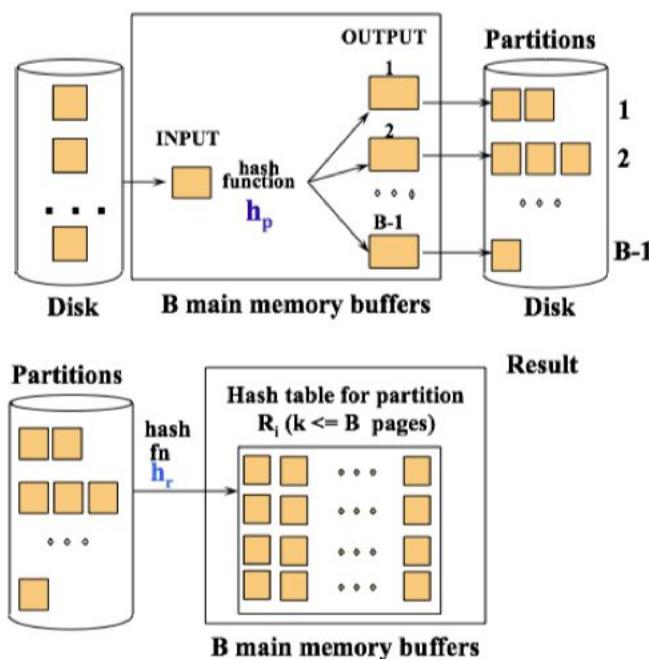
In realtà l'operatore distinct ha una richiesta meno stringente sull'input, chiede infatti che l'input sia raggruppato, non è necessario che sia ordinato. Questa richiesta è meno forte, ad esempio se abbiamo un record che ha tre campi A,B,C, posso volere un raggruppamento in base ad A,B,C ma posso ordinare sia in base a A,B,C che B,C,A ad esempio, perchè comunque sono raggruppati.

- HashDistinct(O): abbiamo in input l'output di un altro operatore. Prendiamo tutte le pagine e per ogni pagina calcoliamo la funzione hash  $h_1$  di ogni record. Ogni record viene mappato in un bucket di una hash table. Abbiamo  $B$  bucket in memoria e quando se ne riempie uno scriviamo su disco. Alla fine su disco avremo  $B$  bucket con  $B$  pagine ciascuno.

Ora prendiamo i  $B$  bucket dal disco e per ognuno prendiamo le  $B$  pagine, per ogni pagina applichiamo un hash  $h_2$  ai vari record e li mappiamo in  $B$  bucket in modo che ogni record venga mappato in un bucket differente, quando un record è mappato in un bucket dove c'è già qualcosa allora quel record viene eliminato perché è duplicato. La procedura diventa ricorsiva se ho più di  $B$  pagine in ogni bucket, in tal caso ci vogliono  $\log N_{pages}$  per riuscire a completare l'eliminazione.

Con l'hashing non facciamo un ordinamento, siamo solamente capaci a controllare se due record sono differenti tra loro.

Il costo è =  $C(O) + 2^*N_{pages}(R)$ .



Stima della dimensione dell'output:

Dobbiamo considerare due casi, il primo è quando abbiamo un solo attributo A nei record che abbiamo in input, in questo caso abbiamo che

il numero di record che vengono restituiti è pari a  $N_{key}(A)$ .

L'altro caso è quello in cui abbiamo invece più di un attributo tra quelli in input all'operatore distinct, in questo caso Erec:

$$- |Q| = \min(|O|/2, \prod_i N_{key}(A_i))$$

## Operatori per Sort

$\text{Sort}(O, A_i)$ : È l'operatore per il sorting, si tratta di un operatore che legge i dati in input che arrivano da un altro operatore e poi ordina in base all'attributo  $A_i$ . Se consideriamo che il numero delle pagine da ordinare è minore di  $B^*(B-1)$  ovvero è minore della dimensione del buffer al quadrato allora vuol dire che abbiamo un costo per l'ordinamento che è pari a  $2^*N_{page}(O)$ .

La cardinalità del risultato è pari a  $Erec = N_{rec}(O)$ .

## Operatori per Selezione

- $\text{Filter}(O, \text{condizione})$ : è l'operatore più semplice per la selezione, prende in input dei dati che arrivano da un altro operatore e poi manda in output i record che rispettano la condizione.  
Il costo di questo operatore è  $C=C(O)$ , quindi è 0 e dipende dal costo degli operatori precedenti. Il numero di record che vengono mandati in output dipende dal selectivity factor della condizione:  
 $Erec = sf(\text{condizione}) * N_{rec}(O)$
- $\text{IndexFilter}(R, \text{Idx}, \text{condizione})$ : questo operatore mi permette di filtrare i record in base alla condizione. È essenziale che l'indice sia definito sugli attributi usati nella condizione perchè per prima cosa scorro le foglie e cerco di capire quali RID rispettano la condizione e poi vado a recuperare i record utilizzando il RID.  
Il costo di questo operatore è  $C = Ci + Cd$ .  
Quindi, se abbiamo un indice che è clusterizzato abbiamo:  
 $Ci = sf * N_{leaf}$   
 $Cd = sf * N_{page}(R)$

Se l'indice è unclustered:

$$C_i = sf^* N_{leaf}$$

$$C_d = (sf^* N_{key}) * \text{Cardenas}(N_{rec}/N_{key}, N_{page})$$

Il numero di record che vengono restituiti da questo operatore di selezione è Erec = sf^\* Nrec.

- IndexSequentialFilter(R, I, condizione): I record di R sono memorizzati usando come primary organization un indice I che utilizza gli attributi della condizione. Quindi con una sola scansione delle foglie trovo i record che soddisfano la condizione.

Costo:  $sf^* N_{leaf}(I)$

- IndexOnlyFilter(R, I, {Ai}, condizione): in questo caso l'operatore fa sia selezione sia proiezione e i record sono restituiti ordinati . L'indice è con l'attributo Ai e la condizione ha solamente l'attributo Ai (o gli attributi Ai). Scansiono le foglie e trovo i record che rispettano la condizione, restituisco questi record facendo però una proiezione con i soli attributi Ai.

Costo =  $sf^* N_{leaf}$ .

Se Ai è chiave primaria allora il risultato non avrà neanche i duplicati.

- AndIndexFilter(R, {li, condizione}): in questo caso dobbiamo avere vari indici e per ognuno di questi indici dobbiamo considerare una condizione. Quindi prima usiamo i vari indici e prendiamo da ogni indice i RID che rispettano la condizione, facciamo l'intersezione e poi andiamo in memoria e prendiamo i record che abbiamo selezionato.

Il costo complessivo è  $C = C_i + C_d$ . Il  $C_i$  è la somma di tutti i costi degli accessi agli indici, il  $C_d$  invece è l'accesso ai dati, quindi dobbiamo considerare Erec che è il numero di dati che mi aspetto di estrarre Erec =  $sf^* N_{rec}$  e quindi  $C_d = \text{Cardena}(Erec, N_{page}(R))$ .

## Operatori per Grouping

Gli operatori per il grouping hanno un funzionamento che è simile agli operatori che si usano per il distinct.

Anche in questo caso è necessario avere in input una serie di record che devono essere ordinati sulla base dell'attributo  $A_i$ .

L'idea è che per ogni raggruppamento di record viene svolta una funzione e si restituisce solamente un record.

Se abbiamo l'input ordinato allora l'ordinamento implica il raggruppamento, il contrario però non vale.

- $\text{GroupBy}(O, \{A_i\}, \{f\})$ : arrivano in input i dati da un operatore precedente, questi dati sono ordinati in base all'attributo  $A_i$ . Per ognuno dei gruppi viene applicata la funzione  $f$  e si restituisce un record.

Il costo è  $C = C(O)$ .

Vengono restituiti record ordinati.

Il numero di record che vengono restituiti è  $N_{key}(A_i)$  se abbiamo solamente un attributo invece è, come nel distinct,

$$- |Q| = \min(|O|/2, \prod_i N_{key}(A_i))$$

- $\text{HashGroupBy}(O, \{A_i\}, \{f\})$ : il funzionamento di questo operatore è simile a quello con l'hash che abbiamo nel caso del Distinct. In output abbiamo dei record che non sono ordinati, abbiamo comunque eseguito la funzione  $f$  e quindi riduciamo il numero di record, il calcolo è lo stesso del caso precedente.

Tra i due operatori se non abbiamo i dati ordinati il costo è lo stesso.

L'hash può essere ottimizzato meglio quando abbiamo una dimensione dell'input che è leggermente più grande del buffer, non quando è troppo più grande.

## Operatori per la Join

Quando facciamo la Join abbiamo una tabella di record esterna e una tabella “interna”, poi abbiamo una condizione che ci permette di capire se il record della tabella interna va unito con quello della tabella esterna.

- NestedLoop(O<sub>e</sub>, O<sub>i</sub>, condizione): questo algoritmo non viene mai usato ma è il padre degli altri algoritmi per la join.

Abbiamo:

```
foreach r in OE do
    foreach s in Oi do
        if r.r1 = s.s1 then add <r, s> to result
```

Ovvero ci troviamo un algoritmo quadratico che prende ogni record della tabella esterna e confronta con i record della tabella interna, se la condizione è rispettata inserisce nella soluzione il risultato.

Il costo di questo operatore:

$$C = C(O_e) + Erec(O_e) * C(O_i)$$

È quadratico e il numero di operazioni che vengono svolte dipendono anche da qualche tabella usiamo come esterna e quale come interna.

Il numero di record che troviamo nella soluzione dipende dal selectivity factor della condizione e dal numero di record delle due tabelle:

$$Erec = Sf(Condizione) * Erec(O_e) * Erec(O_i).$$

- PageNestedLoop(O<sub>e</sub>, O<sub>i</sub>, condizione): Il costo del Nested Loop lo possiamo ridurre se invece di considerare la tabella O<sub>e</sub> record per record la consideriamo pagina per pagina. Prendiamo una pagina di R e la portiamo in memoria, poi prendiamo una pagina di S e la portiamo in memoria. Poi per ogni record di R facciamo un confronto tra quel record e i record della pagina di S portata in memoria.

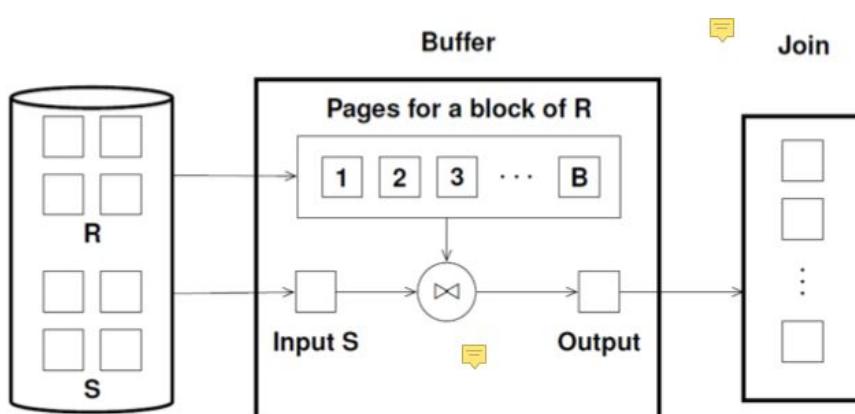
In questo modo riusciamo ad abbassare il numero delle pagine che vengono lette e lo portiamo a  $C = Npag(R) + NPag(R)*NPag(S)$ .

- BlockNestedLoop(Oe, Oi, condizione):

Si tratta di un operatore simile al precedente, in questo caso però consideriamo un buffer che ha dimensione  $B+2$ , 1 pagina viene usata per memorizzare una per volta le pagine di S, poi B pagine sono usate per memorizzare le pagine di R e una pagina è per l'output.

Quindi prendiamo una pagina da S, con l'hash confrontiamo i record di questa pagina con i record nelle B pagine e se la condizione viene rispettata scriviamo nella pagina di output il record.

Il costo in questo caso è:  $C = \text{Npag}(R) + (\text{Npag}(R)/B) * \text{Npag}(S)$



- IndexNestedLoop(Oe, Oi, condizione): in questo caso creiamo un indice su una delle due tavole utilizzando la colonna che poi uso per fare la join.

L'index Filter è probabilmente l'algoritmo più importante per la join perché ogni DBMS costruisce un indice sulla chiave primaria e sulla foreign key perchè servono per controllare i vincoli.

La regola generale è che questo metodo non viene usato quando il numero dei record di entrambi le relazioni è molto alto perchè il costo sarebbe troppo alto. Quando invece abbiamo uno dei due che ha un numero basso di record allora è più interessante perchè il costo si abbassa e possiamo usare l'indexNestedLoop.

Quindi ho un indice, poi prendo ogni record nell'altra tabella e calcolo l'indexFilter usando l'indice e la condizione.

Se la condizione è rispettata aggiungo la coppia dei record

all'interno del risultato.

```
foreach r in R do
    foreach s in IndexFilter(S, l, s.s1=r.r1) do
        add <r, s> to result \
```

Il costo è  $C = C(Oe) + E_{rec}(Oe) * (C_i + C_d)$ .

È importante sapere che l'index Nested Loop è formato da due parti, indexNestedLoop e IndexFilter. In particolare l'index Filter va sempre nella parte a destra mentre a sinistra ci può andare qualsiasi altra cosa di una complessità arbitraria. Nella parte destra ci va solamente l'index filter e al più un filter sopra all'index filter. Il motivo è che il filter è un operatore trasparente che fa passare i dati che arrivano da sopra e li fa arrivare all'index Filter. Nell'index filter arriva una certa costante e viene fatto il confronto tra questa costante e l'attributo.

- Merge Join(Oe, Oi, condizione): abbiamo le tabelle R e S che devono essere già ordinate, la tabella esterna Oe è ordinata in base alla sua chiave primaria.  
Facciamo il merge della tabella esterna con la tabella interna, è importante avere la Oe ordinata rispetto alla chiave primaria e usare questa chiave primaria come condizione di join perchè in questo modo per ogni record di Oe facciamo più confronti con i record di Oi. Se avessimo invece varie chiavi differenti dovremmo partire dalla prima in Oe e scorrere Oi, poi passare alla successiva in Oe e ripartire anche dalla prima in Oi.  
Se assumiamo che ognuna delle due tabelle è ordinata allora possiamo dire che l'operatore ha un costo pari a 0, quello che costa è l'ordinamento.  
Costo:  $C = C(Oe) + C(Oi)$ .  
È importante che la condizione della join sia una equality, con l'inequality in pratica non conviene usare questo metodo.  
Nel caso del merge join abbiamo un principio importante che deve essere rispettato:  
Se abbiamo un grande parallelismo e vogliamo fare la join tra R ed

S allora abbiamo 2 possibili approcci:

1) Possiamo sfruttare la proprietà distributiva della join, prendi R ed S e li dividi in modo random, poi prendi ogni blocco che hai creato in R e prendi il primo di S e li mandi in una CPU, poi prendi il primo di R e il secondo di S e vai in un'altra CPU e così via. In questo modo abbiamo che devi calcolare la join tra tutte le coppie di blocchi che hai. È una soluzione molto simile al Nested Loop.

2) La seconda soluzione invece mi splitta R in base ad un certo criterio sull'attributo della join, ordino R in base ad A e poi divido in blocchi in base al valore di A, (tipo A compreso tra 0 e 10). Poi faccio la stessa cosa anche con B e la join la faccio solamente tra i blocchi che so che mi daranno un risultato. Quindi non considero tutte le coppie.

In questo caso ho un costo maggiore all'inizio per la divisione ma poi la join mi costa di meno.

- HashJoin(Oe, Oi, condizione): possiamo eseguire la join utilizzando l'hash in due passaggi.

Il primo passaggio è il partizionamento di R ed S, abbiamo una funzione h1 e viene applicata la funzione h1 a tutti i record di R e di S creando quindi una serie di bucket ognuno con una serie di pagine all'interno.

La funzione che dobbiamo scegliere dipende dalla dimensione del buffer perchè vogliamo fare tutto in streaming nel senso che leggiamo il record, facciamo l'hash in memoria e poi dopo lo scriviamo sul disco dove il file che creiamo potrà essere di dimensione arbitraria. Se abbiamo 100 pagine nel buffer allora faremo il mapping dei record in 100 bucket.

Dopo la fase di Partitioning c'è la fase di Matching, prendiamo ogni bucket di R e usiamo una hash table con una seconda hash function differente dalla prima. Quindi, per ogni record nel bucket che stiamo considerando applichiamo la funzione h2 e inseriamo in una hash table. Poi portiamo in memoria ogni pagina di S e per ogni record calcoliamo l'hash e andiamo a vedere se combacia con uno di quelli di R, in quel caso allora mettiamo in output quel record.

Nella fase di partizionamento abbiamo il seguente costo per leggere e scrivere:  $C = C(Oe) + C(Oi) + Npag(Oe) + Npag(Oi)$ .

Nella fase di Matching invece abbiamo che ogni partizione viene letta una volta quindi  $C = Npag(Oe) + Npag(Oi)$ .

Assumiamo che  $Npag(R)/B < B$ :

In tutto il costo è  $C = C(Oe) + C(Oi) + 2(Npag(Oe) + Npag(Oi))$ .

Il costo è lo stesso del merge sort però c'è una differenza perchè quando usiamo il Merge Join abbiamo bisogno di un ordinamento delle due tabella su cui facciamo la join. Quindi se abbiamo tabelle con molti record ci mettiamo di più a fare l'ordinamento rispetto invece a fare l'Hash Join in cui comunque abbiamo sempre due fasi. La situazione migliore comunque ce l'abbiamo quando una delle due tabelle entra in memoria principale.

Consideriamo un caso non bilanciato delle dimensioni delle pagine delle due tabelle su cui vogliamo fare la join. Prendiamo per esempio studenti, una tabella con 10k pagine che entra in memoria e poi esami che invece ha un milione di pagine e non entra in memoria.

Possiamo comunque usare solamente due passaggi per eseguire la join?

Consideriamo la fase di partitioning. Creiamo B bucket per entrambe le tabelle e la differenza è che per esami abbiamo un file su disco più grande e ogni bucket avrà molti più pagine.

Poi nella seconda fase, possiamo subito portare in memoria tutte le pagine di studenti mentre ne portiamo una alla volta di esami e quindi non abbiamo problemi.

La cosa fondamentale è che in memoria principale dobbiamo portare sempre la tabella che ha meno pagine mentre l'altra la usiamo per fare i confronti portando però una pagina alla volta. Il costo totale dipende da entrambi alla fine perchè devo leggere sia la relazione più piccola che quella più grande

Ci possono essere situazioni in cui dobbiamo fare delle join complesse, in questo caso potremmo avere ad esempio una congiunzione o una disgiunzione.

In questi casi si preferisce fare prima la join sulla condizione più semplice e poi fare una filter sulla seconda condizione.

## Set Operators

- Union, Except e Intersect: in questo caso ci serve avere le tabelle su cui usiamo questi operatore ordinate. Di questi tre operatori esiste anche la versione che funziona con l'hashing, questa soluzione è simile all'Hash Join approach. Questi operatori assumono che tu abbia già fatto il distinct e quindi che hai già eliminato i duplicati.
- Union All Operator: non elimina i duplicati, scansiona il primo input e poi il secondo input e poi unisce tutti

## Lezione 8: Query Optimizations

Ci stanno varie alternative per eseguire una query e l'obiettivo, in generale, è trovare una ottimizzazione che migliori le prestazioni della query che è stata scritta inizialmente.

Le ottimizzazioni di una query possono essere dinamiche o statiche, quelle dinamiche vengono generate a runtime quando la query viene eseguita, quelle statiche invece sono generate nel momento in cui viene compilato il programma.

Abbiamo 4 fasi per l'ottimizzazione delle query:

- Query Analysis: viene generato il query plan logico basato sull'algebra relazionale. Si esegue un'analisi statica della query, controlliamo la sintassi e l'esistenza delle tabelle, controlliamo se i numeri sono interi e poi vengono riscritte le condizioni booleane in modo da produrre un albero logico della query.
- Query Transformation: prima trasformazione della query in modo da ottenere delle performance migliori. In generale vengono spostate le restrizioni più in basso nella query.
- Physical Plan generation: generazione di un access plan fisico. Si cerca di generare l'access plan fisico migliore, non può essere perfetto perchè non abbiamo informazioni sulla correlazione degli attributi. Per avere proprio l'ottimo dovrei generarli tutti quanti e poi andare a vedere il migliore, questo ha un costo decisamente troppo alto quindi si usano delle heuristiche per generare gli access plan migliori.  
Se anche non troviamo il migliore, almeno evitiamo i peggiori.
- Query evaluation: valutazione dell'access plan fisico

Ci sono delle trasformazioni che possiamo fare sulla logica della query prima di andare a fare l'ottimizzazione, in particolare possiamo eliminare il distinct quando questo non è necessario ed è ridondante, possiamo togliere il groupby e le subquery usate con il where, in alcuni casi è anche possibile eliminare le view.

La maggior parte di queste ottimizzazioni sono basate sulle dipendenze funzionali.

### Dipendenze funzionali:

Dipendenza funzionale: Data una tabella con gli attributi X e Y, X determina Y (quindi abbiamo una dipendenza funzionale) quando prese due istanze di R abbiamo che le due istanze sono uguali sia sull'attributo X che sull'attributo Y:

$X \rightarrow Y$  ( $X$  determines  $Y$ ) iff:

- $\forall r$  valid instance of R.  
 $\forall t1, t2 \in r. \text{ If } t1[X] = t2[X] \text{ then } t1[Y] = t2[Y]$

Consideriamo alcune dipendenze funzionali:

StudCode	Name	City	Region	BirthYear	Subject	Grade	Univ
1234567	Mary	Pisa	Tuscany	1995	DB	30	Pisa
1234567	Mary	Pisa	Tuscany	1995	SE	28	Pisa
1234568	John	Lucca	Tuscany	1994	DB	30	Pisa
1234568	John	Lucca	Tuscany	1994	SE	28	Pisa

- $\text{StudCode} \rightarrow \text{Name, City, Region, BirthYear}$  
- $\text{City} \rightarrow \text{Region}$
- $\text{StudCode, Subject} \rightarrow \text{Grade}$  
- $\emptyset \rightarrow \text{Univ}$  
- $\text{StudCode, Name} \rightarrow \text{City, Univ, Name}$  

Ci troviamo ad avere:

- La prima dipendenza è dovuta al fatto che la tabella è una join tra esami e studenti, quindi abbiamo che quando troviamo un certo student code alcuni dei dati saranno sempre uguali, ovviamente cambia il nome dell'esame e il voto.  
Notare che per la prima dipendenza vale che StudCode determina le altre ma questo mi dice anche che Stud Code determina gli altri attributi presi uno per uno.

- La seconda dipendenza è dovuta al fatto che la città determina anche la regione di appartenenza
- Dato che uno studente viene esaminato solamente una volta in ogni materia allora vuol dire che la coppia studentCode, Subject determina anche il voto dello studente
- Dato che il nome dell'università è sempre lo stesso allora possiamo scrivere che l'insieme vuoto determina l'università
- L'ultima è una dipendenza derivata, la regola mi dice che se  $X \rightarrow Y$  allora posso anche scrivere che  $X, Y \rightarrow Y$ .  
Quindi in questo caso diciamo che student code preso insieme a name determina name, determina City e Univ. Univ lo scrivo perchè l'insieme vuoto lo determina, Region non lo scrivo perchè sappiamo che City  $\rightarrow$  Region.

**Dipendenza Canonica:** Una dipendenza si dice canonica quando la parte sinistra della dipendenza è minimale ovvero non possiamo eliminare nessuno degli attributi perchè altrimenti non abbiamo più la dipendenza. Ad esempio StudentCode, Subject  $\rightarrow$  Grades è canonical perchè se elimino uno dei due non ho più la dipendenza. Invece se prendiamo StudentCode, Name  $\rightarrow$  Univ, Name,City, questa non è canonica perchè StudentCode da solo mi determina già gli altri.  
In una tabella di persone, fiscal code determina tutti gli altri campi e quindi è una chiave però la coppia fiscal code, family name anche se determina tutti i campi non è canonica perché il family name è ridondante.

Le dipendenze non trivial possono essere trasformate e possiamo farle diventare canoniche.

In un database progettato bene le uniche dipendenze sono dipendenze che hanno nella parte sinistra la chiave della tabella.

Le dipendenze possono anche derivare partendo da altre dipendenze utilizzando l'Armstrong Rule ovvero usando riflessività, Augmentation e Transitivity.

**Implicazione Logica:** dato un set F di dipendenze funzionali e uno schema relazionale R, una dipendenza  $X \rightarrow Y$  è logicamente implicata da F se ogni istanza che soddisfa F soddisfa anche  $X \rightarrow Y$ .

Si scrive:

$$F \vdash X \rightarrow Y$$

**Closure:** quando ci troviamo ad avere un attributo e vogliamo l'elenco degli attributi che sono determinati da quell'attributo allora creiamo una closure. La closure è l'insieme degli attributi che sono determinati da un attributo.

Partiamo da un set iniziale e poi aggiungiamo gli attributi che sono determinati fino ad arrivare al punto in cui non ho da aggiungere altro. Per sapere se un certo attributo è determinato da un altro cerco nella closure.

Dato uno schema relazionale  $R < T, F >$  la closure di X che chiamiamo  $X^+$  è definita come:

$$X^+ = \{A_i \text{ in } T \mid F \vdash X \rightarrow A_i\}$$

Consideriamo una query in cui abbiamo anche una join, cosa possiamo dire delle dipendenze?

Se ho delle dipendenze che valgono prima della join allora queste dipendenze funzionali varranno anche dopo la join, l'unica differenza è che non sono più chiavi primarie come erano prima perchè avendo fatto la join, quella chiave non è più primaria nella tabella complessiva.

Se nella query abbiamo una selection con una costante, ad esempio  $Year = 2000$  allora vuol dire che abbiamo una dipendenza nuova, empty set infatti mi determinerà Year nel risultato della query. Questa diventa una dipendenza costante.

Se nella query abbiamo una condizione join tipo  $Course.idCourse = student.idCourse$  allora vuol dire che nel risultato l'idCourse dello studente  $\rightarrow$  idCourse del course e viceversa.

Quindi, la join aggiunge anche delle dipendenze funzionali e queste possono essere interessanti perchè alcune volte possono essere delle chiavi primarie nella tabella e partendo da una condizione transitiva posso scrivere altre dipendenze.

Assumiamo di avere una query e che vogliamo sapere se un set di attributi sono chiavi per il risultato e se determinano altri set di attributi.  
Una query del tipo:

```
Select familyname, City
From Student join Courses
```

Vogliamo sapere se la coppia familyName, City determina altri attributi.  
Dobbiamo calcolare la closure di familyName, City.

La closure all'inizio contiene solamente familyName, City, se nella query abbiamo delle condizioni come Year = 2000 allora anche Year diventa parte della closure perchè è una costante ed è determinato dall'empty set.

Se poi abbiamo una join con una condizione del tipo  $A_j = A_i$  o  $A_j = A_k$  e abbiamo che  $A_k$  è nella closure allora anche  $A_i$  è nella closure perchè  $A_k \rightarrow A_j \rightarrow A_i$ .

Se la closure contiene ora lo studentID, dato che questa è una chiave della prima tabella metto nella closure tutti gli attributi di studenti.

Algoritmo per creare una closure:

- Abbiamo un attributo iniziale X (o più attributi iniziali) e creiamo la closure  $\{X\}^+$  che li contiene tutti.
- Prendiamo la condizione del where e consideriamo quelle condizioni tali che  $A_i = c$  dove c è una costante,  $A_i$  viene inserito all'interno della closure.
- Consideriamo gli attributi  $A_j$  del where tali che  $A_j = A_k$  se k appartiene già alla closure.
- Aggiungiamo tutti gli attributi di una tabella se nella closure è presente la chiave primaria di quella tabella.

## Eliminazione del Distinct

Spesso si mette il distinct anche quando non serve e in questo caso abbiamo un operatore ridondante che, quando deve essere utilizzato, richiede anche l'utilizzo del sort. Quindi se sono ridondanti riusciamo a ridurre il numero di operazioni da fare eliminando il distinct.

Per capire se dobbiamo eliminare il distinct dobbiamo creare la closure partendo dagli attributi che troviamo nella Projection. Partendo da quelli poi aggiungiamo tutti gli altri che riusciamo a generare e poi alla fine controlliamo se tutti gli attributi delle tabelle su cui abbiamo fatto la join sono nella closure oppure no. Se non ci sono il distinct serve, altrimenti non serve.

```
Products(PkProduct, ProductName, UnitPrice)
Invoices(PkInvoiceNo, Customer, Date, TotalPrice)
InvoiceLines(FkInvoiceNo, LineNo, FkProduct, Qty, Price)
```

and the query

```
SELECT DISTINCT FkInvoiceNo,TotalPrice
FROM InvoiceLines, Invoices
WHERE FkInvoiceNo = PkInvoiceNo;
```

To check whether **DISTINCT** is unnecessary let us compute the closure of the projection attributes.

$$\begin{aligned}\mathcal{A}^+ &= \{FkInvoiceNo, TotalPrice\} \\ &= \{FkInvoiceNo, TotalPrice, PkInvoiceNo, Customer, Date\}\end{aligned}$$

```
SELECT DISTINCT FkInvoiceNo,TotalPrice
FROM InvoiceLines, Invoices
WHERE FkInvoiceNo = PkInvoiceNo AND LineNo = 1;
```

$$\begin{aligned}\mathcal{A}^+ &= \{FkInvoiceNo, TotalPrice\} \\ &= \{FkInvoiceNo, TotalPrice, LineNo\} \\ &= \{FkInvoiceNo, TotalPrice, LineNo, PkInvoiceNo, Customer, Date, \\ &\quad FkProduct, Qty, Price\}\end{aligned}$$

**DISTINCT** becomes unnecessary because  $\mathcal{A}^+$  contains the primary keys of all tables.

Se vogliamo utilizzare il distinct quando utilizziamo anche GroupBy dobbiamo fare una cosa diversa, partiamo sempre con la closure con gli attributi della proiezione, senza le funzioni del group by. Poi espandiamo e controlliamo se nella closure troviamo gli attributi della group by o no. Se ci sono entrambi allora il distinct non serve, altrimenti serve.

## Eliminazione del Group By

Il groupBy non è più necessario se ogni gruppo consiste di un solo record oppure se abbiamo un solo gruppo.

In questo caso consideriamo la query e creiamo la closure, se abbiamo che tutte le chiavi primarie delle tabelle su cui viene fatta la join sono nella closure allora vuol dire che avremo gruppi di 1 solo elemento e quindi possiamo eliminare il groupBy.

Nella scrittura della nuova query modifichiamo anche l'operazione relativa al groupBy che abbiamo all'interno del select, ad esempio il count viene sostituito con 1.

Se invece, data la query, la closure che creiamo contiene tutti gli attributi che stanno nel groupBy allora possiamo eliminare il groupBy perchè abbiamo un gruppo unico di record.

## Eliminazione del Where nella subQuery

Per quanto riguarda l'eliminazione del Where svolto in una subquery, il problema è più complicato e non esiste un algoritmo preciso che può essere utilizzato in ogni situazione.

Il problema in questo caso è che abbiamo una clausola Where esterna e poi per ogni attributo abbiamo EXIST ed una clausola interna. In pratica per ogni clausola esterna facciamo un nested loop ed eseguiamo la query interna un numero di volte pari al numero di record esterni.

Quindi l'idea è quella di eliminare il where della subQuery cercando di trasformarlo in un join.

```
select *           ← nested correlated
from studenti s ←
where exists (select * from exams e where e.sid=s.sid)
```

Il caso più semplice è quando abbiamo solamente EXISTS e non abbiamo un GroupBy nella sotto query.

#### Tipi di Join:

- Natural Join: è la join classica ovvero abbiamo uniamo due tabelle a patto che PrimaryKeyA = ForeignKeyB
- Left Join (Right Join): in questo caso oltre ad unire le due tabelle come nel primo caso, consideriamo anche i record della tabella di sinistra(destra) che non compaiono in quella di destra(sinistra)
- Natural Full Join: Uniamo tutti i record in una tabella senza ripetizioni

#### Esempi:

```

SELECT *
FROM Courses C
WHERE CrsYear = 2012 AND 
EXISTS (SELECT FROM Transcripts T
    WHERE T.CrsName = C.CrsName
    AND T.Year = CrsYear);

- The unnested equivalent query is

SELECT DISTINCT C.*
FROM Courses C, Transcripts T
WHERE T.CrsName = C.CrsName AND T.Year = CrsYear
AND CrsYear = 2012;

```

In questo caso per rimuovere il where con la subquery facciamo solamente una join con gli attributi che troviamo nel where interno. È importante inserire il distinct perchè in questo caso abbiamo per ogni corso più transcript possibili e noi vogliamo solamente i dati relativi al corso.

- **SELECT C.CrsName, C.Teacher**  
**FROM Courses C**  
**WHERE CrsYear = 2012 AND**   
**EXISTS ( SELECT count(\*) FROM Transcripts T**  
**WHERE T.CrsName = C.CrsName AND T.Year = CrsYear**  
**HAVING 27 < AVG(Grade))**
- The unnested equivalent query is
- **SELECT C.CrsName, C.Teacher**  
**FROM Courses C, Transcripts T**  
**WHERE T.CrsName = C.CrsName AND T.Year = CrsYear**   
**AND CrsYear = 2012**  
**GROUP BY C.CrsName, C.Teacher**  
**HAVING 27 < AVG(Grade);**

Altro esempio, in questo caso nella prima query prendiamo solamente i corsi che hanno una media maggiore di 27.

La prima query viene modificata e otteniamo la seconda in cui eliminiamo l'Exists andando ad inserire una join e raggruppando in base agli attributi della select. Poi per filtrare i record si usa l'having. Il groupBy nel where interno non c'è perchè sono già raggruppati in base al nome del corso.

La stessa query di sopra è sbagliata se invece di minore nell'having usiamo uguale e questo è chiamato “Count Bug problem”.

In questo caso vogliamo i corsi che non hanno transcript:

- **SELECT C.CrsName, C.Teacher  
FROM Courses C  
WHERE C.CrsYear = 2012 AND  
EXISTS ( SELECT count(\*) FROM Transcripts T  
WHERE T.CrsName = C.CrsName AND T.Year = CrsYear  
HAVING 0 = Count(\*))**
- The following is wrong (the *count bug* problem)
- **SELECT C.CrsName, C.Teacher  
FROM Courses C, Transcripts T  
WHERE T.CrsName = C.CrsName AND T.Year = CrsYear  
AND CrsYear = 2012  
GROUP BY C.CrsName, C.Teacher  
HAVING 0 = Count(\*);**

Nella prima query abbiamo che per ogni corso contiamo i transcript e se è 0 allora va bene e restituiamo quel corso. Quando abbiamo però Having nella seconda query, in questo caso il conteggio sul gruppo completo non darà mai 0 e quindi questo è un problema e non possiamo riscrivere la query in questo modo.

Possiamo risolvere il problema con una Left Join e poi nell'having contare le righe in cui manca l'attributo relativo ai transcript.

## View Merging

In SQL è possibile creare delle viste ovvero possiamo partire da una query e trasformarla in una tabella “virtuale” per poi utilizzarla in un qualsiasi momento successivo per fare una query.

Questa vista può rimanere memorizzata oppure la possiamo creare una temporanea direttamente all'interno della query utilizzando “WITH” e dando un nome alla vista temporanea.

L'eliminazione di una View che viene creata con un groupBy richiede lo spostamento del groupBy nel logical plan della query stessa. In pratica mettiamo la query sopra alla view e poi dobbiamo trovare il modo di spostare il groupBy dalla view alla query.

Quando spostiamo il groupBy dobbiamo considerare che l'operatore della groupby ora non considera più l'attributo che già stava nella groupBy ma considera anche gli attributi della select esterna.

Il problema quindi è avere la join sopra e il groupBy sotto, abbiamo una situazione in cui questo problema può essere risolto.

L'unica situazione in cui possiamo portare la join sotto alla groupBy è quando la join è sulla condizione  $Fk(R) = Pk(S)$ .

Consideriamo il counting, con questa forma di join, viene fatta la join tra una riga di R e una sola riga di S, perchè facciamo la join tra una primary key e una foreign key, questa join quindi non è espansiva ma è flat, moltiplichiamo R per 1.

Ad esempio se abbiamo la tabella Esami e studenti, ogni esame ha solamente uno studente quindi se prima della join abbiamo 1000 esami dopo la join avremo 1000 esami e per ogni riga avremo il dato dello studente.

Quindi vuol dire che le operazioni del tipo Count, Sum, Avg, daranno tutte lo stesso risultato.

Questa è la sola situazione in cui possiamo posizionare la join sotto al groupBy.

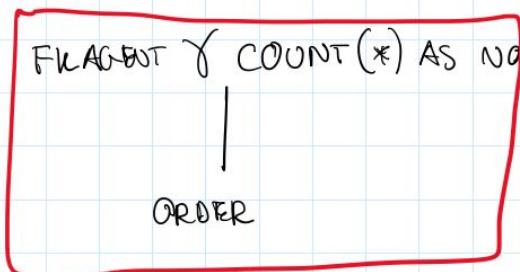
Esempio:

```
CREATE VIEW FKAgent_GBY AS
SELECT FKAgent, COUNT(*) AS No
FROM Order
GROUP BY FKAgent;
```

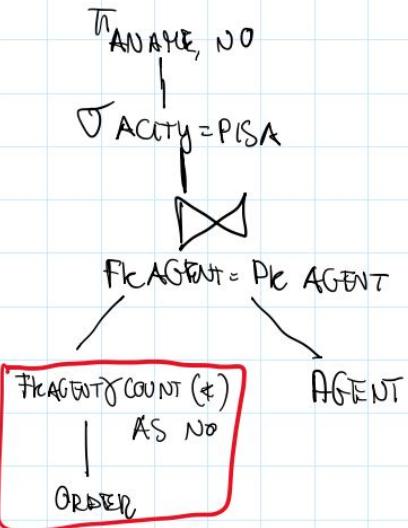
```
SELECT
FROM
WHERE
```

```
AName, No
FKAgent_GBY, Agent
FKAgent = PKAgent
AND ACity = 'Pisa';
```

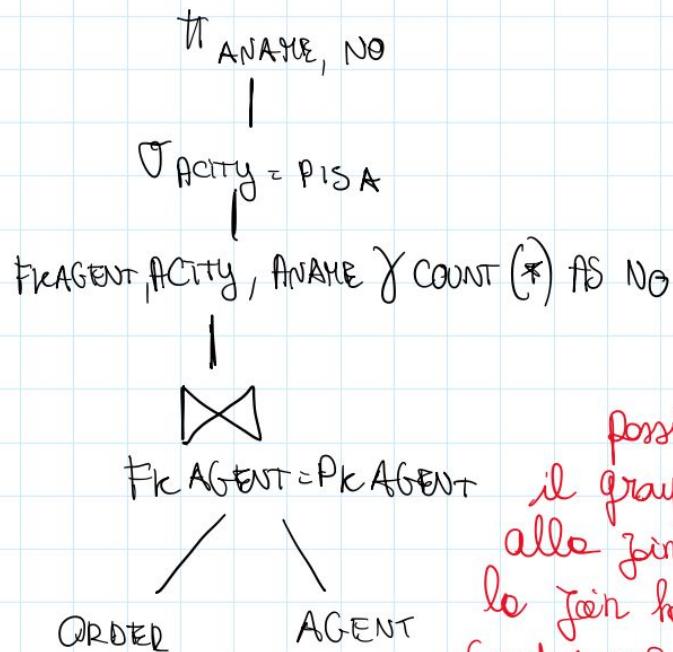
Scriviamo il logic plan della view:



logic plan query



Ora spostiamo il groupby sopra al join:



possiamo spostare  
il groupby sopra  
alle join perché  
la join ha come  
condizione  
 $FKAgent = PKAgent$

## Ottimizzazione di query con grouping e aggregations

Se abbiamo una query in cui abbiamo all'interno una groupBy e poi all'esterno una selezione, in alcuni casi è possibile spostare la selezione all'interno della groupBy e quindi eseguire il sorting su un numero minore di record.

$$\sigma_{\phi}(x \gamma_F(E)) \stackrel{?}{=} x \gamma_F(\sigma_{\phi}(E))$$

Ci sono due casi:

- Dopo la groupBy avremo solamente gli attributi della groupBy e quelli che ho aggiunto, su questi vogliamo fare la selezione.  
Se gli attributi su cui faccio la selezione sono gli stessi che usa poi il groupBy o comunque non sono gli attributi che aggiunge il groupBy allora posso spostare la selezione all'interno e fare il groupBy dopo la selezione. Se voglio portare all'interno la restrizione lo posso sempre fare, si tratta di una ottimizzazione logica.
- Quando invece abbiamo una select con un attributo che però è creato dalla groupBy: non possiamo mettere l'having come un where perchè se prima non faccio la groupBy non ho l'attributo su cui voglio fare la selezione. Quindi questa è una condizione in cui non posso spostare la groupBy.

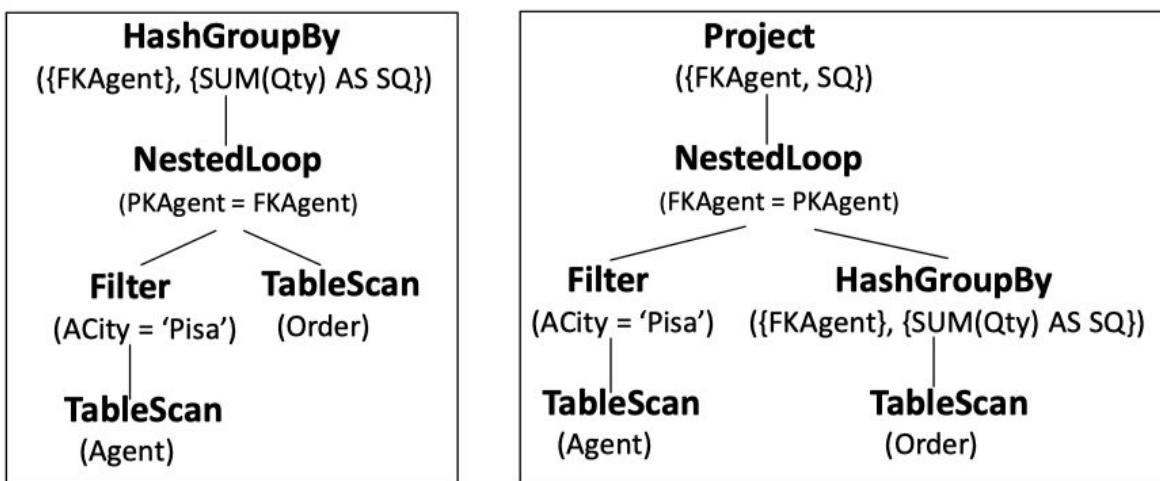
Ci sono però delle condizioni in cui posso spostare la select prima del groupBy, un esempio è il caso in cui la select mi seleziona solamente i record in cui un attributo è maggiore di X e poi abbiamo che la groupBy mi calcola il massimo. Prima di fare la groupBy posso togliere tutti gli attributi che sono minori di X e poi su quelli che rimangono faccio la groupBy. Facciamo la stessa cosa per il minimo.

Ottimizzazioni di questo tipo sono pensate per i sistemi che vengono usati per l'analisi dei dati.

Se ci troviamo ad avere sia la groupBy che la join, il modo standard consiste nel fare prima la join e poi la groupBy.

Per avere una ottimizzazione e quindi produrre un access plan fisico che costa di meno rispetto al primo possiamo pensare di spostare la join dopo la groupby perchè in questo modo il sort che è necessario prima della groupBy verrebbe effettuato su una tabella più piccola rispetto a quella su cui facciamo il sort.

Questo tipo di ottimizzazione la possiamo svolgere solamente in condizioni specifiche.



Questa tecnica del pre grouping quindi mi fa spostare il groupBy.

La condizione basilare è che la join non deve essere espansiva, per ogni linea di R devo avere una sola linea di S e questo avviene quando abbiamo  $\text{FK}(R)=\text{PK}(R)$ .

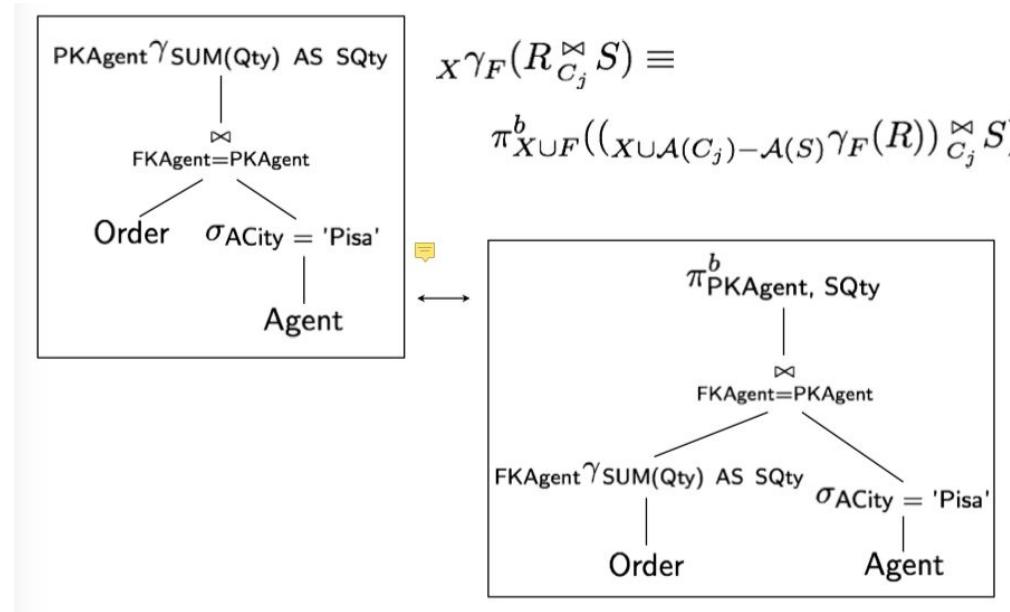
Per muovere la groupBy all'interno:

- Dobbiamo cambiare gli attributi del groupBy perchè potremmo avere degli attributi che arrivano da R ma anche alcuni che arrivano da S, dopo questa riscrittura gli attributi del groupBy devono utilizzare solamente gli attributi di R.
- Come diminuiamo il numero di attributi?  
Eliminiamo gli attributi di S, potremmo avere il caso in cui anche se elimino gli attributi di S ottengo sempre le stesse regole, se avviene che abbiamo sempre le stesse regole e se l'attributo che stiamo eliminando è determinato funzionalmente allora possiamo eliminare l'attributo.

Consideriamo gli studenti, raggruppiamo per studentID e Names,

ho un gruppo per ogni studente, se raggruppo per studentID solamente, ho comunque un gruppo per ogni studente perchè lo studentID determina il nome. Se invece raggruppo per Name ho gruppi in cui abbiamo studenti differenti ma con nomi uguali. Quindi posso eliminare l'attributo senza cambiare la dimensione delle rules solamente se l'attributo è ridondante ed è determinato dagli altri. Se l'attributo X determina tutti gli attributi in A(S) allora posso eliminare gli attributi di A(S). Alla fine dobbiamo fare la proiezione su X U F per essere sicuri che il risultato della query di sinistra sia uguale al risultato della query di destra.

Esempio:



Prima la groupBy veniva fatta su PkAgent, ora la spostiamo di sotto quindi non abbiamo più l'agent quindi non possiamo usare PkAgent. Però possiamo usare la formula:

$$x \gamma_F(R \bowtie_{C_j} S) \equiv \\ \pi_{X \cup F}^b((x \cup \mathcal{A}(C_j) - \mathcal{A}(S)) \gamma_F(R)) \bowtie_{C_j} S)$$

Prendiamo PkAgent, aggiungiamo gli attributi della join condition, rimuoviamo pkAgent perchè dobbiamo togliere l'attributo di S che in questo caso è agent. Quindi rimane FKAgent e possiamo spostare quindi groupBy prima della join.

## Generazione degli access Plan fisici

Se vogliamo trovare il miglior access plan da utilizzare per eseguire una certa query abbiamo vari possibili algoritmi.

Ad esempio l'algoritmo più banale consiste nel generare tutti i possibili access plan e calcolare il costo di ognuno di essi.

Se all'interno della query per cui vogliamo calcolare un access plan fisico abbiamo poche tabelle questa procedura si può anche fare, se però all'interno della query utilizziamo tante tabelle diventa troppo costoso, ad esempio con 10 tabelle abbiamo  $10!$  possibili access plan e quindi il costo è esponenziale rispetto alla dimensione della query.

L'obiettivo finale di questa procedura è trovare l'access plan fisico che ha il costo minore e per farlo è necessario comprendere il costo degli operatori fisici che utilizziamo, la dimensione dei risultati che otteniamo da ogni operatore e anche se il risultato che otteniamo è ordinato o no. Per valutare il costo, l'ottimizzatore memorizza un catalog che contiene tutte queste statistiche appena indicate (numero di record e di pagine delle varie relazioni, numero di chiavi e di foglie...).

Il catalog viene aggiornato con il comando update statistics.

Il caso più semplice per la generazione di un access plan è quello in cui abbiamo una sola tabella da cui leggere, in questo caso dobbiamo solamente capire da quale indice leggere, se abbiamo un solo indice che però ha tutti gli attributi che vogliamo usare nella query allora possiamo usare un IndexOnly.

Se la query che vogliamo considerare (e per cui vogliamo cercare un access plan che sia il meno costoso) legge i dati da varie tabelle e quindi

ha delle join al suo interno, possiamo creare tanti piani di accesso differenti.

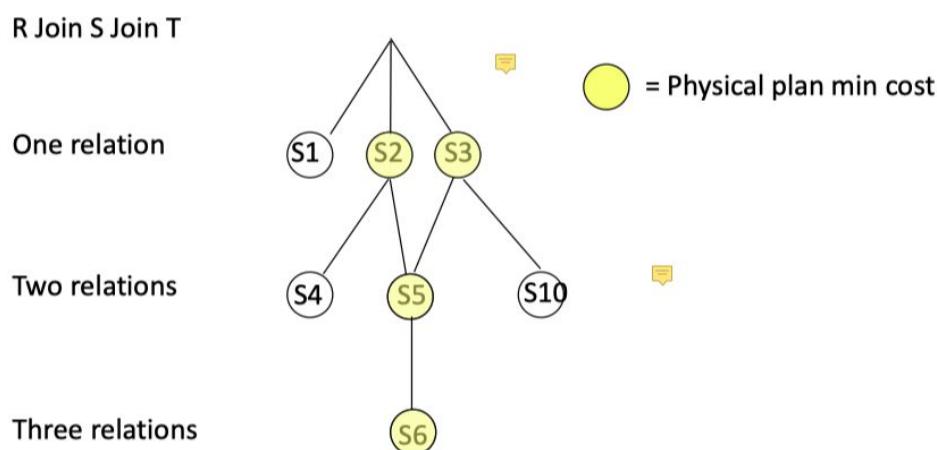
I vari access plan che possiamo creare dipendono dal modo in cui posizioniamo le parentesi all'interno delle join, quindi dipende dall'ordine in cui vengono prese in considerazione le varie relazioni.

- $Ax(Bx(CxD))$
  - $(Ax B)x(CxD)$
  - $(Ax(BxC))xD$
  - $Ax((BxC)x D)$

Una situazione di questo genere ci porta ad avere un numero esponenziale di possibili opzioni per eseguire la join.

Full Search è un algoritmo che ci permette di trovare un access plan ottimo senza però generare tutti i possibili access plan andando ad eseguire il pruning di alcuni rami dell'albero che non devono essere espansi.

In questo modo non eseguiamo il check di tutti gli access plan ma comunque otteniamo l'access plan migliore.



Esempio di Full Search con 3 relazioni:

- Assumiamo di dover fare una join tra R, S e T, la prima cosa è trovare l'access plan ottimo per R, S e T. Questo perchè oltre alla join avremo alcune condizioni sulle tre relazioni.
- Poi si decide quali delle tre ha il costo minimo e prendo quella, quindi in questo caso abbiamo S1, S2 e S3 e prendo quella di costo minimo.
- Consideriamo S2 e ho due possibilità, Join con R e Join con T e quindi calcoliamo l'access plan ottimo per R join S e per R join T e ora ho 4 costi differenti per queste join, di questi ne avrò uno di costo minimo.
- Il fatto è che dobbiamo considerare anche tra i vari costi anche quello di S3 che avevamo calcolato in precedenza, quindi siccome questo è il costo minimo allora possiamo utilizzarlo per eseguire il prossimo step.
- Poi passiamo a vedere l'access plan migliore tra i vari che abbiamo e quindi partiamo da S5 che ha il costo minore ed eseguiamo la terza join.
- Dopo la terza join avremo un'altra foglia nell'albero e dobbiamo vedere se il costo di questa foglia è minore degli altri, in questo caso è migliore quindi non è necessario espandere gli altri nodi.

Quindi alla fine possiamo cercare all'interno dello spazio trovando l'ottimo reale senza rinunciare però a trovare l'ottimo.  
 Questo algoritmo comunque è esponenziale, anche se non consideriamo tante espansioni, se siamo molto fortunati possiamo avere anche un caso lineare perchè potremmo trovare ogni volta la soluzione migliore e quindi non tornare mai indietro ad espandere i nodi che non ho espanso.

```

repeat {
    extract from Plans the fastest plan P
    if P is complete, exit.
    else, expand P:
        join P with all other plans P' on disjoint relations
        for each P join P', put the best tree in Plans
        remove P
}
  
```

Il codice dell'algoritmo, abbiamo Plans che è la struttura dati ordinata che contiene tutti i plan che abbiamo generato fino a questo momento. Manteniamo Plans ordinato in base al costo, quindi in ogni momento il primo access plan in Plans è quello che costa di meno e ad ogni ciclo dell'algoritmo seleziono l'access plan migliore.

Quando estraggo l'ultimo nodo dell'access plan completo allora abbiamo finito l'esecuzione dell'algoritmo.

Se invece quello estratto non è l'ultimo nodo dell'access plan, devo estrarre questo nodo ed espanderlo ovvero devo guardare a tutti gli altri nodi e fare una join con gli altri. Ad esempio, quando espando S2, faccio la join con S1 e con S3, quindi tolgo dal Plans S2 ma aggiungo i due risultati S4 e S5.

Primo punto importante: In principio per questo algoritmo vorrei utilizzare per ogni relazione l'access plan migliore ma se voglio anche avere la possibilità di usare indexNestedLoop non è necessario accedere alla seconda relazione con il suo access plan migliore, mi basta l'index filter. Se faccio una hashjoin invece mi serve il migliore per entrambi.

Secondo punto importante: non è abbastanza calcolare per ogni relazione il suo miglior access plan, assumiamo di avere una query che deve anche essere ordinata in base al family name. Se uso l'index filter su family name e ho i dati ordinati con il family name devo considerare il miglior access plan ma anche gli access plan che non sono i migliori ma che presentano un ordine particolare dei dati che può essere interessante per eseguire in seguito il groupBy, l'ordinamento o il distinct più rapidamente.

Un ottimizzatore reale ha la nozione di “sort interesting” e capisce se questo specifico access plan può essere migliore di un altro.

Quindi ci sono tanti modi per gestire questa situazione di ordinamenti interessanti, l'algoritmo Full Search però non gestisce gli ordinamenti particolari e nemmeno l'indexNestedLoop.

## Ottimizzazioni dell'algoritmo:

L'algoritmo Full Search comunque è troppo costoso perché in ogni caso è esponenziale, quindi possiamo utilizzare alcune euristiche per renderlo più veloce:

- Left Deep: una prima possibile ottimizzazione per l'algoritmo consiste nell'espandere gli alberi solamente a sinistra eseguendo la join da quel lato dell'albero, nella parte destra invece non facciamo le join. In molti casi utilizzando questa euristica otteniamo un access plan che non è troppo lontano da quello ottimo.
- Greedy: dopo aver espanso un nodo non torniamo indietro e espandiamo solamente quel nodo e i successivi. Non facciamo backtrack e questo comporta che se prendiamo una decisione errata all'inizio poi la stessa decisione errata ce la portiamo dietro anche agli step successivi.  
In questo caso controlliamo un numero quadratico di scelte possibili, non abbiamo quindi un costo esponenziale ma otteniamo comunque un buon risultato.
- Iterative Full Search: alterniamo l'algoritmo full search con l'algoritmo Greedy, in particolare per una serie di livelli usiamo il Full Search e poi il greedy scegliendo sempre un nodo senza espandere gli altri.
- Interesting-order: in questo caso consideriamo degli access plan che sono “interessanti”. Interessanti nel senso che se ad esempio i dati sono ordinati allora potrebbe essere utile per quello specifico access plan (Full Search non considera l'interesting order).

## Un esempio di ottimizzazione:

Data questa query e queste relazioni possiamo fare subito una prima ottimizzazione statica andando ad inserire la selezione prima di fare la join:

$R(N, D, T, C)$ , with indexes on C and T  
 $S(C, O, E)$ , with indexes on C and E

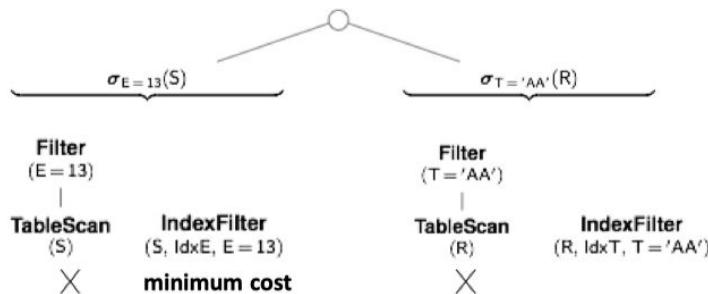
```
SELECT S.C, S.O
FROM S, R
WHERE S.C = R.C AND E = 13 AND T = 'AA';
```

 $\pi_{S.C, S.O}^b(\sigma_{E=13 \wedge T='AA'}(S \bowtie R))$ 
 $\pi_{S.C, S.O}^b(\sigma_{E=13}(S) \bowtie \sigma_{T='AA'}(R))$ 

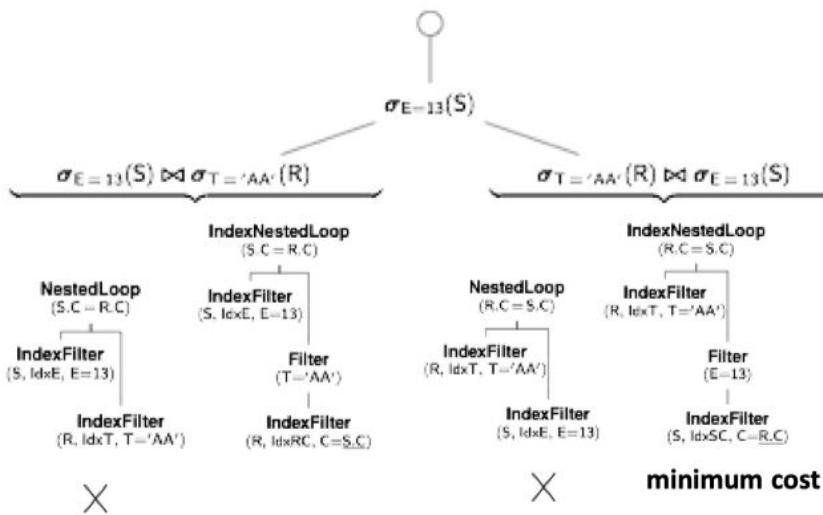
Poi dobbiamo considerare il costo per l'accesso alle due relazioni S ed R, quindi proviamo a vedere quale access plan sia migliore in questa situazione e alla fine scegliamo quello che ha il costo minimo.

 $\pi_{S.C, S.O}^b(\sigma_{E=13}(S) \bowtie \sigma_{T='AA'}(R))$ 

Physical plans for subexpression on relations

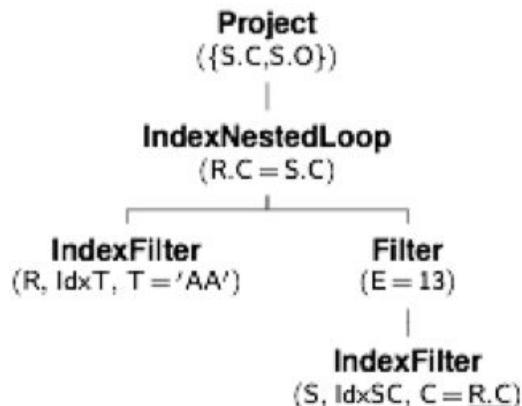


Generiamo poi i possibili access plan per l'esecuzione della Join e anche in questo caso scegliamo quello di costo minimo.



Alla fine otteniamo un access plan fisico che ha il costo minimo tra i vari che potevamo prendere in considerazione:

### Final physical plan



## Lezione 9: Transactions

Quando viene creato un database abbiamo dei vincoli di integrità statici che devono essere garantiti in qualsiasi stato del database, indipendentemente dalle modifiche che effettuiamo. Ci sono poi altri vincoli che sono dinamici e mi limitano le possibili transizioni tra gli stati.

**Uno stato consistente del database è uno stato in cui sono soddisfatti tutti i vincoli.**

Per passare da uno stato consistente all'altro effettuiamo delle transazioni.

Una transazione è una sequenza di una o più operazioni SQL definite dal programmatore ed eseguite dal DBMS che deve trattarle come se fosse un'unica operazione atomica.

Una transazione ha le seguenti proprietà:

- **Atomicity (NO COMMIT NO EFFECT):** se una transazione fallisce, non ci sono effetti parziali sulla base di dati. Solamente quelle che terminano normalmente, ovvero quelle “committed” possono modificare il database, quelle “aborted” invece non comportano modifiche e quindi lo stato rimarrà immutato come se quell’operazione non fosse mai accaduta.  
L’atomicity riguarda le transazioni che non vanno a buon fine, possiamo pensarla dal punto di vista logico come  
Effect => commit  
Ovvero la modifica del database implica che ci sia stato un commit.  
L’atomicity dice se non ho un commit non troverò modifiche nel database (Not commit => Not effect).
- **Durability (IF I HAVE A COMMIT THEN I WILL FIND AN EFFECT AFTER A CRASH):** è la situazione duale, in questo caso si parla di transazioni che hanno fatto il commit e che quindi devono comportare una modifica al database. In particolare se il DBMS crasha dopo un commit, la transazione deve comunque rimanere nel database.

La durability mi dice che se ho un commit allora troverò la modifica nel database.

Atomicity e Durability sono due problemi duali, se voglio garantire l'atomicity posso utilizzare i “deferred update” ovvero solamente dopo il commit sposto l'aggiornamento nel database, però se dopo il commit e prima dell'update ho un crash non garantisco la durability e la transazione non finisce nel database.

Se invece uso la tecnica “deferred commit” posso spostare prima la transazione nel disco e poi alla fine eseguo il commit, in questo caso però perdo l'atomicità.

Non c'è un modo ovvio di avere l'atomicity e la durability nello stesso momento perchè l'atomicity prima aggiorna e poi committa mentre la durability è il contrario.

- Isolation: in questo caso si parla di concorrenza, se abbiamo varie transazioni che devono essere eseguite, dobbiamo fare in modo che queste siano eseguite contemporaneamente senza però che ognuna di queste veda i risultati intermedi delle altre. Alla fine dell'esecuzione in parallelo bisognerà fare in modo che la situazione finale sia la stessa che avremmo se ogni transazione venisse eseguita singolarmente.

La versione più forte dell'isolation è la serializability.

I due problemi, Atomicity e Durability hanno a che fare con i fallimenti e non con la concorrenza, la parte del DBMS che se ne occupa è il “Transaction and Recovery Manager”, invece la parte che si occupa di garantire l'isolation è il “Concurrency Manager”.

Oltre alle tre proprietà appena descritte ce ne sta anche un'altra che è la consistenza, queste quattro proprietà tutte insieme vengono indicate con l'acronimo ACID (Atomicity, Consistency, Isolation, Durability).

La proprietà Consistency indica varie cose tutte insieme, per prima cosa cosa il sistema garantisce la possibilità di definire dei vincoli e poi il sistema userà l'atomicity per verificare che ci sia consistenza. Ad esempio se ho una transazione e ho un vincolo che mi dice che il balance non può essere minore di 0, se qualcuno fa una transazione che fissa il balance ad un valore minore di 0 allora il sistema blocca la

transazione e riporta lo stato alla situazione che avevo prima che la transazione venisse eseguita. Quindi è la combinazione di vincoli di integrità e di atomicity.

Tutti i DBMS hanno la possibilità di implementare vincoli di integrità e questi, combinati con il fatto che ripuliamo ogni possibile problema causato dalle transazioni formano la consistency.

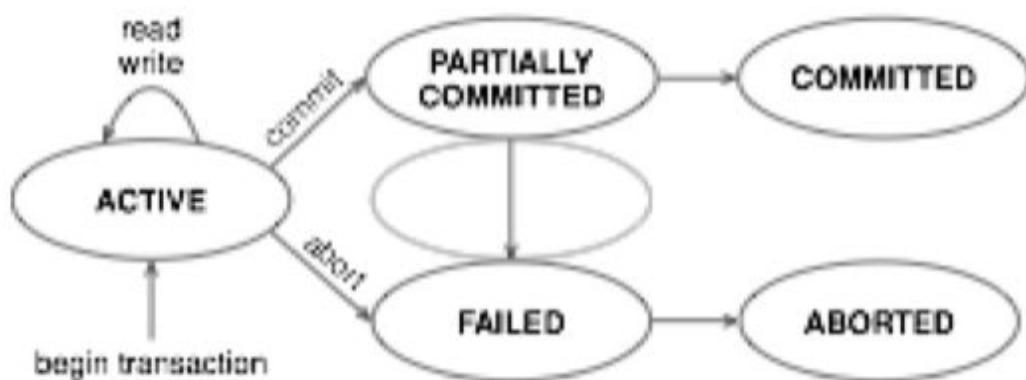
Quindi la consistency non è una proprietà vera e propria perché deriva dall'atomicity.

Delle proprietà ACID quindi le più importanti sono Atomicity, Isolation e Durability.

## Atomicity e Durability

Un modo astratto per parlare di una transazione dal punto di vista del DBMS è che la transazione è una sequenza di operazioni lettura/scrittura/lettura/scrittura che vengono richieste da un codice scritto dal programmatore.

Abbiamo una operazione “**beginTransaction**”, il transaction Manager assegna a questa operazione beginTransaction un transactionID. In questo modo le operazioni successive di lettura e scrittura verranno associate a questo stesso TransactionID, alla fine avremo o il commit della transazione con quel TransactionID oppure l'abort.



Il commit è una richiesta del **query Manager** e diventa una azione solamente dopo che il **Transaction Recovery Manager** dice che può essere eseguito il commit (può anche rifiutare).

Tra la richiesta di commit e lo stato di committed c'è una differenza, non sono la stessa cosa perchè nel mezzo c'è un processo che mi porta nello stato di committed.

**Abort** invece può essere richiesto sia dal Query Manager che dal Transaction Recovery Manager, il Transaction Recovery Manager può decidere di inviare una richiesta di abort in qualsiasi momento, indipendentemente dal Query Manager, questo perchè l'operazione viola un vincolo di integrità ad esempio oppure per motivi tecnici, ad esempio il disco che non funziona correttamente.

Fallimenti:

Il transaction Manager è la parte del DBMS che garantisce atomicity e durability quando si verifica un fallimento.

Quando consideriamo un DBMS distinguiamo vari possibili tipi di fallimento:

- **Transaction failure:** in questo caso la transazione viene bloccata a causa di un'azione che è stata svolta dalla transazione stessa (ad esempio quando si viola un vincolo di integrità). La transazione viene bloccata ma il resto del sistema continua a funzionare.  
C'è anche un altro caso di Transaction Failure, è quando il sistema decide di bloccare la transazione, ad esempio per risolvere una situazione di deadlock, anche in questo caso si tratta comunque di transaction failure perchè il fallimento riguarda solamente un numero limitato di transazioni ma il sistema continua a funzionare correttamente.
- **System Failure:** Problema del DBMS o del sistema operativo che riguarda la memoria principale ma non il disco.  
Lo stato della memoria principale in questo caso non è corretto. È una condizione comune, ad esempio quando salta la corrente quello che stava nella ram si perde ma quello che sta nel disco rimane salvato, oppure capita se c'è un bug nel DBMS.

- **Media Failure:** è la situazione peggiore, riguarda il disco, quindi la memoria permanente. È molto più complicato gestire un fallimento di questo genere.

Come ci si protegge da problemi di questo genere?

La tecnica per gestire problemi di questo genere è molto semplice, ogni notte viene effettuato un backup del database inoltre, per ogni azione che è stata eseguita nella base di dati viene salvato un log.

All'interno del log mi salvo quando una transazione inizia, quando scrive o legge, quando committa e quando c'è un abort. Per ogni transazione devo memorizzare anche l'ID della transazione che sto prendendo in considerazione perchè all'interno del log avremo un interliving di scritture, letture, inizi, abort e commit.

Quando viene modificata una pagina del database devo anche memorizzare nel log quale pagina è stata modifica, il contenuto che aveva in precedenza e il nuovo contenuto.

In questo modo quando abbiamo un crash sarà semplice comprendere l'errore e ricreare uno stato coerente del database.

Prendiamo infatti il database che abbiamo salvato come backup, poi scorriamo il log e consideriamo le transazioni:

- ReDo Operation: abbiamo il committ di una transazione e poi iniziamo a scrivere la nuova pagina all'interno del database, può capitare un fallimento. Dal log vediamo che c'è stato un commit ma che non abbiamo fatto la scrittura nel database, quindi a questo punto dobbiamo andare a memorizzare nel database la AfterImage ovvero i nuovi dati che non avevo memorizzato in precedenza.
- Undo Operation: se un aggiornamento di una transazione non committata arriva nel database devo eseguire l'algoritmo di UnDo e quindi all'interno del database devo copiare la BeforeImage nella pagina che ho modificato.

Se il backup è stato fatto prima che la transazione cominciasse allora non devo proprio fare l'UnDo della transazione.

Questo è il modo in cui tutti i sistemi sono in grado di fornire atomicity e durability nello stesso momento. Memorizzo la beforeImage e l'afterImage e poi le uso per garantire atomicity e durability.

In particolare beforeImage viene utilizzata per l'atomicity perché se faccio abort non devo avere effetti, quindi reinstallo la beforeImage. L'afterImage invece è per garantire la durability.

## CheckPoint

Uno dei metodi per proteggersi dai fallimenti all'interno di un DBMS consiste nell'esecuzione di un backup (ad esempio a mezzanotte) e nella scrittura di un file di log.

Il problema di questo sistema è che se noi abbiamo un fallimento dopo tante ore dall'esecuzione del backup, dovremo prendere il backup fatto a mezzanotte, scorrere tutto il file di log ed eseguire tutte le operazioni che troviamo nel file.

Questa procedura ha un costo alto, e potremmo trovarci a non poter fornire un servizio per un'ora ad esempio.

Quindi questo metodo non va bene per il recupero perché il tempo di restart è troppo lungo.

Per limitare il tempo di restart e quindi il lavoro che si deve svolgere in caso di fallimenti, molti sistemi implementano dei checkpoint.

Con i checkpoint non rendiamo il sistema più robusto evitando i fallimenti ma diminuiamo il tempo di restart, è preferibile avere 10 crash al giorno con un restart time di 1 minuto che 1 crash al giorno ma con un restart time di 1 ora.

Il checkpoint è un metodo che mi permette di dire che tutto quello che è successo prima del checkpoint va bene quindi lo stato fino al checkpoint è consistente. Se creo checkpoint ogni 5 minuti e ho un crash, devo ricontrillare solamente quello che è avvenuto negli ultimi minuti e quindi ho un restart time molto breve.

Per gestire i checkpoint ci sono vari algoritmi differenti:

- **Commit Consistent checkpoint:** ogni 10 minuti creo un checkpoint, l'algoritmo funziona in questo modo:

- Smettiamo di accettare nuove transazioni
- Attendiamo che le transazioni in corso terminino
- Tutte le pagine presenti nel buffer che hanno subito una modifica sono scritte sul disco e scrivo anche nel log cosa ho fatto
- Scrivo il punto del checkpoint nel log

Questo sistema è in pratica inutilizzabile soprattutto per il fatto che devo fermarmi, non accettare nuove transazioni e attendere quelle in corso. Se una transazione ci mette 5 minuti a completarsi, io attendo 5 minuti e il servizio non è disponibile per 5 minuti. Quindi in pratica questo metodo è inutilizzabile e non viene usato.

- **Buffer Consistent Checkpoint:** ogni 10 minuti faccio un checkpoint senza però attendere, l'algoritmo funziona in questo modo:
  - Blocco l'attivazione di nuove transazioni
  - Devo fare il flush del buffer, quindi faccio la scansione delle pagine del buffer e per ogni pagina che ha il dirty bit a 1, la scrivo nel disco e il dirty bit viene portato a 0. Avrò delle transazioni che sono ancora attive e che non ho ancora scritto nel disco.

Se ho un crash so che **tutte le transazioni che hanno fatto il commit prima del checkpoint sono sul disco** mentre quelle che erano ancora in esecuzione nel momento del checkpoint possono comportare degli effetti parziali e quindi per queste devo distinguere due situazioni:

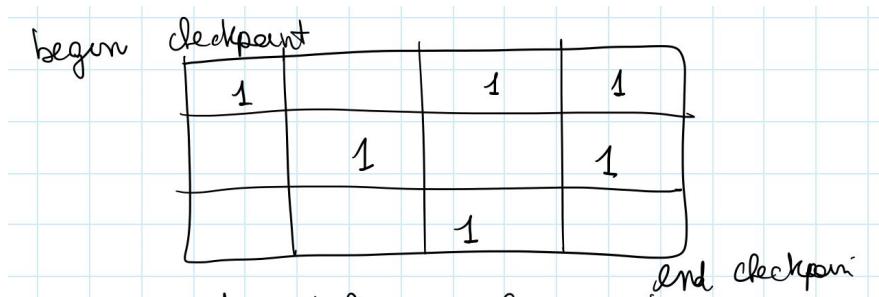
- **Se hanno fatto il commit dopo il checkpoint** allora devo eseguire di nuovo tutte le operazioni che queste operazioni hanno svolto **dopo il checkpoint**.
- Se hanno avuto un **abort** o **non sono state committate** allora devo fare **l'undo** delle operazioni che queste transazioni hanno fatto **prima del checkpoint**.

Il checkpoint non è un firewall completo, è un firewall solamente per le operazioni di redo perchè non devo rifare tutto quello che è accaduto prima del checkpoint, la stessa cosa non la posso dire per l'undo operation. Questo è un problema minore perchè la maggior parte delle

transazioni di solito fanno il commit e perchè tutte le transazioni che sono finite nel momento del checkpoint non vanno toccate.

Un problema di questo metodo è che devo fare il flush di tutto il buffer che, in alcune situazioni può essere di grandi dimensioni.

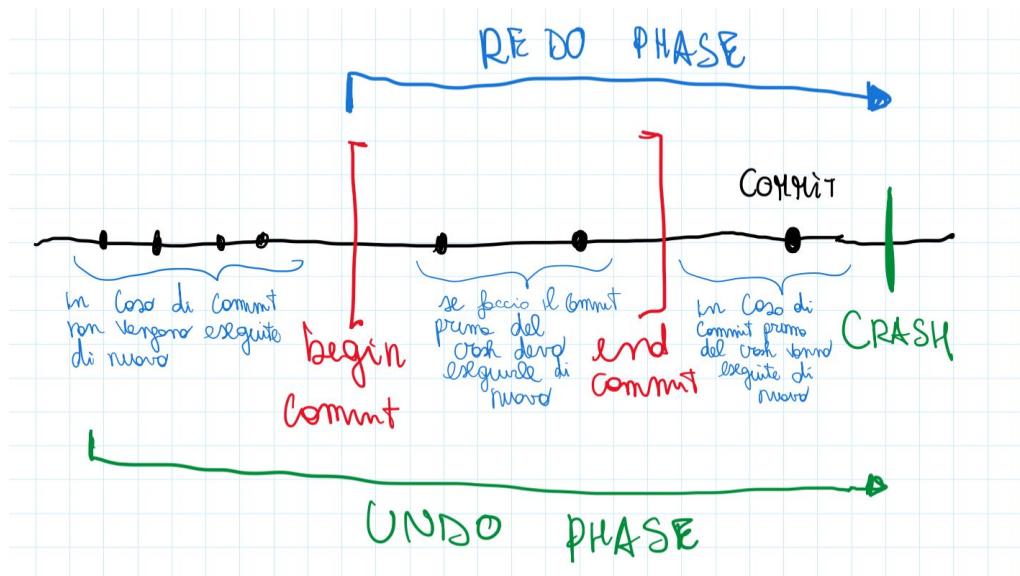
- **No Stop CheckPoint:** è un metodo simile al buffer consistent checkpoint ma abbiamo due thread che lavorano in parallelo, uno esegue tutte le richieste delle transazioni e l'altro esegue il flush del buffer (attualmente si usano anche più thread).



Dato il buffer abbiamo alcune pagine con il dirty bit a 1, parte il checkpoint thread che trova le pagine a 1 ed esegue il flush sul disco.

In parallelo abbiamo il thread delle transazioni, questo le esegue e poi può a sua volta modificare le pagine del buffer, il thread che controlla il buffer le vede e fa il flush sul disco anche di queste qua. Ad un certo punto si arriva alla fine del checkpoint. Ho uno slice temporale tra l'inizio e la fine del checkpoint.

Durante questo periodo posso avere una transazione T1 che ha varie operazioni all'interno, alcune delle operazioni possono avvenire tra l'inizio e la fine del checkpoint.



Assumiamo di avere un crash dopo la fine del checkpoint e che prima del crash la transazione ha fatto il commit.

Quello che avviene prima del begin checkpoint non devo eseguirlo di nuovo perché il checkpoint ha già fatto il flush del buffer quindi se hanno fatto modifiche al buffer, queste modifiche saranno già state scritte sul disco. Le operazioni che sono state eseguite dopo il checkpoint le devo eseguire di nuovo perchè potrebbero ancora essere nel buffer e potrebbero non essere state scritte nel disco.

Fare il redo equivale a mettere nel disco l'after image, mettere questa immagine nel disco è una operazione "safe", è una operazione idempotente, anche se la faccio più volte non cambia niente.

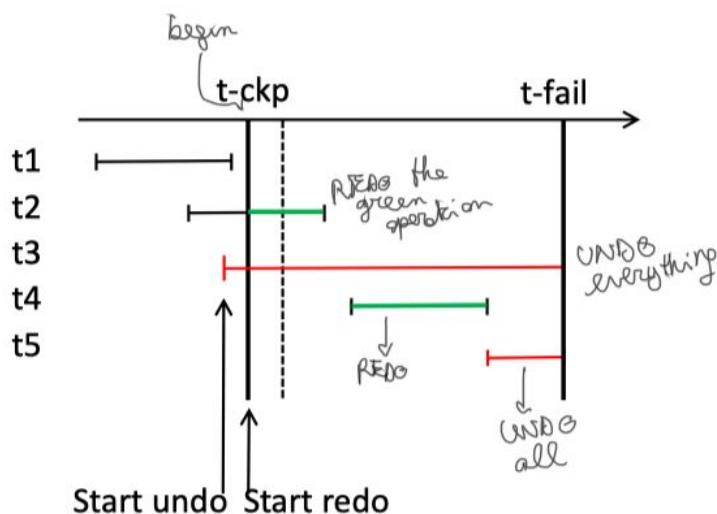
Le operazioni che sono state eseguite tra l'inizio e la fine del checkpoint le dobbiamo fare di nuovo, quindi anche qua usiamo la redo operation perchè non sappiamo se il checkpoint thread è stato in grado di prendere queste operazioni e portarle nel disco. La fase di redo inizia nel punto del begin checkpoint e va avanti fino al momento del crash.

Se invece prima del crash non ho il commit allora quando ho il crash e devo recuperare devo considerare tutte le varie operazioni che sono state effettuate e che riguardano quella transazione e devo fare l'undo di tutte le operazioni, indipendentemente da dove

si trovano. Quindi l'undo Phase è più lunga della redo phase.

È buona norma avere dei checkpoint che vengono creati molto spesso e che richiedono poco tempo per ripristinare la situazione in cui lo stato è consistente.

## Come funziona l'algoritmo del Restart



Consideriamo l'algoritmo di Restart, nel disegno sopra abbiamo un punto in cui viene creato il checkpoint e un punto in cui abbiamo il fallimento. Il punto del checkpoint mi dice che tutto quello che ha committato prima non deve essere eseguito nuovamente, quindi la transazione T1 non la prendiamo in considerazione. Quello che c'è tra il checkpoint e il momento del fallimento va gestito in modo differente.

L'algoritmo lo possiamo dividere in due fasi:

- RollBack (fa le undo operations): partiamo dal momento del fallimento e andiamo indietro leggendo il log in modo da fare l'undo delle operazioni che non hanno portato al commit e il redo di quelle che invece hanno committato.

In questa fase utilizziamo due set, uno è  $L_r$ , quello delle transazioni che devono essere eseguite nuovamente, l'altro è  $L_u$  che è il set di quelle transazioni che devono essere cancellate.

```

- ckp=false; toUndo=toRedo={};
  for backward r in log           -- rollback
    until (ckp and empty(toUndo)) {
      if r = (commit,T) and not ckp then toRedo+={T};
      elseif r = (write,T,x,bi,ai) and not (T in toRedo)
          then {toUndo+={T}; undo(x,bi)}
      elseif r = (begin,T) then toUndo-= {T}
      elseif r = (b-ckpt,TList) then {ckp=true;
→ we found the checkpoint
quelle che furono non committute
          toUndo+=TList-toRedo}
    }
  }

```

*begin  
la  
la  
begin  
list  
begin checkpoint with  
a list*

*add to Undo but all the transactions that*

L'algoritmo è il seguente, si va avanti fino a quando ckp != true e la lista Lu ovvero ToUndo non è vuota:

- Se nel log troviamo un commit allora quella transazione viene aggiunta alla lista Lr perchè deve essere eseguita nuovamente
  - Se troviamo una operazione di scrittura ma la transazione corrispondente non è in Lr allora facciamo l'undone dell'operazione e poi inseriamo nella lista Lu la transazione corrispondente.
  - Se troviamo l'operazione di Begin di una transazione, eliminiamo la transazione dalla lista toUndo
  - Quando troviamo il checkpoint prendiamo la lista delle transazioni che sono iniziate prima del checkpoint e aggiungiamo alla lista delle transazioni che devo annullare la differenza tra l'insieme delle transazioni iniziate prima del checkpoint e l'insieme delle transazioni che devono essere svolte nuovamente. Settiamo anche il booleano del checkpoint a true.
- Poi si fa l'operazione di roll forward.
- RollForward (fa le redo operation): parto dal momento del checkpoint e poi guardo se l'operazione che trovo nel log è una operazione di commit o no. Se è una operazione di write e la transazione corrispondente è nella lista dei Redo allora eseguo il redo della operazione. Se invece ho una operazione di commit

allora tolgo la transazione dal set dei Redo.

È importante fare prima la fase di Undo e poi la fase di Redo, se eseguo l'algoritmo nell'ordine sbagliato posso fare un disastro se lavoro su una stessa pagina su cui faccio entrambi le operazioni.

Se invece faccio l'undo di una operazione e riporto alla situazione che avevo nel passato e poi faccio il redo e metto le pagine corrette del futuro va bene.

Se una pagina è stata modificata solamente da un redo o da un undo non ci sono problemi.

Esempio di esecuzione dell'algoritmo di restart:

Log					
LSN	Record	LSN	Record	LSN	Record
1	(begin, 1)	6	(begin, 3)	12	(begin, 5)
2	(W, 1, A, 50, 20)	7	(W, 3, A, 20, 30)	13	(W, 5, E, 50, 30)
3	(begin, 2)	8	(CKP, {2, 3})	14	(commit, 2)
4	(W, 2, B, 10, 20)	9	(W, 2, C, 5, 10)	15	(W, 3, B, 20, 30)
5	(commit, 1)	10	(begin, 4)	16	(commit, 4)
		11	(W, 4, D, 5, 30)		

rollback				
LSN	Operation	L <sub>r</sub>	L <sub>u</sub>	Action
16	(commit, 4)	{4}	{}	no action
15	(W, 3, B, 20, 30)	{4}	{3}	undo: B = 20
14	(commit, 2)	{4, 2}	{3}	no action
13	(W, 5, E, 50, 30)	{4, 2}	{3, 5}	undo: E = 50
12	(begin, 5)	{4, 2}	{3}	no action
11	(W, 4, D, 5, 30)	{4, 2}	{3}	no action
10	(begin, 4)	{4, 2}	{3}	no action
9	(W, 2, C, 5, 10)	{4, 2}	{3}	no action
8	(CKP, {2, 3})	{4, 2}	{3}	no action
7	(W, 3, A, 20, 30)	{4, 2}	{3}	undo: A = 20
6	(begin, 3)	{4, 2}	{}	L <sub>u</sub> is empty, end of phase

**Figure 9.11:** Actions during the *rollback* phase

rollforward			
LSN	Operation	L <sub>r</sub>	Action
9	(W, 2, C, 5, 10)	{4, 2}	redo: C = 10
10	(begin, 4)	{4, 2}	no action
11	(W, 4, D, 5, 30)	{4, 2}	redo: D = 30
12	(begin, 5)	{4, 2}	no action
13	(W, 5, E, 50, 30)	{4, 2}	no action
14	(commit, 2)	{4}	no action
15	(W, 3, B, 20, 30)	{4}	no action
16	(commit, 4)	{}	L <sub>r</sub> is empty, end of phase

**Nota Bene (importante):**

È importante considerare un “dettaglio” piuttosto importante, tutte le operazioni di restart descritte fino ad ora vengono eseguite nel buffer e non nel disco. L’idea principale è di usare il buffer che poi lavora (tramite il buffer manager) in modo indipendente e può decidere quando eseguire il flush su disco e quando non eseguirlo.

Fare il restart nel buffer ci garantisce una velocità maggiore rispetto all’esecuzione della stessa procedura sul disco ed è importante perché noi vogliamo proprio che il restart sia veloce.

L’obiettivo del restart è permettere al cliente di continuare a lavorare servendo quindi le sue richieste, l’obiettivo non è rendere il database consistente nel disco perché il database che sta nel disco è già

consistente se lo consideriamo in coppia con il log.

L'obiettivo è avere un buffer che sia in uno stato coerente in modo da poter rispondere alle richieste dell'utente.

Se invece consideriamo solamente il disco come database allora non abbiamo mai la consistenza perchè dal buffer posso scrivere in qualsiasi momento e dove voglio.

Se durante il periodo di un restart avviene un fallimento non ci sono problemi, eseguiremo un ulteriore restart.

La maggior parte dei database usano 2 copie dei log, su due dischi separati e poi c'è il disco con il database.

### **Aries (Solo accennato, non fa parte del programma)**

La maggior parte dei sistemi non usano gli algoritmi visti fino ad ora per il restart ma usano un algoritmo più complesso che è chiamato Aries.

Ci sono molti dettagli da considerare in Aries soprattutto per la gestione delle performance.

Aries non usa il checkpoint facendo il flush del buffer ma usa un'idea differente, l'algoritmo mantiene informazioni su quali pagine sono dirty al momento del checkpoint e quali transazioni sono relative alle pagine che sono dirty. Quindi non fa il flush delle pagine dirty ma si limita a segnare nel log nel momento del checkpoint quali sono le pagine che sono dirty e cosa contengono in modo da poterle ricreare al momento del crash.

Questo rende il restart più lento ma il checkpoint è molto veloce quindi posso farlo ogni minuto e ho più checkpoint.

I checkpoint sono “virtuali” in pratica.

Altro punto interessante, nel log non metto tutte le informazioni che riguardano una pagina ma solamente le differenze rispetto alla precedente scrittura della pagina, è una versione compressa.

## **Recovery Algorithms**

Ci sono vari tipi di algoritmi per il recovery, in particolare abbiamo 4 tipologie di accoppiamenti:

- Undo-Redo
- Undo-NoRedo
- NoUndo-Redo
- NoUndo-NoRedo

### **Undo Algorithm:**

Ci sono 2 possibilità per gestire la scrittura delle pagine nel database, il deferred update e l'immediate update.

- Il deferred update è un metodo semplice per garantire l'atomicity, in questo caso non facciamo le operazioni di undo perché l'aggiornamento delle pagine del database viene rimandato fino a quando ho il commit.

In questo modo non devo fare l'undo e non devo considerare le transazioni che non hanno fatto il commit, per evitare che una pagina venga aggiornata da una transazione, le pagine sono "pinnate" all'interno del buffer fino a quando non arriva il commit. Il deferred update non viene mai utilizzato in pratica perchè ha un problema, una transazione infatti può toccare più pagine della dimensione del buffer, se ho un buffer di 100 pagine e la transazione ne deve modificare 101 allora la transazione fa crashare il sistema perchè quando chiedo la 101 pagina non la trovo una libera perchè sono tutte pinnate. Il problema si ripresenta appena riavvio il sistema perchè eseguo di nuovo la transazione e crasha di nuovo il sistema.

Il problema quindi è che ho una dimensione massima pari alla memoria temporanea e questo è un problema perchè il DBMS non dovrebbe avere un limite dal punto di vista della dimensione. Per questo motivo l'idea dei deferred update non viene usata. C'è anche una questione legata all'efficienza del buffer manager.

- La seconda possibilità è l'Immediate Update o Free Update o Buffer Stealing che però richiede anche l'operazione di UNDO. Free Update permette al buffer di eseguire il flush dei dati immediatamente o anche nel futuro. Qua il discorso è che io **POSSO** scrivere anche prima che la transazione sia committata. Si chiama anche buffer stealing (opposto di Buffer pinning) perché ho il buffer, scrivo e leggo nel buffer, ad un certo punto non c'è spazio e il buffer manager fa il flush dei dati e ti prende il buffer anche se lo stai usando. Non si aspetta il commit per eseguire la scrittura, ecco perchè è libero di prendere il buffer e fare il flush. C'è molta libertà in questo metodo e la pago perchè devo fare l'undo.

Un DBMS che utilizza i Free update deve anche rispettare la “Log Ahead Rule”, è fondamentale per fare l'undo che quando modifco una pagina prima del commit, prima di fare il flush devo memorizzare nel log anche la Before Page in modo da poterla ripristinare nel caso in cui dovessi avere un crash.

Se prima facessi il flush e poi memorizzassi la pagina nel log potrei avere un problema perchè se c'è un crash nel mezzo la BeforeImage non ce l'ho più nel disco e non l'ho salvata nemmeno nel log.

Quindi prima salvo nel log la before page, poi faccio il flush e memorizzo nel disco la nuova pagina.

## Redo Algorithm

È il duale del primo.

Abbiamo due possibilità:

- NoRedo (deferred commit/Force Writes): in questo caso adottiamo una politica deferred commit, questo ci permette di eseguire il flush del buffer in qualsiasi momento, anche se il buffer manager potrebbe non essere contento. Solamente quando riceviamo il segnale che tutto è stato scritto nel disco allora scriviamo nel log che è stato fatto il commit della transazione. Qua siamo

**OBBLIGATI** a scrivere tutto nel database prima che il commit venga scritto nel log. In questo modo non devo fare il Redo di una transazione perchè siamo sicuri che tutto è stato salvato nel disco perchè ho il commit nel log.

In deferred commit rimando il commit fino a quando tutte le pagine sono state scritte in memoria.

Ci sono dei problemi con questa politica:

- Eseguo poco buffering e perdo la possibilità di fare delle ottimizzazioni sulla scrittura nel disco.
- Nella versione “Force” in cui io faccio il flush più spesso il problema principale è che rimuovo delle libertà dal buffer perchè questo vorrebbe fare una sola scrittura per 10 transazioni ma io non glielo faccio fare perchè ne faccio di più. Questo comporta un delay nel commit e inoltre il transaction manager è meno efficiente.
- Oltre alla versione force c’è anche la versione “Wait” del deferred update che però non è accettabile perchè non posso aspettare che tutto il buffer sia pieno e si faccia solamente una scrittura per poi fare il commit. In alcuni casi questa cosa non è fattibile, ad esempio se va gestita una transazione voglio vedere subito se va a buon fine e non voglio aspettare che si riempia il buffer.
- **BIGGEST PROBLEM:** fino ad ora abbiamo pensato solamente in termine di system failure e in questo caso l’approccio va bene. Se però ho un media failure ho bisogno di un backup e di un log e tramite questo devo prendere le varie modifiche che sono state apportate ed eseguirle di nuovo. Il redo log è comunque necessario per il media failure. Praticamente nessun DBMS commerciale implementa questa soluzione.
- Immediate Commit: in questo caso **POSSIAMO** scrivere nel log il commit anche prima che le pagine siano effettivamente scritte nel database. Quindi ci troviamo a dover fare una procedura di REDO.

## NoUndo NoRedo

C'è una quarta possibilità, quelli No Undo e No Redo.

Per avere NoUndo e NoRedo devo fare allo stesso tempo sia i deferred update (quindi update dopo il commit) sia il deferred commit (quindi commit dopo l'update). Questo è impossibile a meno che non faccio update e commit allo stesso momento.

Mi serve una tecnica che mi permetta di fare l'update e il commit in una sola operazione, la tecnica che è stata inventata è la Shadow Pages technique, abbiamo il disco e una directory ovvero una tabella delle pagine del disco che è una tabella che dice che la pagina 1 del file è in un punto, la 2 in un altro punto (le varie pagine potrebbero essere distribuite nel disco) se vogliamo aggiornare 10 pagine del file nello stesso momento abbiamo una tecnica:

- Abbiamo un descrittore con un puntatore alla tabella delle pagine corrente, ogni cella di questa tabella punta ad un certo indirizzo p di una pagina. Quando devo modificare una transazione creiamo una nuova tabella delle pagine e facciamo puntare le celle di questa tabella alle pagine già esistenti.
- Quando ho una transazione che fa delle modifiche sul disco, creo una nuova pagina e scrivo qua i cambiamenti rispetto alla vecchia.
- Modifico il puntatore della nuova tabella delle pagine e inserisco un puntatore alla pagina che ho appena creato
- La pagina vecchia viene chiamata shadow e non viene modificata
- Quando ho finito aggiorno in modo atomico il puntatore del descrittore che ora non dovrà più puntare alla vecchia tabella delle pagine ma alla nuova tabella delle pagine.

È una tecnica interessante che viene utilizzata spesso nei transactional file system (se il computer crasha, quando lo riaccendo il file system è comunque coerente).

Questo metodo può comportare problemi perchè le pagine occupate dai record della stessa tabella potrebbero non essere vicine tra loro nel disco, inoltre serve un algoritmo per gestire il garbage collection.

## Lezione 10: Concurrency

All'interno di un DBMS abbiamo il Concurrency Manager che è il componente che si occupa di gestire la concorrenza e di fare in modo che le transazioni possano essere eseguite in parallelo senza che entrino in conflitto tra loro.

Consideriamo due transazioni che vengono fatte contemporaneamente a due ATM in cui si chiede di prendere i soldi da uno stesso conto.

A seconda di come vengono svolte le transazioni sui due conti, potremmo ritrovarci in una situazione in cui le due transazioni leggono entrambe un certo valore iniziale e poi lo aggiornano. Il problema è che l'aggiornamento viene fatto da entrambi e entrambi hanno letto un valore iniziale, quindi l'aggiornamento di T2 va a sovrascrivere quello di T1 e ci troviamo con un risultato errato.

T1	T2
r1[x]	
x:=x-250	r2[x]
	x:=x-250
w[x]	
commit	
	w[x]
	commit

La concorrenza è un problema che non è semplice risolvere e che non può essere neanche testata in modo certo. Quindi serve un concurrent manager che possa gestire il problema e garantire che le transazioni vengano eseguite nel modo corretto.

Il concurrency manager si preoccupa di garantire che le scritture e le letture concorrenti avvengano in modo corretto, è differente rispetto al recovery manager che non si preoccupa delle letture e gestisce solamente le scritture.

**Esecuzione seriale:** dato un set T di transazioni, definiamo seriale l'esecuzione delle transazioni se, prese due transazioni Ti e Tj dal set T, le operazioni di Ti vengono eseguite tutte prima delle operazioni di Tj o

vice versa. In questo caso non abbiamo problemi con la concorrenza perché le transazioni non interferiscono tra loro.

Noi non vogliamo l'esecuzione seriale perchè è troppo lenta, vorremmo un interliving di operazioni che vengono eseguite in parallelo.

**Esecuzione serializzabile:** data una serie di transazioni, una esecuzione concorrente delle transazioni è serializzabile se questa esecuzione ha lo stesso effetto che avrebbe una esecuzione seriale delle stesse transazioni.

L'effetto che otteniamo è che le varie transazioni sono in parallelo senza però che le varie transazioni abbiano effetto l'una sull'altra.

In questo caso abbiamo sia il beneficio della concorrenza sia il beneficio della serialità perchè non ho transazioni che interferiscono l'una con l'altra.

Per i vari esempi e per la teoria consideriamo le transazioni come insieme di operazioni che vengono eseguite. Queste operazioni sono solamente lettura e scrittura, poi abbiamo il commit per determinare la fine della transazione e abbiamo l'abort nel caso in cui la transazione non va a buon fine. (Negli esercizi se non scrive niente consideriamo che committa sempre).

Tra le operazioni non consideriamo quelle più complesse come ad esempio la creazione di nuovi dati o l'inserimento in una lista, questo perchè mentre scrittura e lettura posso ripeterli quante volte voglio, l'inserimento in una lista non posso ripeterlo più di una volta altrimenti avrei dei dati duplicati.

**History:** dato un set di transazioni  $T = \{T_1, T_2, \dots\}$ , la history è un set ordinato di operazioni in cui c'è un interliving tra le operazioni che appartengono al set di transazioni  $T$ . In pratica la history è una unione degli insiemi  $T_1 \dots T_n$ .

Date varie transazioni possiamo avere varie possibili history.

La history inoltre mantiene l'ordine tra le operazioni della stessa transazione.

Nella history dell'esempio abbiamo tanti possibili conflitti.

Il conflitto tra due differenti transazioni, avviene in questi casi:

- Le due operazioni sono in conflitto se una è una operazione di **write** e l'altra è una operazione di **read** oppure se sono entrambi operazioni di write
- Le due operazioni lavorano sullo stesso dato, se hanno dati diversi su cui lavorare non sono in conflitto
- Il conflitto avviene tra operazioni che non commutano ovvero se cambio l'ordine di esecuzione delle operazioni, qualcosa potrebbe cambiare e il valore finale che verrà scritto tipicamente sarà differente rispetto all'ordine precedente.

Solitamente possiamo ridurre il numero dei tipi dei possibili conflitti a 3:

- **Dirty Reads**: questo problema si verifica quando faccio una lettura di un valore e una modifica in una transazione, poi leggo il valore dall'altra. La prima transazione però ha un abort quindi non va a buon fine quindi io sto usando nella seconda un valore che ho letto ma che in realtà non è valido.

T1: r[x=200] w[x:=100]	abort
T2: r[x=100] r[y=500]	<b>Dirty Read</b>

- **Lost Updates**: in questo caso abbiamo che due transazioni fanno la read e poi la write, quindi succede che l'aggiornamento che viene fatto dalla seconda transazione copre quello che è stato fatto dalla prima.

T1: r[x=100]	w[x:=600]	Lost Updates
T2: r[x=100]	w[x:=500]	

- **Unrepeatable Read**: in questo caso le modifiche di una transazione che viene eseguita in contemporanea rispetto alla mia vengono lette anche dalla mia transazione e questo non dovrebbe accadere perchè due transazioni che runnano in parallelo dovrebbero essere completamente isolate tra loro.

T1: r[x=100]	r[x=500]	Unrepeatable Read
T2: w[x:=500]		

Un esempio di History:

Time	T1	T2	T3
	r1[x]		
	w1[x]	r2[x]	
			r3[x]
			w3[x]
			c3
	w1[y]	w2[y]	
	c1		
		c2	

**H1**

Nell'esempio abbiamo che il tempo scorre verticalmente, poi in orizzontale abbiamo le varie transazioni con le operazioni che vengono svolte da ognuna di esse.

Nell'esempio la history è molto pericolosa perché abbiamo dei conflitti tra le varie operazioni, questa situazione viene creata da uno scheduler che dovrebbe decidere in che modo intervallare le varie operazioni.

Ad esempio possiamo vedere che abbiamo:

- Un conflitto tra w1[y] e w2[y] con un potenziale problema di tipo Lost Update
- Un conflitto tra w1[x] e r2[x] e anche con r3[x]

Immaginiamo questa situazione: abbiamo il DB e tre client che inviano richieste, lo scheduler deve decidere in che modo eseguire le operazioni e in quale ordine. Può eseguirle mantenendo l'ordine in cui vengono ricevute oppure in modo differente.

Di solito lo scheduler funziona in modo più complesso, quando arriva una richiesta da un client infatti non è detto che la richiesta venga subito accettata ed eseguita, per orchestrare l'esecuzione di operazioni da più client potrebbe prendere la richiesta da un client ma farlo attendere senza dargli subito l'ok per eseguire l'operazione successiva.

Alla fine comunque il client decide l'ordine delle operazioni all'interno

della transazione mentre lo scheduler decide come fare l'interliving delle varie operazioni mantenendo l'ordine.

**Una history è serializzabile se** la versione che otteniamo serializzando l'esecuzione delle varie transazioni mi porta in uno stato del database che è uguale a quello che avremmo eseguendo le varie transazioni in modo concorrente.

**Due history sono equivalenti** se sono definite sullo stesso set di transazioni e se producono lo stesso effetto nel database.

La definizione, presa in modo letterale, non ha senso in alcuni casi, ad esempio nel caso di transazioni di tipo analitico perchè in realtà il database non viene modificato, assume un minimo di senso se diciamo che alla fine la transazione viene anche scritta nel database.

Abbiamo definito già prima quando due transazioni sono in conflitto e abbiamo definito ora quando due history sono equivalenti.

Ora consideriamo la definizione di C-Equivalent (Conflict-Equivalent).

**Due history sono C-equivalent** quando:

- Sono definite sullo stesso set di transazioni (stiamo considerando due history che prendono le transazioni da uno stesso set e quindi avremo delle esecuzioni ordinate in modo differente delle stesse transazioni).
- Per ogni coppia di transazioni committate (le aborted non le consideriamo) e per ogni coppia di operazioni che sono in conflitto, allora abbiamo che in entrambe le history l'ordine delle operazioni in conflitto viene mantenuto

Questo vuol dire che se ho la history H e la history L allora quando faccio una read in H e in L leggo gli stessi dati e quando ho una operazione di scrittura in H e in L allora scrivo gli stessi dati.

Se abbiamo due history che sono C-Equivalenti allora sono anche equivalenti, il contrario NON VALE.

Un esempio:

T1	T2	T3	T1	T2	T3	T1	T2	T3
r1[x]	r2[x]			r2[x]		r1[x]	r2[x]	
w1[x]		r3[x] w3[x] c3	r1[x]	w2[y]		w1[x]		r3[x] w3[x] c3
	w2[y]		w1[x]		r3[x] w3[x] c3		w2[y]	
w1[y]			w1[y]		c3	w1[y]		
c1	c2		c1			c1	c2	
H1			H2 c-equivalent to H1?			H3 c-equivalent to H1 ?		
YES			NO					

Consideriamo la prima history H1 e la seconda H2 e le varie variabili che troviamo nella history. In questo caso le due history sono C-Equivalenti perchè i conflitti sono:

- Tra r2[x] e w1[x]
- Tra w2[y] e w1[y]
- Tra w1[x] e r3[x]
- Tra w3[x] e r1[x] e r2[x]

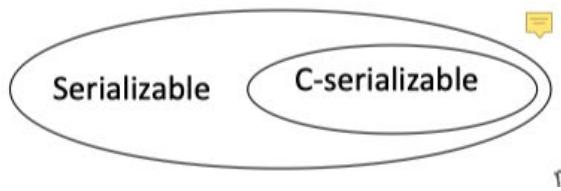
Se prendiamo la seconda History H2 vediamo che l'ordine dei conflitti viene mantenuto, abbiamo sempre che r2[x] è prima di w1[x], w2[y] è prima di w1[y], w1[x] è prima di r3[x] è dopo r1 e r2.

Se invece consideriamo la terza history, vediamo che è stato invertito l'ordine di w1[x] e di w2[x] e quindi non sono C-Equivalenti.

**Una history H su un set T di transazioni è serializzabile** se nel database ha lo stesso effetto dell'esecuzione seriale delle varie transazioni.

**Una history è C-Serializable** se è C-equivalent rispetto alla history seriale.

C-serializable implica che la history è anche serializzabile, però non è vero il contrario, ovvero se una history è serializzabile non posso dire che è anche C-serializable.



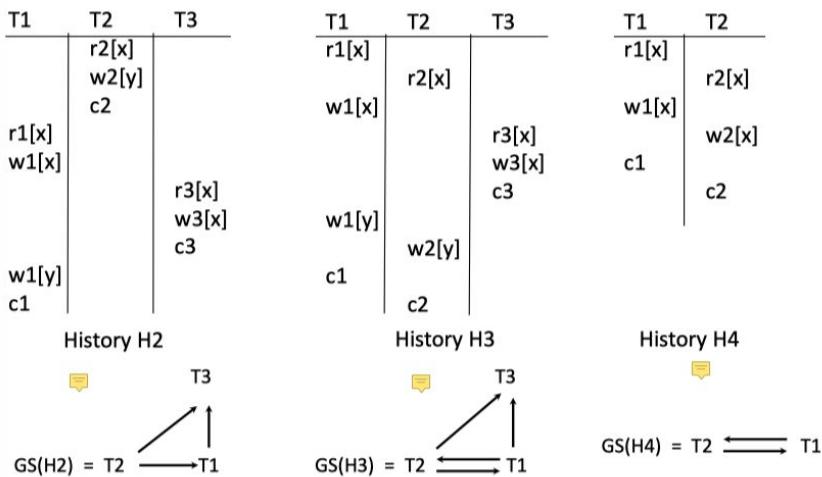
Ad esempio se consideriamo questa history, possiamo dire che è serializzabile perchè possiamo avere T1, T2, T3 eseguiti in modo seriale ma allo stesso tempo non è C-serializable perchè quando serializziamo modifichiamo l'ordine di W1 rispetto alle operazioni di W2 e quindi non è conflict equivalent. Il motivo per cui è serializzabile sta nel fatto che le variabili che modifico sono y e x. La y viene aggiornata solamente da T2 quindi non ci sono problemi, la X che viene aggiornata da tutti e tre potrebbe essere un problema, mettendo però t3 come transazione finale nella versione serializzata, risolviamo perchè abbiamo che il valore di X sarà quello aggiornato da T3, quindi comunque il più recente come nella versione non serializzata.

T1	T2	T3
r1[y]	w2[y] w2[x] c2	
w1[x]		w3[x] c3
c1		

### Serialization Graph

Per capire se una history è C-serializzabile dobbiamo controllare il suo serialization graph.

Il serialization graph viene definito in questo modo, data una history H che si basa sulle transazioni del set  $T = \{T_1, T_2, \dots\}$  il serialization graph è il grafo diretto che contiene un numero di nodi pari al numero delle transazioni che hanno avuto un commit, gli archi sono diretti e vanno dal nodo  $T_i$  al nodo  $T_j$  solamente se tra la transazione  $T_i$  e  $T_j$  c'è un conflitto (l'ordine del conflitto mi dice quale sarà il verso dell'arco nel grafo).



Nell'esempio vediamo che tra T2 e T1 c'è un conflitto così come tra T2 e T3, quindi mettiamo un arco, poi c'è un conflitto anche tra T1 e T3 quindi c'è un arco anche tra questi.

L'ordine dipende da quando compaiono le varie operazioni.

In questo caso abbiamo che la prima history è serializzabile perchè il grafo che esce fuori non contiene dei cicli, quindi possiamo subito dire che usando T2,T1,T3 ho una history C-serializable. La stessa cosa non posso dirla per gli altri due grafi che invece hanno un ciclo e quindi non sono C-serializzabili.

Dimostrazione:

Se abbiamo una history H che è C-serializzabile e creiamo una H<sub>s</sub> che è C-serializzabile e C-equivalente allora possiamo dire che se in H ci sono due transazioni che hanno un conflitto, ci sarà all'interno del grafo SG(H) un arco tra le due transazioni e in particolare possiamo dire che la transazione da cui parte l'arco è prima di quella in cui l'arco arriva. Non può esistere un ciclo all'interno di questo grafo perchè altrimenti avremmo che la transazione precederebbe se stessa.

Se abbiamo m transazioni in H e SG(H) è aciclico allora c'è un ordinamento topologico delle transazioni. Dato che il grafo è aciclico c'è almeno un nodo che non ha archi entranti, partiamo da quello e poi lo togliamo dal grafo, poi prendiamo un altro nodo che ora è diventato senza archi entranti e consideriamo quello come secondo nodo della history e così via.

## Strict 2PL protocol

Lo strict 2PL è un protocollo e non un algoritmo, questo comporta che le parti che lavorano in questo protocollo abbiano molta libertà, partendo dal protocollo possiamo vedere tanti algoritmi che usano questo protocollo.

Il protocollo è definito da tre regole, le prime due sono regole che troviamo in qualsiasi protocollo che usa la lock, la terza regola invece mi definisce lo strict 2PL protocol vero e proprio.

Definiamo per prima cosa le lock che possono essere utilizzate in questo protocollo:

- Abbiamo l'Exclusive Lock: questa lock viene data solamente ad una transazione che chiede una risorsa. Mentre quella transazione ha la lock le altre transazioni che la richiedono non possono ricevere quella stessa lock. Quindi due lock di tipo exclusive sono in conflitto tra loro.
- La Shareable Lock: in questo caso la lock viene data a più transazioni nello stesso momento (per la lettura), queste lock non sono in conflitto tra loro. La shareable lock è in conflitto con una lock di tipo exclusive.

Quando definisco vari tipi di lock posso definire anche una compatibility matrix che mi indica quali sono le lock compatibili tra di loro.

Questa matrice la consideriamo quando le richieste arrivano da due transazioni differenti tra loro e non quando arrivano dalla stessa transazione.

	S	X
S	Yes	No
X	No	No

Le regole del protocollo:

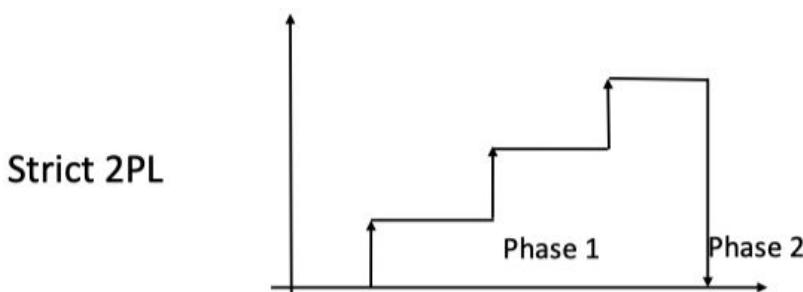
- Prima di lavorare su una risorsa, il client è obbligato a chiedere al transaction manager la lock corrispondente
  - Il transaction manager risponde alla richieste e non deve fornire due lock in conflitto tra loro nello stesso momento.
- Il fatto che il transaction manager sia libero di dare e prendere lock è una regola che dà molta libertà, lo scheduler potrebbe infatti essere del tutto unfair e quindi dare la lock sempre alla stessa transazione e non alle altre.

L'importante è che non fornisca delle lock che sono in conflitto tra loro.

Lo scheduler può gestire la transazione in tre modi differenti:

- Fornisce la lock al richiedente
- Mette il richiedente in uno stato di attesa
- Lo scheduler può dare il segnale di abort a chi ha chiesto la lock. Può bloccarla immediatamente oppure può metterlo in uno stato di attesa e poi mandare il segnale di abort.
- **L'ultima regola del protocollo dice che la transazione non deve rilasciare nessuna lock prima della terminazione.** Questo garantisce la serializzabilità mentre le prime due regole garantiscono la protezione delle singole risorse.

Questo garantisce un isolamento completo.

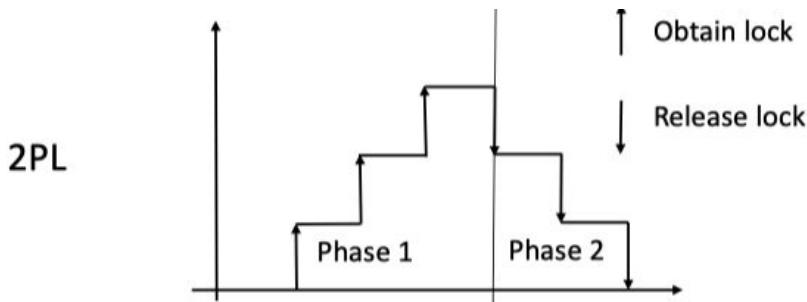


Viene chiamato 2 phase protocol perchè nella prima fase acquisisco le lock, poi ho una fase di releasing in cui rilascio e non prendo lock. Questo è diverso dal protocollo standard delle lock in cui faccio l'acquire e la release.

C'è una versione del protocollo che è meno stringente rispetto allo Strict 2PL, si tratta del 2 phase Lock.

Nel 2 phase protocol la terza regola dice che quando la transazione

inizia a rilasciare le lock, non prenderà altre lock in futuro. La prima lock che rilascio mi porta direttamente nella fase 2 e perdo la possibilità di acquisire la lock.



Nella versione strict invece rilascio tutte le lock nel momento del commit, non posso rilasciare le lock nel mezzo.

**Proprietà:** tutti i protocolli 2 phase garantiscono la serializability perchè se due transazioni hanno dei conflitti allora questi conflitti dovranno obbedire alle seguenti regole di precedenza:

- La seconda transazione deve entrare nella fase 1 soltanto dopo che la prima transazione sta nella fase 2.

**Un protocollo che rispetta la 2 phase rules crea una schedulazione serializzabile delle transazioni.**

Il protocollo 2 phase soffre del problema del dirty read bit perchè se inizio a rilasciare le lock prima di fare il commit allora qualcuno inizierà a vedere i miei risultati interni. Se dopo faccio un abort anche le transazioni che hanno letto i miei dati dovranno avere un abort altrimenti vuol dire che stanno lavorando con dati che non sono corretti.

Quindi prima di fare il commit di una transazione devo controllare che le transazioni che modificano dati che io leggo abbiano committato, se non hanno committato e c'è stato un abort, allora dovrò fare un abort anche io. Questo problema è chiamato Cascading Abort.

C'è anche il cascading commit per cui viene creato un link tra le varie transazioni che devono committare e si crea quindi una sequenza di transazioni che attendono un'altra per committare.

### Importante: 2 Phase Protocol vs 2 Lock Modes

Una domanda tipica dell'orale o degli esami è “Parlare del protocollo 2 phase”.

2 Phases è acquisizione e rilascio, 2 modes Lock è Shared e Exclusive, le modes possono anche essere più di 2 o anche di meno, le fasi sono comunque 2.

La vera regola che mi caratterizza un 2 Phase Protocol è la regola 3, tutta la teoria sul funzionamento delle lock e sul fatto del conflitto tra le varie lock non c'è necessità di descriverla nel dettaglio.

Il 2 Phase Lock è un protocollo dove dopo che una transazione lascia una lock non può prenderne un'altra, possiamo anche parlare della versione Strict in cui indichiamo che la lock non viene rilasciata fino alla fine e poi vengono rilasciate tutte insieme.

Senza il punto 3 non ho la serializability e questa è principalmente il mio obiettivo, quindi è importante scriverlo.

---

### **Implementazione standard del protocollo 2 Phase Lock:**

Lo scheduler tiene traccia all'interno di un set di tutti i dati riguardanti le lock memorizzate in triple del tipo (T, mode, x).

Quando una transazione chiede una lock in una pagina lo scheduler ha due possibilità:

- Se la regola 2 ci fornisce l'opportunità di utilizzare la lock allora lo scheduler fornisce la lock al richiedente
- Se invece non è possibile perchè un'altra transazione ha già preso la lock per usare quella risorsa allora la richiesta viene sospesa.  
(Per sospornerla ci stanno vari modi). Quando il chiamante viene sospeso, lo scheduler ricorda che c'è stata questa richiesta all'interno di una coda d'attesa. Lo scheduler ha varie code d'attesa per le varie risorse che ha a disposizione.

Ci stanno infinite variazioni per questo algoritmo, una prima variazione, posso fare dei check per vedere chi risvegliare dopo che una lock viene rilasciata, se ad esempio ne ho alcuni che richiedono una lock in modo

shared e uno che la richiede esclusiva allora permetto a quelle shared di lavorare. Un'altra variazione, invece di svegliare tutti in un ordine random potrei scrivere una politica che mi indica in che modo vengono risvegliati i thread in attesa.

Potrei avere anche una priorità per le varie transazioni in base all'importanza di quella transazione per me.

Possiamo dire che:

- Se non ci sono lock in conflitto
- Se tutte le transazioni non rilasciano la lock prima del committ

Allora tutte le possibili schedulazioni che vengono prodotte sono serializzabili nel commit order.

Se una transazione T1 committa prima di T2 allora se c'è un conflitto tra una operazione di T1 e di T2 allora l'ordine di queste operazioni è prima tutte le operazioni di T1 e dopo tutte quelle di T2.

Ogni schedulazione che viene prodotta utilizzando Strict 2PL è anche C-Serializable ma non vale il contrario.

Un esempio:

No locks			S2PL scheduler			S2PL history		
t1	t2	t3	t1	t2	t3	t1	t2	t3
r1[x] w1[x]			r1[x], r1[x] wl[x], w1[x]			r1[x]		
	r2[x] w2[x]			rl[x]*		rl[y], r3[y]		
w1[y] c1	c2	r3[y]	wl[y]*			c3, u[y]		
			wl[y], w1[y] c1, u[x, y]				r2[x]	
				rl[x], r2[x] wl[x], w2[x] c2, u[x]			w2[x]	
							c2	
								c3

SG = t3 → t1 → t2

Nell'esempio vediamo che la T1 prende una lock su X e poi T3 prende la lock su Y e committa, a questo punto T1 prende la lock su Y e committa. Solo in questo momento può iniziare a lavorare T2.

La history che viene generalizzata è serializzabile e otteniamo T3->T1->T2.

Se ad esempio consideriamo:

T1	T2	T3
r1[x]		
w1[x]	r2[x]	
	w2[x]	r3[y]
w1[y]		
c1	c2	c3

Vediamo che non è possibile generare questo con uno scheduler Strict 2 phase Lock perchè ci troviamo ad avere un rilascio della lock su X prima che la transazione T1 committi.

## Deadlock

Le deadlock sono un problema grave in un database, quando utilizziamo un protocollo che è in grado di chiedere due lock può capitare di avere un conflitto tra le lock e quindi una deadlock.

Per le politiche del 2 Phase Lock ogni transazione può chiedere varie risorse in ordine del tutto random, quindi la probabilità di una deadlock è molto alta e non possiamo tollerare un DBMS dove avviene una deadlock e non si fa niente per evitarlo.

La situazione è molto complicata perchè spesso succede che viene chiesta la lock su tante risorse differenti tra loro e non soltanto tra due risorse diverse.

Abbiamo due famiglie di approcci per cercare di risolvere questo problema, entrambe però non risolvono totalmente:

- Deadlock Detection: dato che la deadlock si presenta quando abbiamo un ciclo nel grafo, l'idea è di cercare di trovare la situazione di deadlock studiando il wait-for Graph.

Con Wait-For graph intendiamo il grafo dove i vertici sono le transazioni attualmente attive e gli archi sono tali che T1 è collegato a T2 se T2 sta usando una risorsa che vorrebbe usare anche T1 e per cui è in attesa.

Quando una transazione chiede una lock e viene messa in attesa allora un nuovo arco viene aggiunto nel grafo, quando invece una transazione ottiene la lock allora rimuoviamo l'arco.

Quando aggiungo il nuovo arco devo controllare se il nuovo arco genera un ciclo, in questo caso si genera una deadlock e scelgo le transazioni che devono essere abortite e le blocco per sbloccare la deadlock.

Tipicamente le transazioni che vengono bloccate non vengono subito fatte ripartire quindi non si ripresenta subito il problema con il ciclo, dopo un po' però dobbiamo farle ripartire.

Questa soluzione funziona molto bene e risolve il problema ma è molto costosa perchè mantenere un grafo di questo genere ha un costo alto sia per lo spazio necessario sia per il tempo. Con questo approccio rischiamo di avere un sistema che spende più per fare il controllo che per altro.

Una ottimizzazione può rendere migliore questo algoritmo, dobbiamo fare in modo di ritardare il controllo della presenza di eventuali cicli, quindi faremo magari un unico controllo ogni 100 archi inseriti oppure ogni X minuti oppure quando ci sono delle lock che stanno attendendo tanto.

Il costo dell'inserzione nel grafo lo paghiamo ma non paghiamo il

costo dei continui controlli.

Una soluzione alternativa per la detection è chiamata “Timeout Solution”, ogni volta che una transazione attende più di un certo tempo per prendere la lock, si decide di bloccare una delle transazioni, o quella che ha fatto richiesta o quella che tiene la lock. Si usa una politica per decidere quale delle due bloccare, si può privilegiare la più vecchia per evitare la starvation oppure anche una politica che privilegia la più recente. Un'altra possibile politica consiste nel contare il numero di lock che le transazioni hanno oppure nel considerare il lavoro che è stato fatto dalle transazioni.

Il problema di questa politica sta nel modo in cui scelgo il timeout, non c'è una scelta sensata, come lo scelgo? Potrebbe essere troppo corto per capire se una transazione andrà avanti o no oppure troppo lungo. Una stima basata sulla statistica vedendo quanto tempo ci vuole per completare solitamente una transazione potrebbe andare bene. Il problema in questo caso è che il sistema non è uniforme, nel senso che il cliente lo può usare in varie parti della giornata quindi ci sono magari dei momenti della giornata in cui il sistema è rallentato perché lo usano in molti e quindi ci metto 20 secondi a finire il lavoro che voglio fare. Il problema è che se io ho settato un timeout di 10 secondi mi trovo a killare ogni transazione e questo non va bene.

- Deadlock Prevention: si tratta di un approccio differente rispetto al deadlock detection, in questo caso vogliamo proprio evitare che avvenga una situazione di deadlock.

In questo caso quando una transazione viene avviata, gli viene assegnato un timestamp in modo che si possa avere un'idea di transazione più giovane (younger) e più vecchia (older).

Per evitare la formazione di cicli nel grafo viene implementata una politica che fa in modo che quando  $T_i$  aspetta  $T_j$  allora  $T_i$  è older rispetto a  $T_j$  oppure  $T_i$  è younger di  $T_j$ .

Quindi o permettiamo che una transazione attenda una transazione già attiva ma più vecchia di lei oppure una attiva e più giovane di lei.

In questo modo evitiamo la formazione di cicli all'interno del grafo, per evitare la formazione di cicli all'interno del grafo potrebbe anche essere sufficiente assegnare un numero alle varie transazioni del grafo in modo che uno aspetta se l'altro ha un numero maggiore, questo è abbastanza per evitare la nascita di cicli nel grafo.

Le due politiche si chiamano Wait-Die e Wound-Wait e si comportano nello stesso modo per quel che riguarda l'abort della transazione mentre sono differenti per quel che riguarda l'attesa. Entrambi sono concordi sul fatto che quando devo uccidere una transazione, dovrò uccidere quella più giovane perché questo mi evita la starvation (viene sempre bloccata la stessa) perché la transazione younger verrà poi riavviata con lo stesso timestamp e quindi prima o poi non sarà più la younger ma sarà la older.

Nella politica Wait-Die la transazione older attende la terminazione della transazione younger ma non viceversa e infatti quando una younger chiede la lock che è in possesso di una transazione older allora la younger fa un abort e si suicida.

**IF**  $ts(T_i) < ts(T_j)$       ( $T_i$  is older than  $T_j$ )

**THEN**  $T_i$  wait for  $T_j$     (older waits for the younger)

**ELSE**  $T_i$  aborts      (**younger dies**) 

Nella politica Wound-Wait invece se arriva una transazione older e chiede la lock che è in possesso di una transazione younger, allora uccide la younger e prende la lock, se invece arriva una transazione younger e la lock ce l'ha una older allora la younger attende.

**IF**  $ts(T_i) < ts(T_j)$       ( $T_i$  is older than  $T_j$ )

**THEN**  $T_i$  wounds  $T_j$  and takes the lock    (younger dies: lock to older)

**ELSE**  $T_i$  waits      (younger waits for older)

La transazione che muore è sempre la più giovane e mai la vecchia, questo mi garantisce che non ci sarà starvation.  
 Per evitare la formazione di cicli ho la regola che mi manda gli archi del wait-graph sempre nella stessa direzione, o dai nodi older ai nodi younger o viceversa, in questo modo non abbiamo la possibilità che si formino dei cicli.

Possiamo generare tantissimi protocolli differenti basandoci su due idee:

- 1) l'ordine degli archi nel wait-for graph per evitare la formazione di cicli. Quando abbiamo un ordine topologico non abbiamo un ciclo nel grafo, ne possiamo scegliere uno qualsiasi per la nostra politica.
- 2) Cosa faccio quando un arco viola la regola del primo punto? Qua abbiamo detto che il younger muore ma un'altra possibilità può essere che chi tiene la lock sopravvive e muore chi l'ha chiesta. Oppure possiamo far morire la transazione che fino ad ora ha lavorato di meno.

Se imponiamo una ordine topologico random nel grafo allora vuol dire che il 50% degli archi viola la nostra regola, quindi vuol dire che il 50% delle wait for situation richiede che una delle due transazioni muoia, non vuol dire che il 50% delle lock portano ad un abort.

Queste random abortion sono utili o no? Se ho molti errori nel grafo allora le abort sono molte, altrimenti sono poche.

Questo approccio è molto semplice da implementare rispetto al waits-for-graph, non richiede strutture dati, il costo è basso ma possiamo avere un numero alto di abortion (nel wait-for-graph praticamente avevamo un abort solamente in caso di deadlock) e questo è un problema.

Quindi questa non è una soluzione accettabile per il problema del deadlock.

## **Serializability senza Locking**

Negli approcci visti fino ad ora abbiamo un comportamento **pessimistico** perchè vogliamo evitare che il problema del deadlock accada e quindi cerchiamo di bloccare le possibili cause prima che queste causino effettivamente un errore.

C'è però un approccio differente che è **ottimistico** e si comporta in modo del tutto differente, in questo caso permettiamo alle transazioni di fare quello che vogliono, quindi sia di scrivere che di leggere fino al momento in cui non voglio fare il commit.

Solamente nel momento del commit si controlla cosa è stato fatto dalla transazione e se ci sono delle operazioni che sono in conflitto con le altre transazioni che sono state eseguite in parallelo allora la transazione che ha committato per seconda viene bloccata.

Vantaggi:

- Dato che attendiamo fino alla fine di una transazione, permettiamo che questa svolga un numero maggiore di operazioni e quindi noi abbiamo più informazioni per comprendere se ci sono conflitti. Questo protocollo permette di alzare il throughput perchè molte transazioni lavorano in parallelo.
- Dato che il commit è lento, è preferibile avere un overhead solamente nel momento in cui committiamo e non avere un overhead per ogni operazione che viene svolta.

Svantaggi:

- Lo svantaggio è che se la transazione viene bloccata allora deve annullare tutte le operazioni che ha effettuato.

L'approccio ottimistico lo possiamo preferire quando abbiamo poche transazioni che devono essere eseguite, questo perchè riusciamo a mandarle in parallelo ottenendo un grado di parallelismo più alto.

L'approccio pessimistico invece è da preferire quando abbiamo tante transazioni che vogliono lavorare tutte insieme.

Ci sono tanti protocolli che usano l'idea “Optimistic”, uno di questi è lo snapshot isolation:

- Ci focalizziamo sull'isolation e non sulla serializability

- Quando una transazione viene avviata, il sistema crea uno snapshot, in questo modo la transazione leggerà solamente dal suo snapshot e scriverà solamente nel suo snapshot in modo che le altre transazioni non verranno influenzate dalle possibili scritture che vengono effettuate.
- Quando faccio il commit prendo il mio snapshot e il sistema controllerà se va in conflitto con le altre transazioni che sono state avviate nel frattempo oppure no. Se non ci sono conflitti allora posso scrivere sul disco le modifiche che ho effettuato al mio snapshot ma le altre transazioni già avviate non vedranno le mie modifiche perché continueranno ad usare il loro snapshot. Se invece il sistema dice che non posso scrivere sul disco dovrò annullare la mia transazione.

Il check che viene fatto dal sistema controlla che durante l'esecuzione della mia transazione non siano state effettuate delle modifiche alle pagine che ho letto anche io, se sono state fatte modifiche la mia transazione viene abortita, altrimenti posso scrivere sul disco. Se invece qualcuno legge la stessa pagina nel momento in cui la leggo anche io non ci sono problemi e nemmeno se uno la legge e uno la scrive, gli unici conflitti che mi interessano sono Write-Write.

T1	T2	T3	T1	T2	T3
Snapshot(T2) x = y = z = 0	begin w[y:=1] c			begin w[y:=1] c	
	begin r[x=0] r[y=1]			begin r[x=0] r[y=1]	
Snapshot(T1) x = z = 0 y = 1				begin w[x:=2] w[z:=3] c	
Snapshot(T3) x = z = 0 y = 1	r[z=0] c			r[z=0] w[x:=3] c <-- ?	
					abort

Nella prima history committano tutte perché non ci sono dei conflitti di tipo write-write tra le varie operazioni, i conflitti di tipo read-write non ci

interessano in questo caso. Qui il serializability graph ha un ciclo per la read-write ma in realtà non ho problemi nella history e quindi non ho problemi con la serializability anche se ho un ciclo nel grafo.

Quindi qua il serializability problem non esiste.

Nella seconda history invece abbiamo una situazione di abort perchè T1 vuole modificare X che però è stato modificato già da T3. In questo caso abbiamo un conflitto di tipo write-write e quindi questo è un problema che mi porta ad avere un abort della transazione T1 perchè T3 è stata la prima a committare.

Proprietà interessante dello snapshot isolation: non abbiamo nessuna forma di read lock perchè la lettura non causa problemi e possiamo leggere senza usare le lock. Questo è utile per transazioni che devono fare analisi sui dati, in questo caso devo leggere tutti i dati che ho e fare magari dei conti, queste transazioni in un sistema standard prenderebbe la lock su tutto il sistema. Quindi prendere la read-lock con transazioni che svolgono analisi sui dati può essere molto pericoloso e costoso, quindi l'idea che le letture vengono fatte senza usare la lock è vincente nel caso dello snapshot isolation.

Inoltre riusciamo ad evitare le anomalie come ad esempio lost update, dirty read e non repeatable read.

Sfortunatamente abbiamo comunque un problema e non abbiamo la serializability (però garantisce isolation).

Assumiamo di avere due transazioni che vogliono sincronizzarsi, una aggiorna un valore di X e una aggiorna Y, l'obiettivo è fare in modo che i valori di X e di Y vengano invertiti.

Se le due transazioni partono in parallelo usando lo snapshot isolation abbiamo che le due transazioni leggono i valori iniziali di X e di Y e li invertono, quindi alla fine avremo il risultato che aspettiamo.

Nessuna soluzione seriale però ci porta ad avere lo stesso risultato perchè quando faccio prima T1 allora avrò che viene letta la Y iniziale e viene assegnata a X, poi leggo la X che però è stata aggiornata quindi leggo la nuova X e non la vecchia.

T1 (x:=y)	T2 (y:= x)
begin	begin
r[y= ?]	r[x= ?]
w[x :=y]	w[y :=x]
c	c

Anche Oracle utilizza lo snapshot isolation, però sono state effettuate delle modifiche per evitare dei problemi di serializability.

## Concorrenza nei sistemi reali

Non c'è un modo ottimo per risolvere le deadlock, abbiamo visto la deadlock prevention e la deadlock detection.

Queste soluzioni non sono davvero soddisfacenti perchè ad esempio con la deadlock prevention abbiamo un alto numero di situazioni di abort e questo non va bene. Una soluzione alternativa è quella degli snapshot in cui non abbiamo le lock, il problema in questo caso è che è difficile creare lo snapshot di tutto il database, inoltre questo metodo garantisce l'isolation ma non garantisce la serializability e quindi non è facile capire se c'è stato un problema.

Nei sistemi reali ci sono altre complicazioni e riguardano il problema della granularity inoltre ci sono anche operazioni come l'inserzione e la rimozione dal database che non sono state considerate in precedenza perchè consideriamo solamente operazioni di scrittura. Inoltre ci sono problemi con l'aggiornamento degli indici.

Granularity Problem: consiste nel problema di decidere come mettere le lock nel database. Nel database abbiamo una gerarchia, possiamo mettere la lock su tutto il database o ad esempio solamente su un campo di un record.

Se mettiamo la lock su un oggetto molto grande abbiamo un'alta granularità, se invece mettiamo la lock su un oggetto piccolo abbiamo una granularità bassa.

Avere una granularità bassa permette di aumentare la concorrenza ma abbiamo anche un overhead maggiore perchè si creano più lock e poi abbiamo un'alta probabilità di deadlock.

Se invece abbiamo una granularità alta abbiamo meno concorrenza ma anche meno overhead e meno probabilità di avere una deadlock.

Non abbiamo una risposta precisa e ottima che mi indica il modo migliore per mettere le lock nel database.

Una possibile soluzione è il Multiple Granularity Locking, in questo caso per ogni transazione abbiamo due possibilità, o mettiamo la lock su tutto il container (quindi abbiamo una lock shared o exclusive su tutta la relazione che vogliamo usare) oppure mettiamo una intention lock su tutta la relazione e poi una lock sul singolo oggetto. Soltanto dopo che ho preso l'intention lock su tutta la relazione posso prendere la lock sul singolo oggetto.

Quindi su un container posso chiedere 5 tipi diversi di lock:

- S Lock
- X Lock
- IS: intention share lock ci permette di chiedere poi una shared lock su una parte dell'oggetto
- IX: permette ad uno di chiedere poi una lock esclusiva su una parte dell'oggetto
- SIX: questa è l'unica che combina due tipi di lock, nessuna delle due è più grande dell'altra, S è più potente perchè è più grande ma è meno potente perchè è shared, IX è più potente perchè è esclusiva ma meno potente perchè è intention. Non posso creare combinazioni di altri tipi di lock perchè abbiamo sempre una lock più grande e una più piccola, ad esempio S e IS è inutile metterle insieme perchè S include già IS.

Posso creare anche la compatibility Matrix:

- IS e IS lo posso permettere perchè le lock sono shared e perchè sono anche intention.
- IX e IS lo posso permettere perchè do ad una transazione la possibilità di prendere un record come shared, poi l'altra transazione prende l'intention lock e poi prenderà un record come exclusive. Se vuole prendere lo stesso record preso dalla IS allora deve attendere.
- S e IS la permetto perchè S contiene anche IS
- IS e SIX la permetto perchè IS è yes e IS e IX è yes
- X e IS non la permetto perchè X è esclusiva. Per prendere la lock esclusiva devo attendere che IS rilasci la lock.
- IX e IX la permetto perchè probabilmente andranno in record differenti
- S e IX non la permetto perchè se qualcuno modifica un record non posso avere il permesso di leggere tutto
- SIX e IX è no perchè è NO anche IX e S
- SIX con SIX è no perchè è no anche S con IX

Questa tabella fa quello che vogliamo perchè permette a due transazioni che vogliono la lock su tutta la tabella di prendere la lock e lavorare in contemporanea.

Questa tabella ci permette anche una coesistenza tra transazioni che vogliono le lock su tutta la tabella e transazioni che vogliono una lock solamente su un record.

Questo approccio però può causare dei problemi, abbiamo molto overhead perchè ora le transazioni hanno bisogno di due lock, una a livello della tabella e una a livello del record.

Questo in alcuni casi può non essere un grande problema perchè se abbiamo una transazione che legge tanti record ci troveremo ad avere una sola lock su tutta la tabella e non una lock per ogni record.

Il secondo problema è decidere la granularity della lock perchè non vogliamo che sia il programmatore a scegliere la lock granularity, ci sono due scelte possibili, può essere scelto a runtime oppure a tempo di compilazione. Tutti e due gli approcci vengono utilizzati.

Abbiamo un problema se consideriamo l'inserzione dei record:

- Una transazione vuole calcolare la media dell'attributo salary, quindi scorre la tabella e somma tutti i salary
- Una transazione prende la lock IX e inserisce un nuovo professore
- La prima transazione ha un problema perché quando conta il numero di righe ha un numero in più e quindi quando fa la media sarà errata

Quindi una possibile soluzione per il problema sta nel modificare un po' le strutture dati che vengono utilizzate mettendo ad esempio una shared lock sull'end of file della tabella che stiamo utilizzando.

Una possibilità, completamente differente, consiste nello spostarsi dalle lock all'utilizzo di predicati, questo è un approccio molto più complicato.

Un altro problema: concorrenza negli indici.

Se adotto l'idea che quando inserisco un record devo prendere la lock su tutto il file ho lo stesso problema anche nell'indice.

Normalmente i vari sistemi usano un protocollo di lock sugli indici che è stato appositamente pensato per gli indici, la presenza degli indici però causa comunque dei problemi con la concorrenza. Quando devo aggiornare un indice metterò la lock sull'indice e questo aumenta la possibilità di avere una deadlock.

## Lezione 11: Physical Design

La realizzazione di un database relazionale procede in 4 fasi:

- Nella prima fase si analizzano le richieste del cliente
- Poi si fa uno schema concettuale in cui rappresentiamo le varie entità
- Poi si fa uno schema logico che deve essere anche normalizzato
- Poi c'è la fase di Physical Design in cui dobbiamo definire delle strutture dati appropriate per quello specifico utilizzo del DBMS

La fase di Physical Design è guidata dallo studio delle statistiche che riguardano il database e dal workload.

Con statistiche intendiamo il numero di relazioni e record che dobbiamo inserire all'interno del database, per ogni record poi dobbiamo capire anche quanti attributi avremo e la dimensione dei vari attributi.

Con workload invece intendiamo il set di tutte le possibili query che possono essere effettuate all'interno del database.

Alcune di queste operazioni che stanno nel workload sono definite critical perchè sono quelle che verranno eseguite più frequentemente e quindi che dovranno essere eseguite più velocemente rispetto alle altre query.

Idealmente dovremmo capire quante volte ogni query dovrà essere eseguita, quali parametri utilizzerà e il selectivity factor della condizione. Oltre alle critical query dobbiamo considerare anche i critical update, dobbiamo capire in particolare il tipo dell'update, quali attributi saranno modificati da un update, il selectivity factor delle condizioni e quali attributi sono usati nelle condizioni.

Il tuning del database dovrà essere effettuato tenendo a mente le critical operations perchè queste saranno quelle che verranno svolte più di frequente.

Per studiare la frequenza delle critical operations si utilizza una tabella ISUD in cui si analizza il costo delle operazioni e si vede quanti dati saranno modificati e con quale frequenza.

La prima scelta che possiamo fare e che mi può modificare la rapidità delle critical query riguarda la struttura dati che possiamo utilizzare per memorizzare le relazioni.

Ad esempio possiamo utilizzare lo heap quando l'operazione di scan è più importante delle altre operazioni, se invece i dati sono ordinati in modo sequenziale allora usiamo la static sequential organization.

Se facciamo spesso delle ricerche su delle chiavi, ci converrà avere una organizzazione con l'hash, se invece facciamo una range query e vogliamo i dati ordinati sia in base alla chiave sia in base ad un altro attributo potremmo scegliere una tree organization.

Consideriamo la scelta dell'indice. Quando dobbiamo svolgere le query consideriamo le critical query e cerchiamo di capire in quali occasioni sarebbe interessante ed utile avere un indice.

Ci sono alcune occasioni in cui l'indice potrebbe solamente essere una complicazione, in generale l'indice mi migliora la velocità con cui viene svolta la query però peggiora la velocità nel momento in cui devo aggiornare dei dati all'interno del database.

In generale è meglio evitare di usare l'indice se:

- Abbiamo una relazione di poche pagine
- Gli attributi sono modificati spesso
- Le query non sono molto selettive e restituiscono molti dati

Invece è sempre utile avere un indice sulla primary key e sulla foreign key.

Se abbiamo un indice sulla chiave primaria, questo indice permette di avere sia una ricerca rapida sia anche un inserimento rapido perchè altrimenti per capire se quella chiave è già presente dovrei effettuare tutta la scansione della tabella.

Se abbiamo un indice sulla foreign key questo mi permette di velocizzare l'eliminazione dei dati. Ad esempio se abbiamo una tabella studenti e una tabella esami, per ogni esame abbiamo uno studente. Se abbiamo un indice anche sulla foreign key di esami questo mi permette, nel momento in cui elimino uno studente, di eliminare velocemente tutti i dati relativi a quello studente presenti nella tabella esami.

## Quando usare un indice:

- In alcuni casi può essere utile usare un indice se questo ci permette di utilizzare un IndexOnly Plan. Quando abbiamo una query che usa solamente uno o due attributi, se la query è crucial conviene creare un indice che ha al suo interno gli attributi che vengono usati in questa query.  
Quindi un piano di accesso che ci fa accedere solamente all'indice. È utile anche per i MergeJoin e per gli IndexNestedLoop
- Se valutando la condizione vediamo che abbiamo un OR non ci conviene più utilizzare un indice
- Valutando la condizione possiamo vedere che in caso di equality search può tornare utile un hash index mentre in caso di range search un Btree Index.
- Se abbiamo poi una condizione multi attribute (congiunzione di condizioni che da sole non sono abbastanza selettive per creare un indice ma se le consideriamo tutte insieme la query è abbastanza selettiva e conviene avere un indice) allora potremmo creare un indice su più attributi in modo da fare un indice composto. Un indice composto è utile anche se abbiamo due query sulla stessa tabella che usano entrambi un attributo e poi ne usano un altro che cambia nelle due query. In questo caso conviene fare un unico indice con i tre attributi
- In alcuni casi possiamo eliminare un indice se questo è sussunto da un altro indice, ad esempio l'indice su A non è necessario se abbiamo già un indice su A e su B.
- Un indice può anche essere utile se viene utilizzato per query dove abbiamo un Order By, un group By o un distinct.

Per ottimizzare un database e scegliere bene gli indici abbiamo due possibilità, una visione locale e una globale.

Nel caso della visione locale partiamo dalla query che viene svolta maggiormente e poi passiamo a tutte le altre (questo è quello che si fa in pratica).

Nel caso della visione globale invece ottimizziamo l'indice rispetto al workload ovvero consideriamo tutto il workload e capiamo subito se un

indice è utile o no, non solo valutiamo i benefici per la singola query ma anche i benefici rispetto a tutte le altre, in particolare eliminiamo gli indici che non servono e mergiamo quando vediamo che sono simili tra loro. Alla fine si sceglie la configurazione ottima che soddisfa nel modo migliore tutte le query.

In teoria è meglio ottimizzare rispetto al workload e non rispetto alla singola query perchè capita spesso che creo due indici per due query ma poi si vede che in realtà ne basterebbe uno solo perchè un indice sussume l'altro.

Se poi abbiamo tanti dati da inserire nel database allora potremo anche creare gli indici solamente dopo aver caricato tutti i dati all'interno del database.

Esempi di utilizzo di indici:

```
SELECT Name  
FROM Lecturers  
WHERE Position = 'P'  
AND Salary BETWEEN 50 AND 60
```

In questo primo esempio conviene avere un unico combined index su Position e su Salary. In questo modo troviamo subito la rid list dei record che soddisfano le due condizioni.

Se usassimo due indici il costo per l'accesso all'indice sarebbe maggiore perchè accediamo due indici mentre invece il costo per l'accesso ai dati è uguale.

L'idea di avere solamente un indice su position è completamente sbagliato perchè la condizione Position non è abbastanza selettiva. Questa è una visione locale appositamente pensata per questa query, con una visione globale potrei avere invece una query che è più critica e magari potrei non voler usare un indice organizzato in questo modo perchè l'altra query sarebbe più lenta.

```
SELECT ResearchArea,Position
      COUNT(*)
  FROM Lecturers
 GROUP BY Position, ResearchArea
 ORDER BY ResearchArea
```

L'indice in questo caso sarebbe utile nella tabella Lectures?

Possiamo avere un indice su entrambi gli attributi e questo ci permette di velocizzare il group by e l'order by.

Fare la scansione di un indice senza condizione per avere i dati ordinati potrebbe non essere sempre una buona idea perchè c'è sempre il costo per prendere i vari record mentre invece il table scan seguito dal sort potrebbe essere più veloce perchè dipende dal numero di pagine.

In questo caso invece è meglio avere un indice perchè dobbiamo restituire gli attributi che abbiamo nel groupBy quindi possiamo creare un indice con entrambi e mi basta scorrere solamente l'indice, quindi è una query di tipo Index Only per cui conviene avere un indice.

```
SELECT Name
      FROM Lecturers
 WHERE Salary > 70
```

Il clustered index è sempre più veloce del table scan perchè il costo è  $N_{pag} * S_f$  perchè tutti i dati sono uno vicino all'altro. In questo caso un indice standard non sarebbe utile perchè abbiamo una condizione che è poco selettiva ma invece un clustered index è utile.

Il clustered index in realtà è una primary organization quindi sarebbe meglio aver pensato al suo utilizzo già dall'inizio e non solo in questo momento.

I clustered index sono utili sia quando vogliamo utilizzarli con una condizione che non è molto selettiva sia quando vogliamo ordinare i dati ma dobbiamo utilizzarli con molta attenzione perchè ne possiamo avere solamente uno e quando l'abbiamo messo tutti gli altri indici sono poi secondary organizations.

```
SELECT DISTINCT FkDepartment
FROM Lecturers ;
```

```
SELECT Salary, COUNT(*)
FROM Lecturers
GROUP BY Salary;
```

In questi due casi mi serve avere un indice sulla foreign key o su salary in modo che la query la riduco ad una operazione Index Only, in questo caso non ha senso utilizzare un clustered index.

Gli index only Plan sono utili nelle Join (dove la join può anche essere una semi join che vengono usate per filtrare). In questo caso abbiamo una situazione ideale per gli index only plan perchè l'unica cosa che ci interessa delle due tabelle è il FKDepartment e il PkDepartment.

Posso usare un index only plan anche quando abbiamo più di un attributo, avremmo un combined index che funziona come quello appena descritto ma ho anche un secondo attributo che mi permette di avere comunque una operazione Index Only.

Quindi, concludendo la parte relativa agli indici:

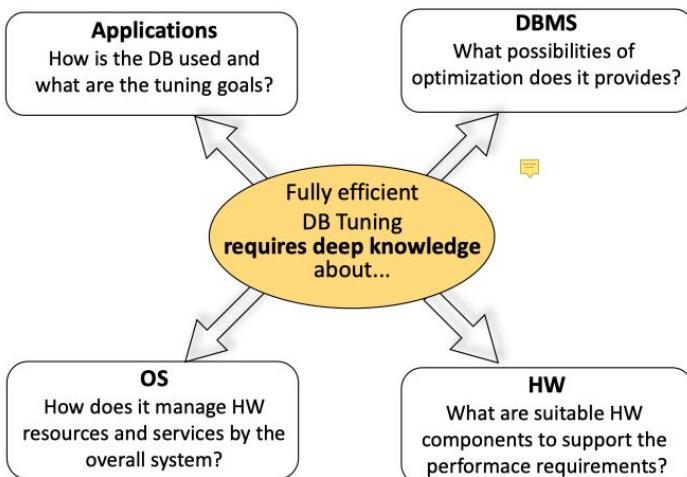
- Non si deve mai esagerare con gli indici perchè hanno un costo alto quando aggiorniamo i nostri dati
- Dobbiamo sempre avere una visione globale per capire se un indice è utile o meno
- I primary index sono molto utili ma non posso metterne più di uno.

## Database Tuning

Vogliamo migliorare le performance del database in modo da avere delle query che vengono eseguite più velocemente.

Il tuning di un database è un procedimento che coinvolge varie aree e quindi chi se ne occupa deve avere una conoscenza approfondita di varie parti del sistema che stiamo utilizzando.

Possiamo fare un tuning a livello di applicazioni, a livello di DMBS, a livello di sistema operativo o di hardware.



Il tuning quindi è una questione di conoscenza, se vogliamo migliorare la parte di applications dobbiamo capire l'obiettivo del cliente e dobbiamo studiare il workload per capire quali sono le query che verranno usate maggiormente.

Se vogliamo fare un tuning del DBMS dobbiamo capire come funziona, quali sono i parametri che utilizza e le possibili ottimizzazioni.

Se vogliamo modificare l'OS per cercare di migliorare il funzionamento del nostro database dobbiamo avere una conoscenza del sistema operativo. Possiamo poi modificare l'hardware e in questo caso dobbiamo capire quali modifiche possono portare miglioramenti al database.

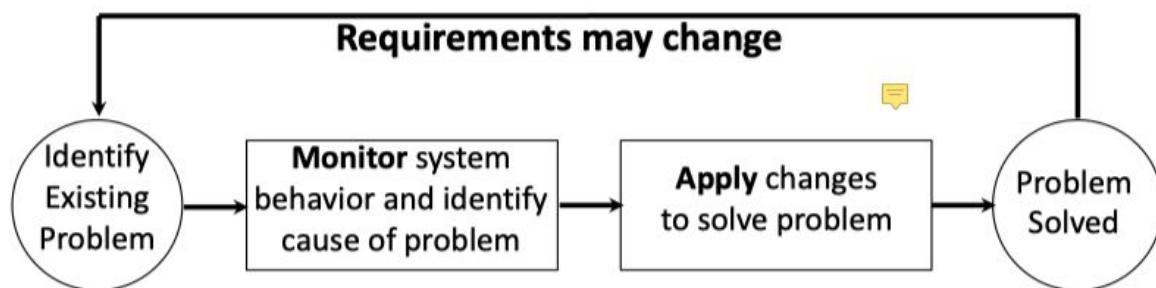
Il processo di tuning non viene svolto da una singola persona, ci sono vari specialisti che se ne occupano e in vari differenti momenti durante il processo di creazione del database.

In particolare abbiamo:

- DB and Application designer: ha conoscenza del DB e delle application, effettua il tuning durante lo sviluppo del database
- DB Administrator: ha conoscenza di DBMS, OS e HW, in alcuni casi può conoscere anche l'applicazione. Il tuning viene eseguito quando il DB è già in funzione e quindi vengono modificati parametri per cercare di adattarsi alle richieste
- DB Expert: ha conoscenza di DBMS, OS e HW e si occupa del tuning quando ci sono problemi al db

Il processo di database Tuning è un processo che va avanti per sempre perchè il database si modifica continuamente, si modifica il volume di dati che lavorano con il database e il numero di query che vengono richieste.

Si tratta di un processo opportunistico perchè prima viene fatto il deploy del sistema e poi cerchiamo di capire se quello che è stato fatto va bene o no e in caso facciamo il tuning modificando quale parametro o modificando le strutture dati che vengono utilizzate.



Il tuning potrebbe essere un procedimento che non termina mai perchè ci può essere sempre qualcosa da migliorare, il punto chiave consiste nel cercare il bilanciamento corretto tra costi e benefici.

Dobbiamo considerare se quello che vorremmo modificare nel nostro sistema mi costa troppo rispetto ai benefici che ottengo con quella modifica.

Ad esempio aggiungere tanti indici inutili potrebbe essere un costo per l'aggiornamento anche se magari mi migliora qualcosa dal punto di vista della velocità delle query.

Posso fare ad esempio la denormalizzazione che sarebbe il contrario della normalizzazione. Se per fare la normalizzazione prendo un database iniziale in cui ci sono ridondanze e le elimino qua si fa l'opposto e si prende un database normalizzato in cui si vanno a splittare i dati. Migliori la velocità delle query ma il costo sarà che poi avrò ridondanza di dati e quando devo fare delle modifiche dovrò fare più aggiornamenti.

C'è una regola empirica che mi permette di capire come fare le ottimizzazioni e quando fermarsi.

È il principio di Pareto (80/20 rule) che mi dice che dovrei ottenere l'80% dell'effetto desiderato svolgendo il 20% del lavoro (cambiando il 20% dei parametri ad esempio).

Questa regola non è sempre valida perchè magari all'inizio quando non ho fatto ottimizzazioni potrei avere un 95/5 mentre dopo un po' che faccio le ottimizzazioni non supererò più l'80%.

Quindi il punto chiave è che la soluzione migliore è nel mezzo, bisogna avere abbastanza conoscenza pratica per capire cosa modificare e come modificarlo per ottenere una migliore ottimizzazione del sistema.

Quando ci troviamo ad avere un sistema che ha delle performance peggiori di quelle che vorremmo la prima cosa che si dovrebbe fare è cercare di rivedere il physical design ovvero dobbiamo capire quali sono gli access plan che stiamo utilizzando, quali sono gli operatori e che costi hanno.

Ogni DBA dovrebbe essere in grado di leggere un access plan, in alcuni casi questa può essere una operazione con un costo molto alto perchè se l'access plan è complicato ci vuole del tempo.

Una volta compreso l'access plan potremo avere però un primo indizio su cosa modificare e su quali indici utilizzare in modo da avere un access plan che sarà migliore.

In alcuni casi può anche capitare che sia lo stesso ottimizzatore del database che sceglie una ottimizzazione che risulta inutile, quindi si deve considerare anche che in alcuni casi le ottimizzazioni non funzionano come previsto.

L'ultima cosa che si tende a modificare è l'hardware.

Per iniziare quindi si parte con le query che hanno performance basse e poi si analizza l'access plan cercando fare delle modifiche che possono portare dei miglioramenti alla velocità di esecuzione della query.

**Index Tuning:** Una delle misure di tuning che viene applicata maggiormente è l'index tuning, quello che si fa è cercare di capire se la presenza di un indice può migliorare o meno la velocità delle nostre query. Magari potremmo aver pensato il nostro database considerando un workload che poi in realtà si dimostra differente, quindi in questo caso potrebbe essere meglio cercare di fare una modifica e creare un indice differente.

Il tuning dell'indice mi permette solitamente di diminuire la quantità di dati che vengono acceduti e quindi mi fa passare magari da un table scan ad una scansione parziale.

Il problema in questo caso è che potremmo avere un overhead dal punto di vista delle lock e poi potremmo avere un alto costo per quel che riguarda l'aggiornamento degli indici.

Query Tuning: se non mi basta la fase di index tuning si passa alla fase di Query Tuning.

Sappiamo che se abbiamo delle condizioni separate da AND possiamo creare un indice che mi permette di avere una migliore performance. In alcuni casi la query potrà anche essere riscritta quando abbiamo un AND che lavora su due attributi identici, in questo caso potremmo mettere un range.

Quando però le condizioni sono in OR possiamo avere dei problemi.

Quindi dovremmo trovare il modo per riscrivere la condizione con OR, se ho una condizione del tipo State=Italy OR State=Germania allora possiamo riscrivere la query come State In {Italy, German} perchè da un punto di vista semantico non cambia niente però dal punto di vista dell'ottimizzatore cambia molto.

Un punto fondamentale che spesso non può essere svolto dall'ottimizzatore è la riscrittura delle subquery. Questo è un procedimento difficile ma allo stesso tempo utile perchè la query viene poi eseguita in modo molto più veloce perchè avere una subquery è come avere un nested loop.

L'ottimizzatore può riscrivere query di questo genere facendo una join ma quando nel select non c'è il distinct non può fare niente. In questo caso è compito del programmatore cercare di riscrivere la query e potremmo riscriverla sostituendo l'exist con una outer join.

Sempre nelle sottoquery è un problema avere delle funzioni di aggregazione perchè sono difficili da ottimizzare, se vogliamo gli studenti che hanno passato meno di due esami non voglio usare una subquery quindi dovrei ottimizzare passando ad un modo differente di scrivere la query, potrei ad esempio riscriverla con un join tra esami e studenti e poi cercare di capire da qua quanti hanno fatto ad esempio

meno di due esami. Questa riscrittura è molto difficile e quindi la fa il programmatore.

È essenziale che questa parte di ottimizzazione venga svolta solamente dopo la fase di controllo del physical access plan. Non si deve mai saltare la prima fase ed arrivare subito alla fase di query tuning perché c'è il rischio di fare del lavoro in più che nella pratica non servirebbe a nulla.

### **Transactions Tuning:**

Possiamo avere un miglioramento delle performance delle transazioni se riusciamo a limitare l'overhead dovuto alla presenza delle lock.

Possiamo limitare l'overhead se riusciamo a capire bene quale meccanismo utilizza il nostro sistema, quindi se abbiamo le lock o se invece viene usato un sistema optimistic.

Il primo punto importante è che si deve cercare di ridurre l'utilizzo delle lock quando creiamo il database e quando dobbiamo fare delle transazioni che comportano molte operazioni.

Queste transazioni molto lunghe sarebbe poi meglio splittarle in transazioni più semplici e più veloci da eseguire.

In alcune situazioni però dobbiamo tenerci la transazione lunga e non possiamo splittare la transazione in alcune transazioni più piccole.

Immaginiamo una transazione che vuole fare una prenotazione di un posto in aereo, la transazione mi mostra una pagina con i posti che sono attualmente liberi, scelgo e poi me lo assegna. In principio dovrei mettere una lock in questa situazione perché è sbagliato che due persone scelgono lo stesso posto.

Questo però non va bene perché non posso mettere una lock e bloccare gli eventuali altri acquirenti.

Quindi quello che fa il sistema: legge la lista delle sedie libere e le fa vedere, poi l'utente sceglie e allora il sistema apre una nuova transazione e fa un controllo ottimistico per vedere se quel posto è

ancora disponibile, se è ancora disponibile me lo assegna altrimenti dice che non è più disponibile.

Questo è come funzionano i veri sistemi, non prendo mai una lock su tutto il database ma uso un sistema ottimistico.

Questa è usata anche in tante altre situazioni, dove abbiamo transazioni lunghe che non possono essere splittate.

Un altro punto importante è la scelta della granularità del blocco corretta ovvero della dimensione del blocco su cui mettiamo la lock.

In molti database abbiamo l'approccio che se il cliente non dice nulla la granularità viene scelta a runtime e il sistema la sceglie liberamente. Il cliente però può anche scegliere la granularità e fare in modo che il sistema lavori in quel modo.

Un altro punto importante riguarda il livello di isolation e questo è fondamentale e viene considerato in ogni database reale.

Ogni DBMS permette di richiedere un livello di isolation più basso rispetto a quello che ci aspetteremmo in teoria. Quando creiamo il sistema siamo noi a scegliere quanto deve essere il livello di isolation.

Ad esempio se vogliamo che tutte le transazioni siano serializzabili allora avremo un alto livello di isolation. In realtà non vogliamo che tutte le transazioni siano serializzabili quindi mi potrebbe bastare anche un livello di isolation più basso.

Decidere il livello dell'isolation è in pratica l'uso più comune della transaction optimization.

Abbiamo vari possibili livelli di isolation in SQL:

- **Read Uncommitted:** è il livello più basso di isolation level che possiamo avere. Le operazioni di scrittura in questo caso vengono fatte prendendo la lock, quelle di lettura invece vengono eseguite senza prendere la lock. Questo può comportare la presenza del dirty bit problem ovvero possiamo avere una transazione che legge un valore di un attributo che è stato aggiornato da un'altra transazione prima che questa committi.
- **Read Committed:** in questo caso abbiamo una lock per le scritture come prima e poi abbiamo anche un'altra lock per le letture che è

una shared lock. In questo caso non abbiamo il problema del dirty bit perchè quando scrivo prendo la lock e in contemporanea non posso leggere. Però si presenta il problema dell'unrepeatable read perchè quando leggo un attributo e poi rilascio la lock, se devo fare di nuovo la lettura potrebbe accadere che questo attributo è stato modificato e quindi avrò un valore differente.

Oltre al problema appena descritto c'è anche il problema del lost update, perchè se leggo l'attributo balance, poi qualcuno lo aggiorna e io lo aggiorno, aggiornando il balance perdo gli effetti che ci sono stati nel frattempo.

In pratica possiamo dire subito che questo tipo di "isolation" rilassato può essere utilizzato solamente con le read only transactions. Di solito quando una transazione modifica i dati e lascia l'effetto permanente nel database vorremmo che la transazione sia eseguita in modo serializable perchè altrimenti avremmo problemi con la concorrenza. Quindi in pratica posso abbassare il livello dell'isolation solamente per le read e non per le read e le write.

- **Repeatable Read:** in questo caso si utilizzano lock che sono condivise nel caso di letture di un record o esclusive nel caso di scritture di un record. Le lock le teniamo fino alla fine della transazione. Corrisponde (più o meno) a quello che chiamiamo serializability.

In questo caso il problema che si può verificare consiste nel phantom record ovvero ci troviamo ad avere un inserimento di un nuovo record senza che venga usata la lock su tutta la tabella e magari nel frattempo abbiamo calcolato la somma del balance di ogni record. Quando calcoliamo la media e contiamo quanti record ci sono troviamo un record in più e la media è sbagliata.

Siamo serializzabili fino a che non viene inserito un nuovo record.

- **Serializable:** se vogliamo avere una soluzione che sia serializzabile allora è necessario che il sistema adotti una soluzione di tipo multi granularity ovvero una lock di dimensione differente a seconda della necessità. Ad esempio una lock su un record oppure su tutta la tabella se devo inserire un nuovo record.

Questo tipo di approccio è molto importante per la concorrenza perchè mi rallenta un po' visto che in alcuni casi mi dovrò fermare ogni volta che devo inserire un nuovo record.

Tutto questo discorso sulla isolation in realtà semplifica un po' la situazione reale che in realtà è molto più complicata. Deve essere visto non come verità assoluta ma solamente per capire che esistono vari approcci per gestire l'isolation.

Capire bene il funzionamento dell'isolation è complicato.

Non tutti i sistemi di database commerciali implementano questo tipo di isolation, molti sistemi adottano una forma differente di isolation level.

Ad esempio Oracle utilizza una sorta di snapshot technique perchè utilizza la lock sui dati che devono essere scritti mentre invece per leggere non mi proteggo con la lock ma utilizzando gli snapshot.

Quando ho una lettura il sistema controllerà se quello che ho letto corrisponde a qualcosa che è stato modificato da quando ho iniziato la mia transazione e in tal caso andrà nel log a prendere il valore che avevo all'inizio della transazione.

Questo mi garantisce una forma di snapshot isolation ma non una serializability.

La verità è che questi tipi di low isolation li posso usare solamente se ho delle letture ma non in occasione di scritture.

## Logical Schema Tuning

Possiamo apportare una modifica che potrebbe migliorare le performance delle query anche dal punto di vista dello schema logico.

Abbiamo varie possibilità:

- Horizontal partitioning: possono esserci delle situazioni in cui voglio dividere la mia tabella "orizzontalmente" in base al valore di un attributo. Se ad esempio scopro che il 99% delle query riguarda l'anno 2019 allora posso considerare solamente quei record che hanno il valore 2019 nell'attributo anno.

Questo vuol dire che posso o creare una nuova vista dedicata all'anno corrente in modo che quando devo fare le query utilizzo

questa e non vado a considerare tutto il passato. Se poi devo considerare anche il passato ho comunque la vecchia tabella. Questo è molto utile quando si devono effettuare delle analisi sui dati. Questa tecnica è molto semplice ma anche molto utile in situazioni di data analysis, se invece ho transazioni classiche non ha molto senso. È comune anche quando abbiamo dei database distribuiti, ad esempio potrei voler memorizzare i dati europei nel data center europeo e quelli americani nel data center americano.

- Vertical partitioning: abbiamo un'altra possibile ottimizzazione, assumiamo che abbiamo fatto tante analisi statistiche ma poi in realtà mi interessano solamente alcuni campi della tabella che in realtà contiene 15 campi. In questo caso sarebbe più sensato prendere i dati che realmente mi servono e creare una nuova tabella in modo che quando devo caricare i dati che mi servono prendo solamente le colonne che mi servono ed evito di prendere tutte le altre. Se devo solamente fare una scansione del database in questo modo sarò più rapido, se invece devo fare anche altre query avrò un rallentamento.
- Denormalizzazione: questa è un'altra tecnica molto utilizzata. Di norma viene svolta la normalizzazione ovvero abbiamo una tabella e noi andiamo ad eliminare le ridondanze. In alcuni casi però è più utile fare la denormalizzazione, pensiamo al caso in cui abbiamo una tabella esami e per ogni esame vogliamo anche il nome dello studente.

Possiamo o fare ogni volta una join oppure in alternativa possiamo creare una view della join in modo che accedo direttamente a quella senza fare ogni volta la join.

Quindi quello che succede è che la query sarà più veloce ma invece l'inserzione e l'update dei dati sarà più lenta e più complicata.

## DBMS Tuning

Se tutte le varie modifiche che sono state fatte non portano benefici possiamo intervenire sui parametri del DBMS.

Ci sono vari parametri che possono essere modificati:

- **Il transaction management:** interveniamo sui parametri del DBMS come ad esempio il modo in cui vengono memorizzati i log, ogni quanto tempo vengono creati i checkpoint e quindi ogni quanto tempo facciamo i dump del database.
- Dobbiamo considerare anche la dimensione del buffer, questo è un parametro molto importante, non dobbiamo avere una dimensione molto grande che supera anche la dimensione del disco perchè non va bene.
- Dobbiamo considerare anche il modo in cui gestiamo il disco, quindi potremmo pensare di utilizzare un raid se abbiamo una bottleneck I/O
- Se abbiamo architetture parallele poi si apre un altro mondo

Il tuning è una questione di conoscenza, quello che ha più conoscenza però è il DBMS stesso che conosce il modo in cui vengono fatte le query, conosce il database all'interno e conosce il sistema operativo. Il DMBS inoltre può anche eseguire dei test e quindi invece di forzare il DBA a fare tanto lavoro possiamo anche permettere al database di eseguire il tuning da solo.

Da questo punto di vista i database stanno facendo grandi passi avanti, 20 anni fa questa cosa non era pensabile a causa del business model delle aziende che producevano database. Con il tempo c'è stata una evoluzione e si è cambiato il business model, quindi ora tutte le aziende stanno cercando di migliorarsi sotto questo punto di vista.

## Lezione 12: Decision Support Systems

I decision support systems sono quei sistemi che sono utilizzati per trasformare i dati in informazioni.

Ci troviamo quindi a fare una differenza tra OLAP (on line analytical processing) ovvero dei processi analitici in cui le query sql toccano tanti dati e OLTP (on line transaction processing) ovvero procedimenti in cui viene modificato un record o magari ne viene aggiunto uno nuovo.

I processi analitici diventano sempre più importanti di anno in anno.

Gli analytical process classici (OLAP) sono quelli che sono stati definiti nel corso degli ultimi 15/20 anni.

Olap e Big Data hanno entrambi lo stesso obiettivo, abbiamo a disposizione dei dati e vogliamo estrarre informazioni da questi dati.

Differenze tra i due:

	OLTP	OLAP
Obiettivo	Usati per eseguire delle operazioni (ad esempio per spostare dati da un conto all'altro)	Usati per prendere decisioni
Utente	Persone che lavorano ogni giorno nella nostra organizzazione	I manager e i data analyst.
Dettaglio dell'analisi dei dati	La granularità minore possibile. Se ho la tabella esami e voglio inserire un nuovo record voglio poter inserire i nuovi record uno per uno	Se devo fare l'analisi non è detto che debba partire da un insieme di dati con una granularità bassa, posso anche partire con una granularità che è più alta, ad esempio potrei avere

		dei dati aggregati e potrei non voler analizzare i dati record per record. (Ad esempio potrei voler vedere quante unità di ogni elemento ho venduto).
Origine dei dati	Io lavoro sui dati interni che ho nel database	Qua i dati possono essere integrati con dati che arrivano dall'esterno, ad esempio se voglio vedere se un brand vende più di me posso inserire oltre che ai miei dati anche i dati dell'altro brand.
Applicazioni	Qua implemento le varie operazioni che devono essere svolte nel database e queste aumentano a mano a mano che lavoro. Ogni volta che voglio aggiungere una nuova operazione chiamo qualcuno che me la implementa. Ho un set fissato di operazioni conosciuto a priori.	Qua ho un set di operazioni che vogliamo eseguire tutte le mattine. Se mi sveglio con l'idea di voler trovare una correlazione che non ho trovato prima vorrei poter fare una query per verificare la mia teoria. Abbiamo quindi delle Ad-Hoc query che sono queste query che faccio solamente una volta per capire se quello che ho pensato è vero o no. Questo è costoso e stressante per il sistema. Il sistema per supportare questo

		dovrebbe essere organizzato diversamente, dovrei chiamare qualcuno che mi implementa l'operazione, in realtà io voglio eseguirla una volta e velocemente anche se sono un manager e se non conosco SQL. È un requisito importante per un sistema che deve supportare il decision process.
Record per operazione	Pochi dati alla volta	Tanti dati alla volta (milioni)
Aggiornamenti	Aggiorniamo i dati di continuo e pochi alla volta, questo può anche comportare problemi con le deadlock.	In un analytical system fissiamo un orario in cui verranno caricati tutti i dati che ho. Quando finisco di caricare i dati posso eseguire la mia analisi senza però toccare i dati di nuovo. La parte di load prende molto tempo e carica molti dati.
Stato dei dati	Dati attuali	Dati storici
Memorizzato in	Memorizzati in un database	Memorizzati in un Data Warehouse. I sistemi che supportano i Data Warehouse sono chiamati "Data support System"

Una differenza fondamentale sta anche nel modo in cui vengono memorizzati i dati, negli operational system i dati sono organizzati in database che sono gestiti da classici DBMS e le applicazioni sono usate per eseguire applicazioni sul database.

Se abbiamo invece un Decision Support System i dati sono memorizzati nei data warehouse ovvero collezioni di strutture che sono progettate utilizzando tecniche che sono differenti rispetto a quelle dei classici database. Anche la gestione sarà affidata a DBMS specializzati e saranno anche utilizzate delle applicazioni di “Business Intelligence” per analizzare i dati.

L’idea di Data Warehouse nasce nel 1990, un Data Warehouse è un database specializzato.

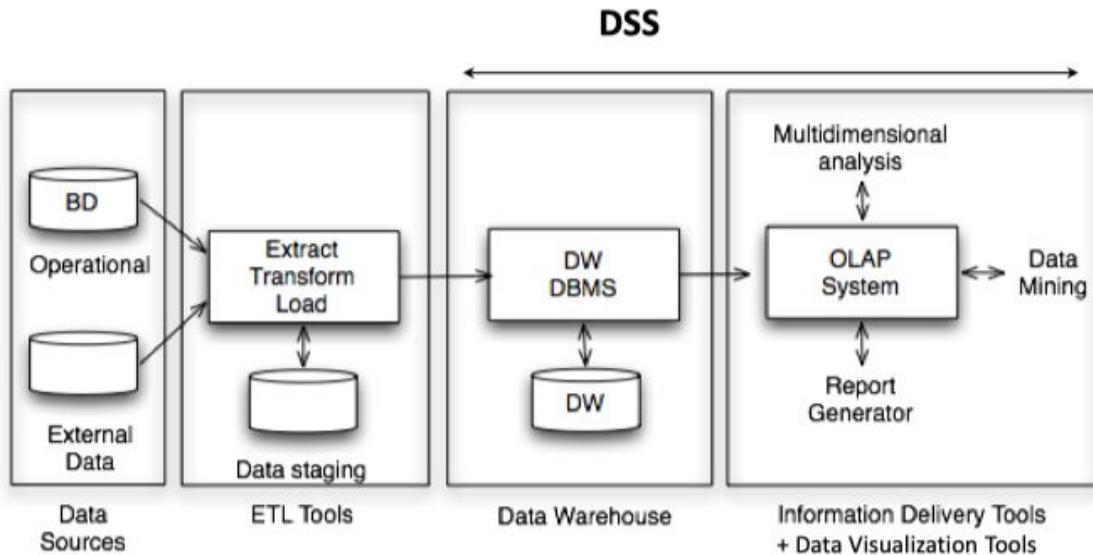
Quando creiamo un database, lo sviluppiamo in modo democratico nel senso che ogni tabella si prende la sua entità e ogni tabella ha la stessa importanza.

Quando organizzo un data warehouse devo iniziare chiedendo quale sarà l’interesse principale del cliente, su quali dati dovranno essere fatte le analisi?

Se vogliamo analizzare le vendite dovrò avere uno schema che non è democratico (“flat”) ma dovremo avere delle tabelle che sono organizzate attorno al concetto di vendita perchè questo è il mio interesse principale, tutte le altre tabelle sono meno importanti.

Se vogliamo analizzare più di una tabella avremo più di una parte all’interno del DW, ognuna centrata sulla parte che dobbiamo analizzare e ci sarà una ridondanza dei dati.

In un database la ridondanza è un problema per via degli update, in questo caso però, nel DW non abbiamo un problema di questo genere perchè i dati vengono aggiornati solamente una volta al giorno quindi paghiamo un costo alto solamente all’inizio e poi lavoriamo senza problemi.



### Come si crea un data Warehouse?

- Si inizia dal database che abbiamo a nostra disposizione e da eventuali dati esterni.
- Si estraggono le informazioni che vogliamo analizzare e che ci interessano.  
I dati poi vengono trasformati e viene eseguita la fase di data cleaning. I dati vanno puliti perchè potrebbero mancare informazioni, ad esempio potrei avere delle informazioni opzionali che non ho inserito. Questa fase non è semplice e potrei dover fare molte modifiche ai dati per ripulirli. Questa fase di trasformazione consiste anche nella discretizzazione dei dati, ad esempio i voti dell'attributo “Voto” di un esame potremmo discretizzarli e creare dei bucket
- Alla fine carico i dati che ho creato all'interno del data warehouse

La parte del lavoro che costa di più è quella che viene fatta nel data warehouse system (la seconda parte della foto) perchè abbiamo la parte di data cleaning che è un task complicato ma comunque va fatto.

Una volta che ho caricato i dati nella data warehouse ho a disposizione vari tool per lavorare su questi dati. Ho un subsystem che è l'OLAP

system che mi permette di gestire questi dati applicando algoritmi di data mining e di data analysis.

Abbiamo a disposizione:

- Dati
- ETL Tools
- Data Warehouse

Questo è l'approccio classico ma ne è stato sviluppato uno differente che si chiama data lake.

In questo caso nell'approccio data lake prendiamo tutti i nostri dati e li buttiamo all'interno di questo "lake" quindi eseguiamo tutte le analisi che dobbiamo fare. Questo approccio ha sia dei vantaggi che degli svantaggi.

È un approccio più veloce perchè mettiamo tutti i dati nel lago sperando che l'algoritmo trovi pattern interessanti e non facciamo nemmeno la parte di data cleaning che porta via tempo. Proprio l'assenza della fase di data cleaning è un problema perchè i dati potrebbero non essere troppo interessanti.

Un altro approccio è il "query database" quello in cui prendiamo il nostro database ed eseguiamo su questo gli algoritmi, questo è simile al data lake e ha sempre vantaggi e svantaggi.

Lo svantaggio è che abbiamo dei task molto complicati che sono ad esempio quelli della fase di data cleaning e che però sono anche utili. Il secondo problema è che se eseguo la query direttamente su questo database ho il contention problem e potremmo avere deadlock e overload del database, quando ad esempio il manager vuole eseguire la query Ad-hoc il sistema sarà in overload per un tot di minuti.

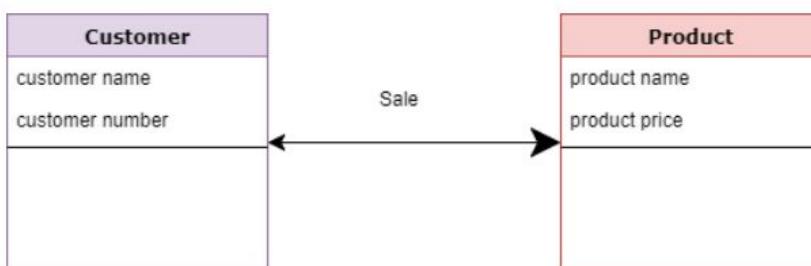
Possiamo anche creare una copia del database in modo che ogni transazione che viene eseguita qua viene anche eseguita sul database di backup.

Abbiamo tre approcci ma in realtà quello canonico è il primo.

La creazione di un Data Warehouse avviene in passi graduali e a differenti livelli di astrazione. Viene creato un modello concettuale, poi un modello logico e poi un modello fisico, come per la creazione di un normale database.

Ripasso:

- Modello Concettuale: è il modello in cui vengono definite le entità che saranno presenti all'interno del database, i vari attributi e le relazioni tra le entità.



- Modello Logico: gli attributi delle varie entità vengono indicati in modo più specifico e in particolare viene indicato anche il tipo. Anche le relazioni sono indicate più nello specifico e per ogni relazione si indica anche il tipo.
- Modello fisico: descrive l'implementazione specifica del database, ad esempio permette di scegliere le chiavi primarie e secondarie. Offre una astrazione del database e permette la generazione dello schema.

Per il modello concettuale nel data warehouse viene utilizzato il Dimensional Fact Model, il logical model invece sarà un normale Relational Data Model.

Poi c'è il Multidimensional Data Model che è un nuovo data model.

Come creiamo il “Data Mart” ovvero la parte del data warehouse che vogliamo studiare ed analizzare?

Si parte con uno schema di un database, abbiamo un modello entità relazioni (se ho un taglio nelle relazioni vuol dire che il valore della tabella che ha il cut è opzionale).

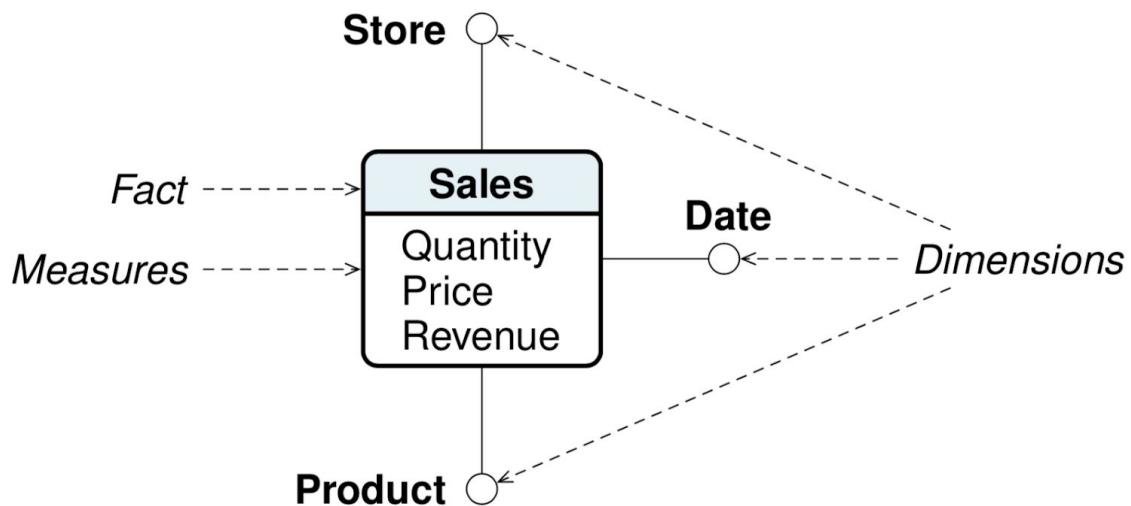
Partendo da questo modello dobbiamo creare un data mart e per farlo dobbiamo capire quali sono le domande che vogliamo fare al data mart. Per esempio vorrei poter estrarre dal database il numero di oggetti che sono stati ordinati, ordinati in base al prodotto, al cliente e al mese in cui sono stati ordinati.

In ognuna di queste richiesteabbiamo una misurazione e un set di dimensioni, quello che vogliamo sapere veramente è come il numero di output dipende dagli input (quindi da cosa dipende il numero di ordini). Spesso ho business question che sono più semplici, ad esempio potrei avere solamente una dimension, ad esempio se voglio "revenue by day" abbiamo che la measure è la total revenue mentre la dimension è il "by day". Spesso abbiamo una correlazione alta tra le query che riguardano una business question.

Tutte queste sono query con un group by, quello che vogliamo veramente capire sono tre cose:

- Quali sono i fatti che il tuo cliente vuole analizzare
- Quali sono le measure
- Un log con le dimensions

Se abbiamo capito quali sono le business question ovvero a quali facts è realmente interessato il cliente allora possiamo passare al modello concettuale.



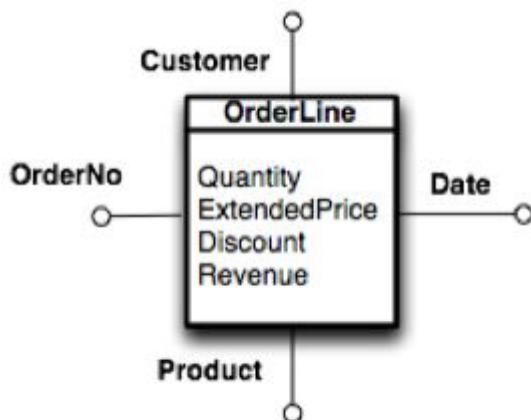
Definizioni:

- **Facts**: sono gli “eventi tipici” ovvero i fatti che accadono, potrei avere dei fatti che accadono una sola volta e che mi interessa analizzare, come ad esempio i dati delle vendite o le visite. Poi ci sono i fatti periodici in cui consideriamo che un fatto in realtà non è una singola vendita ma è una linea nella fact table ovvero sono i fatti che avvengono in un certo periodo. Poi ci sono gli accumulating fact in cui ogni fatto cancella e aggiorna il fatto che lo precede.
- **Measures**: sono informazioni quantitative, l’aggregazione di queste informazioni risulta interessante per il cliente. È una proprietà numerica di un fact che descrive uno dei suoi aspetti quantitativi che sono interessanti per l’analisi. Tipicamente aggreghi informazioni che riguardano la stessa measure. Quando misuri quanti click vengono fatti in una pagina ad esempio devo aggregare l’informazione perché sommo tutti i click che ci sono stati. Ad esempio se ho una tabella di complaints e voglio analizzare quanti ne ho in base al customer o al prodotto, in questo caso non ho una measure vera e propria perché stiamo solamente contando, quindi possiamo mettere solamente un count\* perchè l’unica cosa che vogliamo fare è contare. Di solito abbiamo varie measure che vengono aggregate in una sola.

- Dimensions: sono legate alle domande che vengono fatte dal cliente (Who, Why, Where). Sono le variabili che influenzano le measure e noi vogliamo capire in che modo le influenzano. Forniscono un contesto al fact

In un object data model abbiamo delle classi con gli attributi, le classi però sono tante quante le tabelle che ci servono.

Se siamo in un data warehouse data model invece abbiamo una sola tabella per ogni fact e per ogni tabella abbiamo misure e dimensions. Questo data model lo possiamo rappresentare graficamente con una fact table (un fact tipicamente è un event e tipicamente ce ne sono tanti a cui sono interessato) al cui interno sono presenti le measure a cui sono interessato.

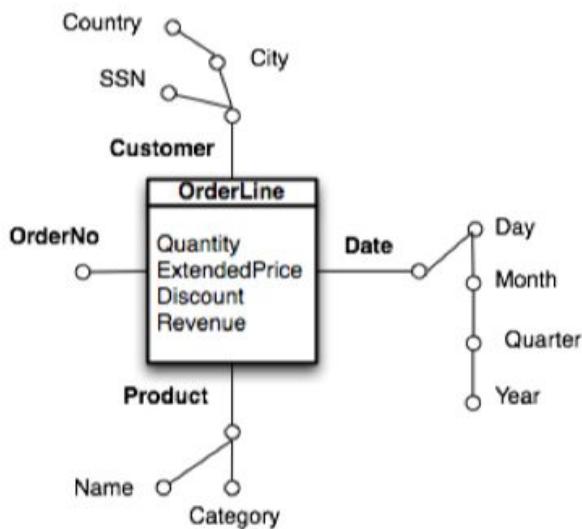


All'esterno di questo rettangolo metto la lista delle dimensions, metto le dimensions all'esterno perchè potrebbero avere i loro attributi perchè sono oggetti strutturati a differenza delle measure che invece sono tipicamente degli interi.

Alcune measures sono poi chiamate degeneri perchè non hanno attributi.

In un database queste sono classi, qua le chiamiamo facts.

Quando creo un data mart con measures e dimensions abbiamo che le dimensions potranno avere vari attributi e questi attributi sono organizzati in gerarchie. Ad esempio data la dimension "Date" abbiamo vari attributi collegati che sono organizzati in una gerarchia, abbiamo l'attributo Day che mi determina il mese, poi il trimestre e alla fine l'anno.



Quando abbiamo un caso di studio e dobbiamo capire quali sono le richieste che vengono fatte dal cliente possiamo creare una tabella in cui studiamo:

- Le dimension che vengono richieste
- La measure
- La metrics: in generale questa è diversa dalla measure perchè la measure è il singolo fatto mentre invece la metrics è il modo in cui aggregiamo le varie measure. Quindi ad esempio una measure potrebbe essere il ricavo di una vendita mentre la metrics è la somma dei ricavi di tutte le vendite.

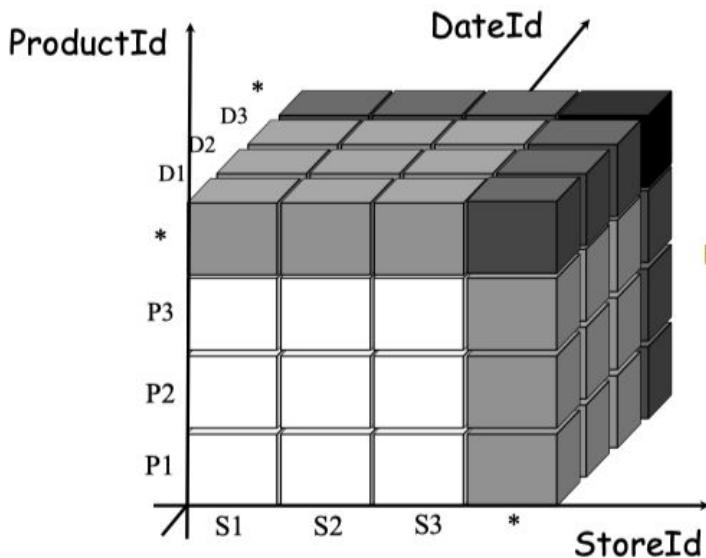
## Multidimensional model

Il cube data model è simile al data model relazionale, abbiamo comunque facts, measure e dimensions, cambia però il modo in cui eseguiamo la query dei dati.

Il cube data model è il core di un modello multidimensionale con un unico fact da rappresentare ma una serie di dimensions da considerare. Ad esempio possiamo considerare l'operazione “group by” come una manipolazione del cubo in blocchi di varie dimensioni.

Assumiamo di avere una tabella con 3 dimensions e 1 measure, abbiamo una tabella con 4 colonne e lo possiamo visualizzare in una

tabella in 2 dimensioni, la stessa cosa però la possiamo visualizzare anche tramite un cubo.



Se ad esempio volessimo vedere per un certo giorno la somma totale del guadagno di ogni prodotto in ogni store, possiamo aggiungere una nuova colonna alla fine del cubo e qua fare la somma.

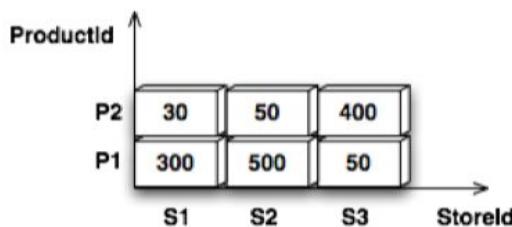
Ha un senso anche eseguire un taglio del cubo, ad esempio possiamo ottenere le vendite in ogni store in ogni giorno.

Se abbiamo una tabella lineare in cui i dati crescono solamente in una direzione, per ogni possibile coppia possiamo creare una informazione da inserire all'interno di un cubo in due dimensioni.

Ad esempio dalla tabella di facts sulla sinistra possiamo ottenere il cubo sulla destra:

Sales		
StoreId	ProductId	Qty
S1	P1	300
S2	P1	500
S3	P1	50
S1	P2	30
S2	P2	50
S3	P2	400

Fact Table



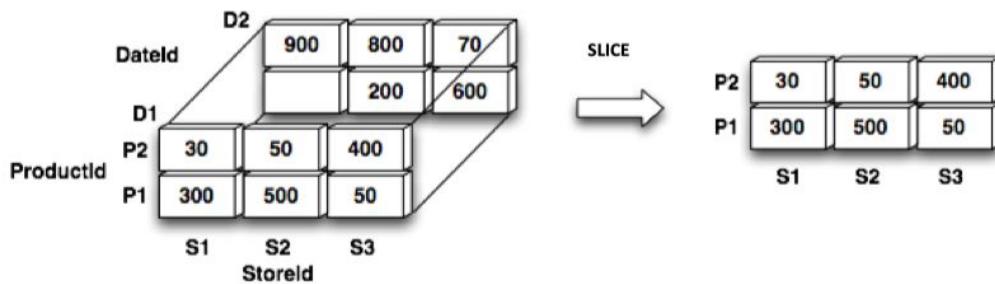
2-D Cube

Con l'aumentare dei dati contenuti all'interno della tabella (quando aumentano le colonne), dovremo aumentare anche le dimensioni del

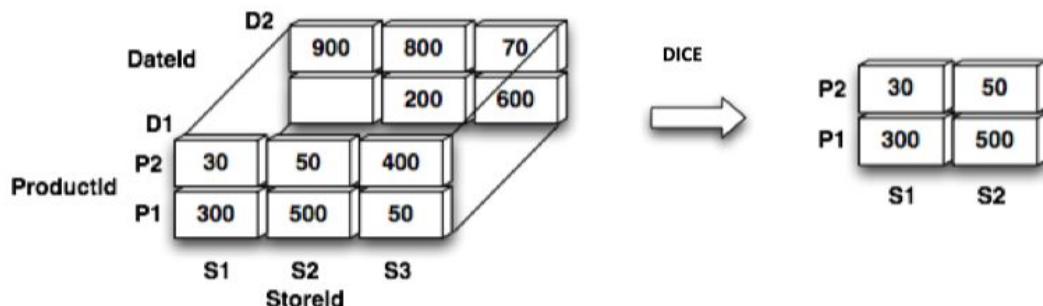
cubo, quindi ad esempio se avessimo 4 colonne allora abbiamo un cubo in 3 dimensioni.

Anche per il Cube data model abbiamo un query language:

- **Slice**: corrisponde all'operazione "Select", prendiamo una slice del cubo, quindi ad esempio in questo caso sarebbe come fare select datelD = D1.

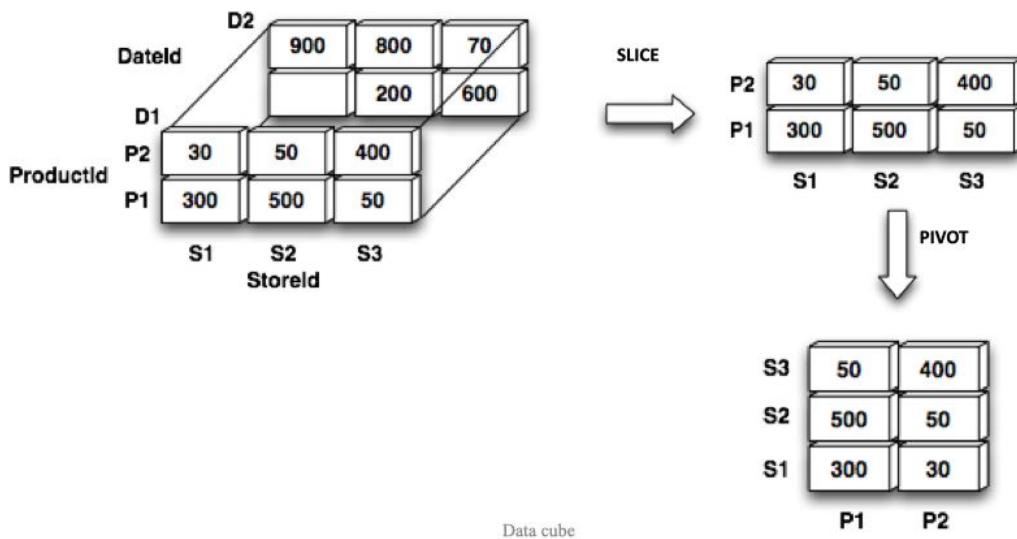


- **Dice**: operatore che viene utilizzato per selezionare degli intervalli in modo da ottenere un cubo più piccolo da quello che avevamo in partenza

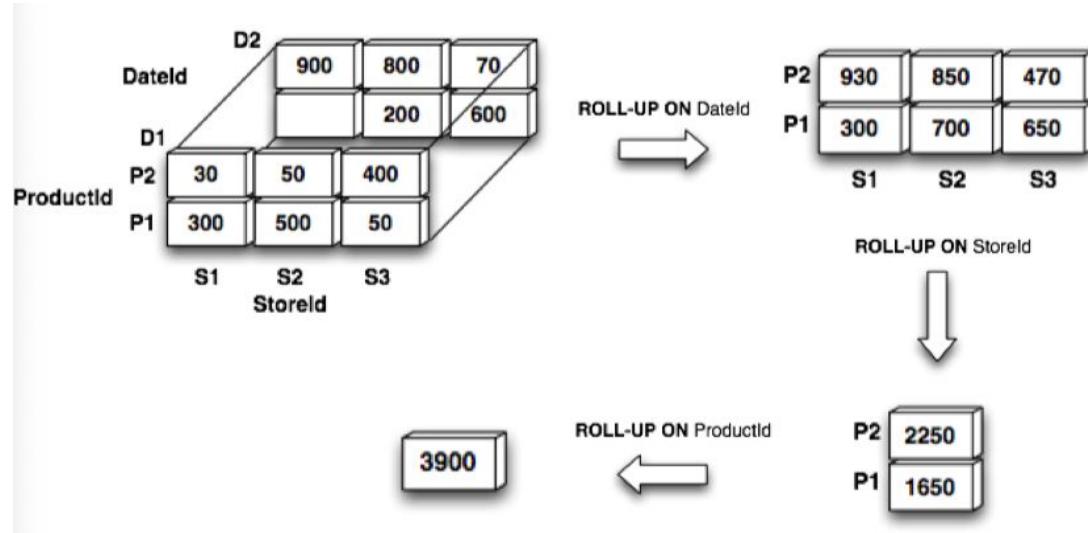


- **Pivot**: questo operatore è unico per i multi relational data model, una volta che abbiamo estratto i dati dal cubo si utilizza l'operatore pivot per eseguire una rotazione dei dati contenuti all'interno del

cubo.



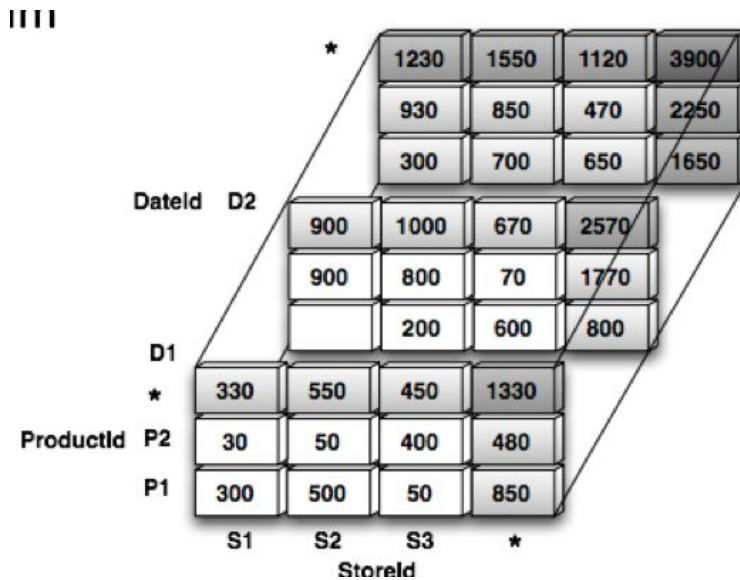
- **Roll-up**: Operazione che ci permette di fare una “somma” dei dati che sono contenuti all’interno del cubo.  
Roll-up è l’operazione duale del group By.



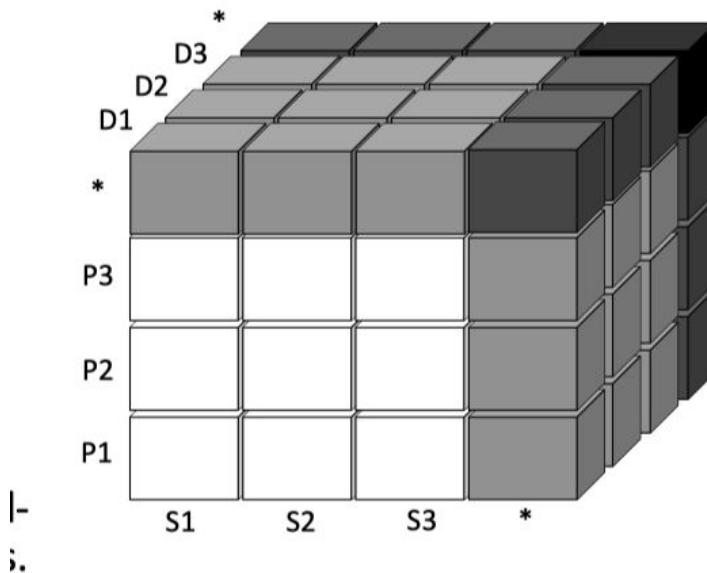
- **Drill Down**: questa operazione è l’opposto del roll-up e ci fa tornare nella situazione di partenza.

È possibile estendere ogni dimensione del cubo utilizzando l’operazione Roll-up, in questo modo abbiamo che la colonna finale è la somma dei contenuti delle celle interne e lo stesso vale con la riga che sta sopra. La cella in alto a destra è la somma complessiva.

Quindi in questo modo possiamo creare un Extended Cube che è un insieme di cuboidi e all’interno avremo sia le celle con i semplici dati che le celle con i dati complessivi.



Osservazione: se vogliamo permettere al cliente di fare delle query su questi cuboidi e vogliamo che le query siano molto veloci, dato che il group by è molto costoso, possiamo memorizzare un cuboide di questo genere e andare semplicemente a prendere il dato complessivo.



La domanda basilare da farci quando memorizziamo un database multi dimensionale è “quanta ridondanza vogliamo? Quanti cuboidi vogliamo

memorizzare?”. Questo è il motivo per cui questa struttura dati è interessante per l’ottimizzatore.

Il materialization problem è un physical design problem nel data warehouse system e si chiede quali cuboidi devo memorizzare e quali cuboidi devo calcolare. Non abbiamo problemi con la ridondanza. Un’altra domanda basilare: dove metto sul disco questi dati? Dove li metto in memoria principale?

Posso o portare tutto in memoria principale, se il cubo c’entra, se però non mi basta la memoria devo poter decidere quali informazioni devo portare in memoria principale e quali invece non vogliamo portare in memoria principale.

## Relational schema of DW

I relational OLAP system sono database relazionali che sono pensati specificatamente per supportare funzioni di business analysis.

Per implementare un data warehouse viene utilizzato un multidimensional relational model perchè abbiamo una fact table e un set di tabelle di dimensions.

Questo modello multidimensionale può adottare uno dei seguenti schemi:

- **Star schema:** in questo caso abbiamo ad esempio un fact che è la tabella “Sales” che contiene tutti gli attributi, per ogni dimension abbiamo una foreign key che mi permette di collegarmi con le varie tabelle delle dimensioni.

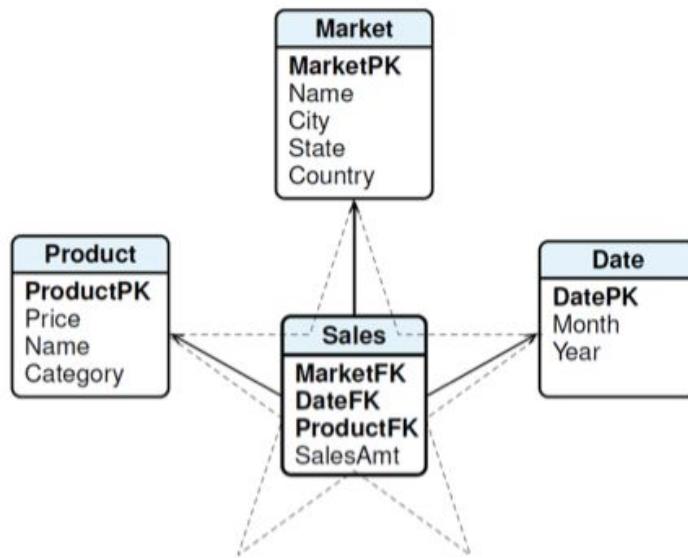
Abbiamo molta ridondanza, ad esempio in market abbiamo city, state e country che sono ridondanti perchè dalla città identifico anche le altre informazioni.

La ridondanza in questo caso non è un problema perchè qua noi non aggiorniamo i dati, ci limitiamo a fare delle query e quindi la ridondanza è un bene perchè mi velocizza le query.

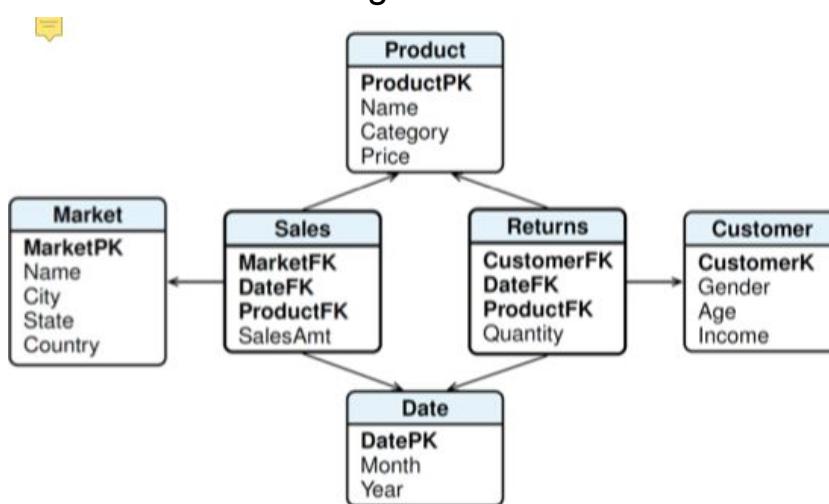
Il problema è quando in un database classico, qua dobbiamo eliminare la ridondanza perchè quando aggiorniamo ci sono problemi perchè devo modificare troppe informazioni.

Quindi in un DW, per ognuna delle tabelle relative ad una

dimension procediamo con una denormalizzazione della tabella e quindi in questo modo creiamo maggiori informazioni da inserire all'interno di ogni tabella.



- **Snowflake** è un altro approccio per questo tipo di problema ma non è molto utilizzato perchè qua le varie tabelle vengono normalizzate e quindi sono spartite in tabelle più piccole.
- **Constellation**: può capitare che in un DW abbiamo vari data marts che condividono le stesse dimensions. Quindi quando siamo in questa situazione possiamo mergiare le tabelle relative ai dimension che contengono dati simili.



## Lezione 13: Column Databases

Prendiamo questo esempio:

- Sales(Date,FKShop,FKCust,FKProd,UnitPrice,Q,TotPrice)
- Shops(PKShop,Name,City,Region,State)
- Customer(PKCust,Nome,FamName,City,Region,State,Income)
- Products(PKProd,Name,SubCategory,Category,Price)

Sales: NRec: 100.000.000, Npag: 1.000.000; Shops: 500, 2; Customers: 100.000, 1.000; Products: 10.000, 100

```
SELECT Sh.Region, Month(S.Date), Sum(TotPrice)
FROM Sales S join Shops Sh on FKShops=PKShops
GROUP BY Sh.Region, Month(S.Date)
```

Dobbiamo proporre una organizzazione primaria o anche un indice per svolgere al meglio questa query. Consideriamo che le date sono nel formato yyyy mm dd.

Può essere utile utilizzare la denormalizzazione o il vertical partitioning?

### Alcune proposte per la soluzione:

- Primary Sequential organization on Date: Dobbiamo fare due operazioni, la groupBy e la join. La join costa 0 perchè la tabella più piccola è grande solamente due pagine.  
Vogliamo ottimizzare la groupBy e la situazione ideale sarebbe quella in cui i nostri dati sono già ordinati. Il problema è che devono essere ordinati sia in base a region sia in base a date ma non abbiamo questi due attributi nella stessa tabella.

Con una sequential organization su Date possiamo caricare in memoria principale tutti i record consecutivi che hanno la stessa data e quando li abbiamo messi in memoria principale possiamo riordinarli in base alla region. Se entrano in memoria il costo sarebbe 0 però non c'è un operatore che mi fa fare questo.

Prendiamo un caso particolare, consideriamo la groupBy solamente su Date e non su region.

La nostra primary organization è sequential su date, il costo in questo caso dobbiamo fare select, from e groupBy, quindi dobbiamo fare una scansione della tabella sales con una lettura di 1000000 pagine. È un numero alto di pagine che devono essere lette.

Come possiamo ridurre questo numero di pagine?

La denormalizzazione (memorizzare il risultato della join) non aiuta perchè in questo caso la tabella mi diventa anche più grande di come era all'inizio e non è necessario. Se invece si fa il vertical partitioning della tabella Sales possiamo ridurre il numero degli attributi da 7 a 2 (Date e Price) quindi riduciamo il numero di pagine che sono necessarie.

Qua i punti chiave per eseguire la query velocemente sono due, dobbiamo tenere la tabella ordinata in base a Date perchè senza questo dobbiamo ordinare i dati e aumentiamo ancora di più il numero di pagine da leggere (avremmo 2 milioni di pagine per il sorting), poi c'è il vertical partitioning che riduce ancora di più.

Tornando alla query originale: possiamo fare anche qua la vertical partitioning ma serve anche una organizzazione che faccia in modo che i dati vengano organizzati sia in base a Date sia in base a Region. Una prima possibilità è quella di fare denormalization+vertical partitioning. Prima fai la join ma poi fai la proiezione sui soli attributi che sono necessari nella query. La denormalizzazione in questo caso ha un costo pari a 0. C'è anche un'altra tecnica possibile, possiamo fare con shop la stessa cosa che facciamo con Date. In date diciamo che ogni

valore è ordinato in base a anno, mese, giorno. La stessa cosa la possiamo fare con shop che potrebbe avere una chiave del tipo state, region, city e questo non sarebbe un problema perché noi i dati li aggiorniamo solamente la mattina una volta al giorno.

Osservazione: vertical organization è molto importante in queste situazioni. Come data organization potremmo anche memorizzare il risultato di questa query (una tabella raggruppata in base a region e date) oltre alla tabella originale.

Se non facciamo il vertical partitioning non c'è possibilità di fare altre ottimizzazioni perchè dovremmo considerare troppe pagine.

Se abbiamo una condizione nella query la stessa organizzazione continua ad essere ottima perchè se i dati sono ordinati in base alla data allora possiamo controllare la condizione molto velocemente.

## Column Stores

Fino ad ora abbiamo considerato solamente database che sono memorizzati linea per linea e per ogni linea possiamo avere più attributi. Questi tipi di database sono adatti in particolare alle applicazioni OLTP. Per quanto riguarda invece le applicazioni OLAP è stato pensato un altro tipo di organizzazione per i database ovvero una organizzazione per colonne. Nel DSM (decomposition storage model) quindi i dati vengono salvati in colonne e in pratica si passa da una tabella unica con vari attributi a tante tabelle con una sola colonna corrispondente ai singoli attributi.

Ognuna di queste tabelle a colonna necessita di un contatore extra che mi possa permettere di identificare quale sia il primo, secondo, terzo, ecc... elemento della colonna.

Se poi volessimo creare la tabella originale da quelle in colonna che abbiamo creato, consideriamo il fatto che le righe sono numerate ed eseguiamo il Merge Join, in alcuni casi poi potrebbe anche essere meno costoso fare la query con le tabelle sulle colonne che sulla tabella unica.

L'utilizzo dei database con column store si è diffuso negli ultimi 10/15 anni per vari motivi:

- Le applicazioni OLAP sono diventate sempre più importanti
- Le organizzazioni orizzontali sono più index-oriented e possiamo fare delle ottimizzazioni usando degli indici
- Poi ci sono stati problemi tecnologici perché la RAM è diventata sempre più grande e quindi il problema si è spostato sulla velocità di caricamento dei dati in memoria, vorremmo minimizzare il seek time
- Nelle colonne possiamo sempre avere un indice perchè è una tabella con due colonne
- L'I/O in questo caso non è più un problema, si cerca di fare operazioni in parallelo e poi di usare bene la cache. I database tradizionali non funzionano bene da questo punto di vista.

Abbiamo dei vantaggi dovuti all'utilizzo del column store:

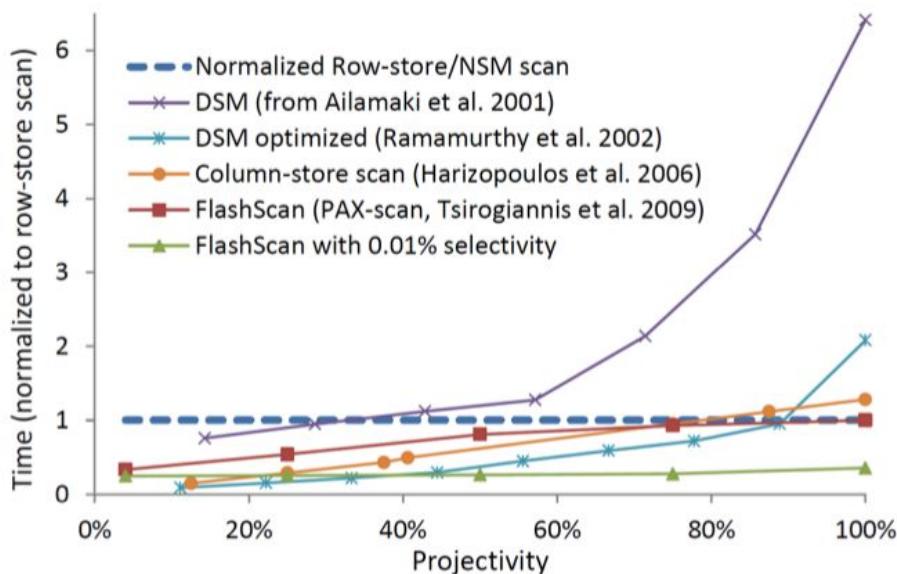
- Se devo caricare in memoria i dati delle tabelle, avrò meno dati da caricare perchè ho una colonna per ogni attributo e quindi porto in memoria solamente i dati che realmente mi interessano
- Se memorizzo colonna in colonna abbiamo la possibilità di comprimere il contenuto della colonna in modo da ridurre ancora di più lo spazio necessario per memorizzare i dati.
- Abbiamo un uso migliore della cache e della parallelizzazione, nel caso del column store infatti spostiamo i dati un blocco alla volta e non una tupla alla volta (1 tuple per la select, per la project...). Usare il 1 tuple at a time approach mi comporta un uso non buono della cache e l'impossibilità di utilizzare single instruction parallelism o una pipeline.

Qua ci spostiamo all'approccio 1 block at a time, muoviamo i dati in base a quanto è grande la cache. Muovere un blocco e andare in quel blocco vuol dire che possiamo utilizzare un for loop all'interno del blocco senza utilizzare delle function calls.

Ovviamente ci sono anche degli svantaggi:

- Se siamo interessanti a più di un campo della tabella diventa obbligatorio eseguire la join. Se abbiamo una organizzazione dei dati come quella indicata sopra, con una colonna per ogni attributo e un contatore della riga la join mi costa 0 ma perdiamo la possibilità di eseguire la compressione perché mantenendo la compressione la join diventa molto costosa.
- Un altro problema l'abbiamo quando dobbiamo inserire una nuova tuple all'interno del database. In questo caso infatti se abbiamo 10 colonne dovremo eseguire 10 operazioni di I/O.

Performance del column store:



Con il DSM originale il tempo per eseguire una operazione era accettabile rispetto ad una standard relational table quando richiedevamo < 30% delle colonne, altrimenti il DSM diventa peggiore. Gli altri modelli hanno ridotto l'overhead, gli altri storage sono buoni anche con un numero maggiore di colonne.

In un column database come possiamo ordinare le colonne?

- Una prima opzione consiste nel memorizzare le tuple ordinate in base al RID, in questo caso non serve memorizzare il RID.

Quindi in questo caso la ricostruzione della tabella originale mi costa 0 perchè viene fatto un merge join con una scansione parallela delle tabelle visto che sono ordinate.

- Una alternativa può essere quella di memorizzare i dati ordinati in base al valore che troviamo all'interno delle tuple, in questo modo riusciamo ad avere una compressione molto efficiente ma è necessario memorizzare per ogni tupla il RID corrispondente. La ricostruzione delle tuple quindi richiede una forma di join e non mi basta fare un merge join.

Per la memorizzazione delle colonne quindi abbiamo varie possibilità:

	RID-value	One RID per page	Implicit RID																								
Sorted by RID	<table border="1"> <tr><td>1</td><td>20</td><td>2</td><td>50</td></tr> <tr><td>3</td><td>45</td><td>4</td><td>20</td></tr> </table>	1	20	2	50	3	45	4	20	<table border="1"> <tr><td>1</td><td>20</td><td>50</td><td>45</td></tr> <tr><td>20</td><td>45</td><td>20</td><td>34</td></tr> </table>	1	20	50	45	20	45	20	34	<table border="1"> <tr><td>20</td><td>50</td><td>45</td><td>20</td></tr> <tr><td>45</td><td>20</td><td>34</td><td>50</td></tr> </table>	20	50	45	20	45	20	34	50
1	20	2	50																								
3	45	4	20																								
1	20	50	45																								
20	45	20	34																								
20	50	45	20																								
45	20	34	50																								
	<table border="1"> <tr><td>5</td><td>45</td><td>6</td><td>20</td></tr> <tr><td>7</td><td>34</td><td>8</td><td>50</td></tr> </table>	5	45	6	20	7	34	8	50	<table border="1"> <tr><td>8</td><td>50</td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>	8	50							<table border="1"> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>								
5	45	6	20																								
7	34	8	50																								
8	50																										
	Value-RIDS	Column + Join index																									
Sorted by value	<table border="1"> <tr><td>20</td><td>1</td><td>4</td><td>6</td></tr> <tr><td>34</td><td>7</td><td>45</td><td>3</td></tr> </table>	20	1	4	6	34	7	45	3	<table border="1"> <tr><td>20</td><td>*3</td><td>34</td><td>*1</td></tr> <tr><td>45</td><td>*2</td><td>50</td><td>*2</td></tr> </table>	20	*3	34	*1	45	*2	50	*2	<table border="1"> <tr><td>1</td><td>1</td><td>4</td><td>6</td></tr> <tr><td>2</td><td>7</td><td>3</td><td>3</td></tr> </table>	1	1	4	6	2	7	3	3
20	1	4	6																								
34	7	45	3																								
20	*3	34	*1																								
45	*2	50	*2																								
1	1	4	6																								
2	7	3	3																								
	<table border="1"> <tr><td>5</td><td>50</td><td>2</td><td>8</td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>	5	50	2	8					<table border="1"> <tr><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>									<table border="1"> <tr><td>5</td><td>4</td><td>2</td><td>8</td></tr> <tr><td></td><td></td><td></td><td></td></tr> </table>	5	4	2	8				
5	50	2	8																								
5	4	2	8																								

- **Sorted by RID**: si utilizza se abbiamo un valore di una lunghezza variabile. In questo caso abbiamo una coppia <RID, VALORE> per ogni tupla.
- **One RID per page**: anche questo lo usiamo se abbiamo valori di lunghezza variabile, per ogni pagina salviamo un rid differente.
- **Implicit RID**: se i valori sono interi possiamo usare questa organizzazione (è una delle più comuni), qua salviamo solamente i valori e non memorizziamo i RID
- **Value-RIDS**: creiamo una sorta di inverted index e per ogni possibile valore memorizziamo i RID corrispondenti.
- **Column + Join index**: questo è uno dei più comuni, vengono separati i valori dai RID. In una prima tabella memorizzo i valori e quante volte compare ogni valore. Nell'altra tabella, chiamata join index, invece memorizzo per ogni valore quali sono i RID

corrispondenti.

I dati sono ordinati e possiamo comprimerli.

C-Store

Come possiamo fare ad evitare l'utilizzo del join index e la tuple reconstruction?

Il C-Store svolge una proiezione dei dati in modo da evitare il costo della ricostruzione.

L'idea è che guardiamo le query che il cliente vuole eseguire maggiormente e per ognuna di queste query memorizziamo la proiezione della tabella che mi corrisponde a quella query.

Se memorizzo la proiezione non ho necessità di eseguire alcuna join, se devo anche eseguire una filter o una group By mi basta memorizzare queste proiezioni ordinate sull'attributo così in questo modo faccio velocemente anche quest'altra operazione.

La proiezione deve essere definita su un set di attributi che mi dicono su quali vogliamo fare la proiezione e su una lista di attributi che mi indicano su quali attributi deve essere fatto il sorting.

Optimal for  $\sigma_{k1 \leq \text{date} \leq k2} \pi_{\text{saleid}, \text{date}, \text{region}}(\text{Sales})$

(saleid,date,region   date)			
	saleid	date	region
1	17	1/6/08	West
2	22	1/6/08	East
3	6	1/8/08	South
4	98	1/13/08	South
5	12	1/20/08	North
6	4	1/24/08	South
7	14	2/2/08	West
8	7	2/4/08	North
9	8	2/5/08	East
10	11	2/12/08	East

(a) Sales Projection Sorted By Date

(prodid,date,region   region,date)		
prodid	date	region
1	5	1/6/08
2	9	2/5/08
3	4	2/12/08
4	12	1/20/08
5	5	2/4/08
6	7	1/8/08
7	22	1/13/08
8	3	1/24/08
9	18	1/6/08
10	6	2/2/08

(b) Sales Projection Sorted By Region, Date

Ad esempio se consideriamo la query dell'immagine, dobbiamo fare una proiezione considerando gli attributi saleid, date e region e poi dato che abbiamo una filter su date, dovremo anche avere un ordinamento della colonna date.

Questa qua non è una vertical partitioning o una horizontal partitioning, abbiamo comunque tre colonne ma sono tre colonne separate.

**Projection:** è una struttura dati definita da un set di projection attributes e una lista di sorting attributes e l'ordinamento della colonna dateid ad esempio comporta che anche le altre colonne saranno ordinate in base a date.

Perchè è importante che le colonne siano realmente separate? È molto importante perchè non ottimizzo solamente questa query ma ottimizzo anche le query che utilizzano meno dati rispetto a questa, ad esempio se voglio fare la proiezione solamente su saleid e date posso comunque usare questa proiezione.

Perchè non fare una proiezione con tutte le colonne? Perchè ogni proiezione seleziona un modo per ordinare le colonne, quindi se avessi solamente una proiezione con tutte le tabelle avrei comunque bisogno di altre proiezioni per ottimizzare le query che fanno la selezione o la groupBy su un attributo differente.

Per ogni query quindi preparo una proiezione differente, se ho molte query da ottimizzare potrei necessitare di molte proiezioni, se due query però hanno attributi differenti ma accettano lo stesso ordinamento allora posso fare una unione degli attributi e fare un'unica proiezione.

Questo metodo è molto ridondante perchè le informazioni sono riportate tante volte in tante proiezioni differenti, questo va bene se il mio sistema è un sistema OLAP ma non va bene se ho un sistema dove vengono eseguiti molti aggiornamenti.

Memorizziamo varie projection che possono sovrapporsi e ogni attributo della tabella deve essere presente almeno in un attributo delle proiezioni.

Quindi posso sempre ricostruire la tabella completa perchè ho tutti gli attributi.

Ci sono vari modi per fare questa ricostruzione delle tuple:

- Possiamo avere un join index tra due differenti proiezioni, se devo rimettere insieme 4 proiezioni avrò una catena di join index, uno che mi connette la prima con la seconda, uno per la seconda con la terza e uno per la terza con la quarta.

Un join index mi dice che una certa tupla corrisponde ad un'altra tupla di un'altra proiezione. Non abbiamo necessità di avere i RID

per creare i join index, abbiamo solamente la necessità di poter connettere una tuple da una tabella ad un'altra.

- Possiamo usare almeno una super projection che contiene tutte le colonne, quando dobbiamo ricostruire la tabella, invece di calcolare la join di tutte le colonne possiamo partire dalla super projection che è ordinata in un certo modo e l'unica cosa che possiamo fare è ordinarla in un modo differente.

In un column store i DBA preferiscono memorizzare la super projection e non definire un join index.

L'obiettivo del column store è eseguire ogni query con un index only access plan.

Quando si parla di access plan possiamo dire che esistono due approcci differenti per l'esecuzione di access plan:

- Tuple at a time: è quello visto fino ad ora con una tuple che parte dalle foglie e arriva al root. Nell'architettura tradizionale questa è la situazione migliore perché qua è comune fare una join tra una tuple e una relazione intera, in questo caso per eseguire la join si preferisce utilizzare l'index nested loop perchè il costo dipende dalla relazione più piccola mentre nel merge join dipenderebbe dalla relazione più grande. Quindi in questo caso sarebbe inutile portare in memoria tutti i dati relativi alla tabella più grande. (Se avessimo invece tanti studenti e tanti esami preferiremmo eseguire un sort merge).
- Operator at a time: questa è sempre stata considerata sempre la scelta più stupida e semplice, prima eseguo la filter, poi la join, poi la groupBy creando ogni volta dei risultati intermedi che però potrebbero non entrare in memoria principale riducendo quindi il parallelismo.

Le applicazioni OLAP devono ripensare all'idea di tuple at a time perchè in questi sistemi eseguiamo sempre delle join tra tabelle grandi.

L'esecuzione completa di una operazione alla volta (operator at a time) è più veloce nei processori moderni perchè c'è la possibilità di riempire la cache, quindi possiamo caricare un set di record alla volta e poi possiamo eseguire operazioni vettorizzate che mi permettono di eseguire un for loop invece di eseguire una function call perchè questa non funziona bene se vogliamo utilizzare un pipeline parallelism.

Internal pipeline parallelism delle CPU: in una CPU quando abbiamo del codice che deve eseguire 10 operazioni, eseguire una operazione richiede l'esecuzione di altre operazioni, nelle CPU moderne eseguiamo prima una prima fase della prima operazione, quando siamo alla seconda in contemporanea eseguiamo la prima fase della seconda operazione e la seconda fase della prima. Quindi possiamo eseguire tutte e 10 le operazioni in parallelo. Questo è possibile solamente se non abbiamo un IF e se non abbiamo una function call.

Quindi nelle CPU moderne ci sono buoni motivi per tornare all'idea della 1 operator at a time ma c'è il problema che il risultato intermedio potrebbe non entrare in memoria e questo è un disastro.

Quindi la soluzione è utilizzare un approccio differente chiamato 1 block at a time, l'operazione next() restituisce un blocco di record, quanti? La scelta tipica è scegliere 1000 valori alla volta che possono entrare nella cache L1.

In questo modo riusciamo a fare del lavoro in parallelo e sfruttiamo bene la cache.

Questa è una ottimizzazione tipica che troviamo nel column store.

Un'altra ottimizzazione tipica è l'idea della compressione, se memorizziamo e leggiamo i dati compressi avremo bisogno di un numero minore di operazioni di I/O per leggere tutte le colonne.

Risparmiamo sulle operazioni di I/O ma avremo una perdita di tempo dovuta alla compressione e alla decompressione dei dati, sarebbe ancora meglio se riuscissimo a lavorare in main memory con i dati già compressi perchè così risparmiamo il tempo della decompressione.

SIMD parallelism: situazione in cui la macchina è in grado di eseguire la stessa istruzione in parallelo su tanti dati. È meno espressiva rispetto al MIMD (Multiple instruction on multiple data), qua abbiamo varie CPU

perchè abbiamo tante istruzioni differenti e su ognuna viene eseguito il codice. MIMD quindi è un parallelismo tra CPU differenti.

SIMD invece è un parallelismo all'interno di una CPU, tutte le CPU moderne infatti hanno la possibilità di eseguire più operazioni su dati differenti a patto che i dati che vengono utilizzati siano memorizzati all'interno di un array.

Alcune operazioni di compressione:

- **RLE (Run lenght encoding)**: i dati in questo caso non sono raggruppati, se sono raggruppati l'encoding è ancora migliore. Per ogni carattere che deve essere compresso memorizziamo il carattere, la posizione in cui si trova e poi quanti caratteri uguali ci sono.
- **Bit-Vector encoding**: in questo caso abbiamo una compressione differente, per ogni carattere che dobbiamo comprimere abbiamo un bit vector e mettiamo ad 1 le posizioni in cui sono i caratteri.
- **Dictionary encoding**: definiamo un dizionario e assegnamo un valore per ogni elemento che dobbiamo codificare, poi dopo il testo codificato diventa un vettore in cui per ogni nome abbiamo il valore corrispondente del dizionario. Quando facciamo una range query sarà anche possibile eseguire una range query nell'encoding del dizionario.
- **Difference encoding**: in questo caso rappresentiamo ogni numero della sequenza come la differenza tra il numero e il precedente.

## Binary Algebra

Nell'algebra binaria ogni tabella è binaria, non abbiamo tabelle ternarie o tabelle più grandi. Una tabella binaria rappresenta (più o meno) una colonna (*rid,value*) nel modello DSM.

Usando solamente questa forma di tabelle binarie è possibile rappresentare tutte le tabelle ma all'interno dell'algebra possiamo anche avere delle tabelle che non sono nel formato *<rid, value>* ma magari sono un join index quindi *<rid, rid>* o un binary table che rappresenta un set di valori.

Solamente le tabelle che hanno la forma  $\langle \text{rid}, \text{value} \rangle$  mi bastano per rappresentare quello che viene rappresentato con l'algebra standard del DSM.

Il MIL data model consiste in un set estensibile di valori atomici e in una collezione, il BAT (Binary Association Table). L'algebra binaria definita per il BAT prevede le seguenti operazioni:

- Select vuol dire che hai una tabella con head e tail con due colonne, c'è una funzione  $f$  da applicare a tutti gli elementi della tabella e per ogni elemento della tabella dove  $b$  rispetta la condizione  $f$  allora restituiamo  $a$ . Il secondo valore viene consumato dal fatto che applichiamo la condizione. Qua la condizione è integrata all'interno di una funzione.

$$\begin{array}{l} \text{select}(\text{bat}[H,T] AB, \text{str } f, \dots p_i \dots) : \text{bat}[H, \text{oid}] \equiv \\ \quad \langle [a, \text{nil}] \mid [a, b] \in AB \wedge (*f)(b, \dots p_i \dots) \rangle \end{array}$$

Situazione tipica:  $a$  è il RID e  $b$  è il valore, quindi se vogliamo selezionare gli studenti che hanno una certa età dobbiamo preparare una tabella con  $\langle \text{Rid}, \text{Age} \rangle$  e la funzione controlla che Age sia maggiore di un certo valore e vengono presi solamente i rid che rispettano la richiesta.

- Binary Join: abbiamo due tabelle e in ogni tabella abbiamo due attributi, dobbiamo unire gli attributi in base alla richiesta della funzione  $f$ .

$$\begin{array}{l} \text{join}(\text{bat}[H_1, T_1] AB, \text{bat}[T_1, T_2] CD, \text{str } f, \dots p_i \dots) : \text{bat}[H_1, T_2] \\ \quad \equiv \langle [a, d] \mid [a, b] \in AB \wedge [c, d] \in CD \wedge (*f)(b, c, \dots p_i \dots) \rangle \end{array}$$

Funziona in questo caso come una tipica join, nella soluzione finale abbiamo solamente i RID delle due tabelle e non i dati.

- Reconstruct: con le due operazioni precedenti capita che vengono consumati alcuni valori, ad esempio se faccio la select abbiamo che la  $b$  non viene presa in considerazione nel risultato e lo stesso

discorso vale per la join perchè nella join restituisco solamente i rid delle due tabelle ma non i valori.

Se ad esempio voglio gli studenti che hanno un'età maggiore di 18 prima faccio la select e poi voglio ricostruire l'età e quindi faccio la join tra la tabella originale e la tabella A, nil che ho preparato.

Reconstruct è una forma di semi-join.

**reconstruct(bat[H,nil] AN, bat[H,T] AB) : bat[H,T]**  
 $= \langle [a,b] \mid [a,b] \in AB \wedge [a,nil] \in AN \rangle$

Esempio: Partiamo con la tabella AB dove abbiamo i rid e associato l'attributo Age. Vogliamo fare la select per selezionare solamente i rid che hanno una age corrispondente maggiore di 27. Quindi da 100 record iniziali magari ne prendiamo solamente 50 e sono nella forma [a, nil].

Ora vogliamo ricreare la tabella e formarne una che abbia la forma [RID, Name]. Utilizziamo l'operatore Reconstruct.

[a,b] viene messo nella soluzione se b è nella colonna dei nomi e se a è tra i risultati filtrati.

- Reverse: operazione che serve a invertire il primo attributo della tabella e il secondo elemento.

**reverse(bat[H,T] AB): bat[T,H]**

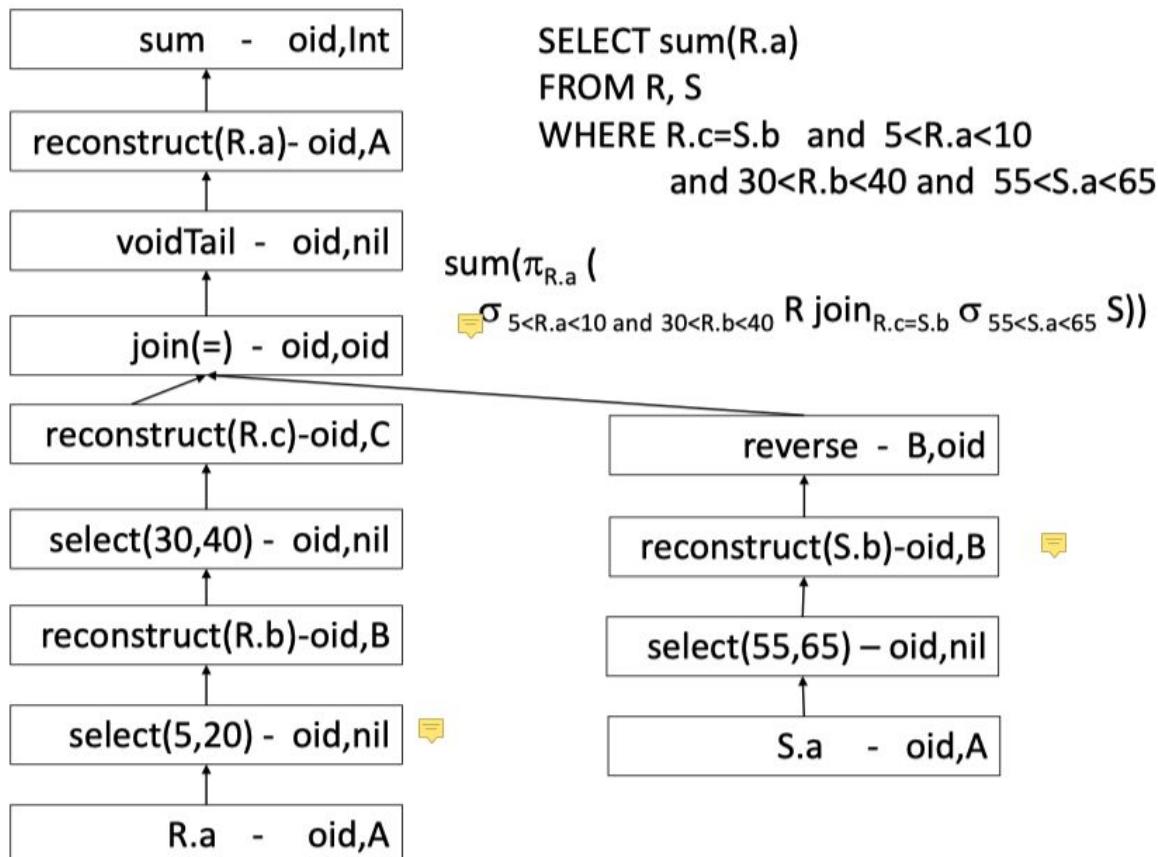
 $= \langle [b,a] \mid [a,b] \in AB \rangle$ 

- VoidTail: operazione che trasforma una tabella binaria in una tabella unaria. È un modo per poi usare la reconstruct dopo.

**voidtail(bat[H,T] AB): bat[H,nil] =  $\langle [a,nil] \mid [a,b] \in AB \rangle$**

- **GroupBy**: tipicamente viene implementato utilizzando l'hash
- **Sum**: somma tutti i valori di una tabella e memorizza il risultato

Esempio di utilizzo della Binary Algebra:



Nella query dell'esempio dobbiamo considerare due tabelle, R e S. In particolare dobbiamo fare due volte la select su R e una volta su S. Quindi nella parte sinistra facciamo la select di R.a e otteniamo delle coppie <RID, nil>, quindi va fatta la reconstruct in modo da ottenere coppie del tipo <RID, B> e su questa tabella che ottengo devo fare la seconda select e una reconstruct perchè per la join mi serve la R.c. Nella parte destra facciamo sempre la stessa cosa con la tabella S e la select ma poi alla fine viene svolta la reverse perchè nella join c'è la necessità di avere la prima tabella nel formato <RID, value> e la seconda nel formato <value, RID> quindi dobbiamo fare una reverse. Dopo la join poi si può tagliare la seconda parte della tabella e ottenere

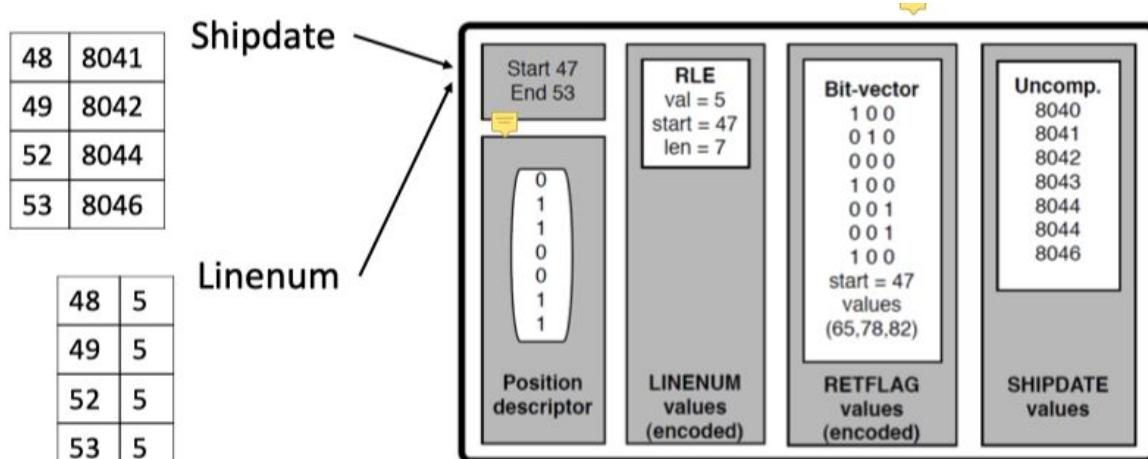
una colonna del tipo <RID, nil> con la voidtail e poi si fa la reconstruct in modo da ottenere quello che vogliamo nella select, ovvero la tabella R.a.

In un column database una tabella binaria non viene sempre implementata come binaria (coppia <id, valore>). Infatti gli id delle colonne possono essere rappresentati come un range e quindi possiamo avere un range e poi i valori associati.

I valori invece li possiamo comprimere, un esempio:

7	87	(7,10)	87	(7,10)RLE	87,1
8	89		89		89,3
9	89		89		
10	89		89		

Partendo da una tabella iniziale, il risultato della select possiamo considerarlo come una bitmap.



Ad esempio partiamo con la tabella shipdate e vogliamo selezionare solamente alcune righe. Quindi posso rappresentare questa select in una bitmap in cui mettiamo 1 nelle righe che devo considerare, 0 altrimenti. Quando devo poi fare una reconstruction parto dalla bitmap che ho creato ed eseguo la reconstruct prendendo in considerazione la tabella che voglio usare per la reconstruct. In realtà se voglio fare una reconstruct con linenum la faccio virtualmente.

Se vogliamo effettuare una join tra le colonne abbiamo un approccio che è differente rispetto alle tabelle classiche perchè qua evitiamo l'utilizzo degli indici, non possiamo fare l'index nested loop, quindi dobbiamo usare Hash o Merge Join oppure una join in main memory quando i dati entrano in memoria principale.

Il metodo più usato per fare la join però è il join index che è una struttura dati che mi rappresenta gli elementi della prima colonna che matchano un elemento nella seconda colonna.

Sono anche disponibili degli algoritmi che permettono di fare la join in presenza di un join index.

---

### Salta le slide che descrivono il Jive Join

---

Il Jive Join ha un costo che è praticamente lineare, deve solamente leggere e scrivere le colonne su cui vogliamo fare la join e l'indice e vengono create delle strutture date temporanee. Poi si fa una scansione delle strutture dati temporanee e poi la jive join è completa.

Perchè dovrei fare tutto questo, creando anche il join index se poi alla fine il costo è lineare come nel caso di un sort-merge?

Il costo è lineare nella outer table ma per la inner table è lineare nella dimensione dei record che sono stati selezionati mentre nel sort merge abbiamo un costo lineare che è lineare nella dimensione totale dei record.

Quindi quando abbiamo una join che è molto selettiva con il join index non faccio la scansione totale ma la scansione della sola parte dei dati che devo selezionare.

Questo Jive Join ci fornisce i vantaggi del Sort-Merge (considero il numero delle pagine e non il numero dei record) ma anche i vantaggi dell'index nested loop che mi dice che non devo fare una scansione della tabella intera ma solamente di una parte della tabella.

Per quanto riguarda l'operazione GroupBy da effettuare con i column database abbiamo due possibili scelte. Se l'input è ordinato allora viene

utilizzato un algoritmo basato sul sort, altrimenti con un input non ordinato si utilizza un algoritmo che utilizza l'hash.

Consideriamo degli inserimenti in un column database o un aggiornamento o una cancellazione di un dato.

Se stiamo usando un sistema pensato per OLAP potrebbe anche essere che queste operazioni non vengono nemmeno implementate perché potremmo essere in una situazione in cui l'utente vuole solamente analizzare i dati e non inserirne di nuovi.

Se siamo in un sistema che è allo stesso tempo OLAP e OLTP invece la situazione è differente e possiamo avere la possibilità di implementare anche queste altre operazioni. In questo caso le operazioni sono molto costose perché ho ridondanza quindi potrei dover aggiornare molte tabelle e allo stesso tempo potrei avere la necessità di decomprimere i dati che sono presenti nelle tabelle.

La soluzione per questo problema è spesso quella del differential file approach.

Questo approccio richiede che ognuno dei record abbiano uno specifico RID, vengono memorizzate due strutture un read store che contiene tutti i dati che sono stati inseriti ad esempio ad inizio giornata e poi un write store che invece contiene i dati che sono stati aggiunti durante la giornata o che sono stati eliminati/modificati.

Ogni volta che viene eseguita una query ci troviamo a dover fare una ricerca all'interno del read store e poi una ricerca all'interno del write store. Quando nel write store trovo dati nuovi li aggiungo al risultato del read store, quando vedo che i dati sono stati eliminati li elimino.

Come eseguo il merge di read store e write store quando ho delle query?

Ho due approcci possibili:

- Stop the word: questo è l'approccio più Olap oriented, faccio il merge del write store all'interno del read store
- Incremental approach: abbiamo un thread che ispeziona il write store muovendo i blocchi dal write store al read store.

Utilizzando questo approccio con Read Store e Write Store non abbiamo bisogno di utilizzare le lock quando svolgiamo una transazione perché in questo modo otteniamo gratuitamente uno snapshot isolation.

Posso usare il read store per creare lo snapshot e poi posso allocare un write store per ogni transazione e un “collective write store” che contiene tutte le write che vengono eseguite da transazioni committate.

In questo modo non abbiamo altri costi per isolare la snapshot isolation.

Posso anche fare un no-undo, no-log approach perché ora faccio il merge del write store con il collective store solamente dopo il commit, quindi antico il commit prima della write.

Ho bisogno di un redo-log che va a ricreare il collective write store.

I column store non hanno un indice perché è la stessa colonna che mi rappresenta un tradizionale indice di una tabella.

Attualmente stiamo andando nella direzione di sistemi ibridi OLAP e OLTP perché i column database sono adatti per la parte analitica mentre gli altri sono adatti per le inserzioni di nuovi dati.

Perchè non partiamo da un database tradizionale e non mettiamo un indice per ogni attributo?

Questo potrebbe essere la stessa cosa di un column store perché con un indice per ogni attributo abbiamo un indice per ogni colonna, se devo lavorare ad esempio su due attributi userei solamente operazioni index only.

Perchè abbiamo bisogno di un approccio completamente differente per sviluppare questo tipo di database?

In realtà avere un indice per colonna non è la stessa cosa di un column database. Avere un indice in una tabella è comodo per un solo attributo ma il problema è quando abbiamo bisogno di più attributi perché ora dovrò fare la join tra questi indici differenti.

In un column database c'è un grande lavoro per evitare le join del tutto oppure per avere un join index per fare in modo che la join sia ottimizzata. Senza queste ottimizzazioni la fase di reconstruction sarebbe molto costosa.

Inoltre avere un indice per ogni colonna vorrebbe dire che la cancellazione di un record diventerebbe molto costosa, il column store ha degli algoritmi per gestire questa situazione ma i db tradizionali no.

Un'altra possibilità che abbiamo, al posto di usare gli indici, possiamo riscrivere le colonne come un set di tabelle binarie con un attributo che mi simula il RID e poi memorizzo tutte le tabelle in modo sequenziale. Questo rende la join di differenti tabelle molto veloce perchè sono ordinate.

Questa è una simulazione migliore rispetto alla precedente ma comunque non ho tutti i vantaggi del column store.

Ad esempio non posso comprimere i dati che si trovano nelle colonne e questo è un grande vantaggio dei column database, poi abbiamo un overhead per memorizzare gli ID e lo spazio totale sarà il doppio rispetto a quanto sarebbe normalmente. Inoltre possiamo avere un alto numero di join e questo può confondere l'ottimizzatore.

Per concludere:

- I column database non sono solamente vertical partitioning
- I database tradizionali non sono ideali da utilizzare con macchine con molta RAM

## Lezione 14: Parallel and Distributed Systems

Esistono vari tipi di database:

- Parallel: alcune operazioni critiche possono essere parallelizzate, in questi sistemi quando abbiamo un fallimento tutto quanto va offline.
- Distributed: in questo caso le transazioni vengono distribuite su varie macchine e vogliamo minimizzare le comunicazioni perché se i nodi sono lontani per la comunicazione possiamo avere la necessità di un po' di tempo. In questo caso però non abbiamo problemi con i fallimenti perché se un nodo ha un problema cade solamente quello.
- P2P: sono sistemi multi CPU differenti dai distribuiti, non abbiamo in questo caso un singolo proprietario che gestisce il database. In questo caso non possiamo fidarci di tutti i nodi del sistema perché alcuni potrebbero essere maligni e potrebbero andare offline appositamente.

Per quanto riguarda il parallelismo abbiamo differenti modelli che possiamo prendere in considerazione, per prima cosa dobbiamo decidere in che modo gestire la CPU e la cache:

- Shared Memory: Ogni CPU potrebbe avere la sua cache e una RAM condivisa. Possono esserci problemi quando devo aggiornare la stessa pagina della RAM da più di una CPU.
- Shared disk machine: in questo caso le CPU hanno ognuna una memoria personale e poi il disco viene condiviso tra tutti
- Share nothing: ogni CPU ha una memoria e un disco e condivide un sottosistema di comunicazione.

Nella vita reale troviamo un mix di questi sistemi qua, ad esempio abbiamo 100 CPU che vengono divisi in blocchi dove per ogni blocco abbiamo una memoria condivisa tra i vari processori del blocco e che però non viene condivisa con i processori del blocco successivo.

Le comunicazioni in questi modelli paralleli avvengono mediante dei messaggi, una CPU prepara il messaggio, invia il messaggio alle altre CPU e queste le riceveranno. Posso inviare questi messaggi in modo sincrono (invio e attendo risposta) oppure asincrono (invio il messaggio e poi vado avanti, poi posso controllare se ci sono messaggi per me). Il punto principale è che i messaggi creano overhead perchè è un processo lungo e complicato e quindi non conviene fare tutto questo per inviare solamente un byte. Quando è possibile vogliamo cercare di produrre un messaggio più grande da inviare.

Il costo di inviare un messaggio è la somma della creazione del messaggio (costo fissato) e dell'invio del messaggio che è lineare nella dimensione del messaggio. Inviamo un messaggio se il costo dell'invio è nello stesso ordine di grandezza del costo per creare un messaggio.

Nel corso si assume un modello di parallelismo “Share nothing”.

La tecnica basilare per ottimizzare qualsiasi forma di parallelismo e anche di distribuzione consiste nel partizionare i dati nel modo corretto. La cosa basilare che dobbiamo decidere quando creiamo un sistema parallelo o distribuito è il modo in cui allochiamo i dati. Questo processo di data allocation e distribution è il risultato di due decisioni che dobbiamo prendere:

- Come partiziono i dati
- Come alloco le partizioni nei nodi

Il partizionamento dei dati viene effettuato tramite le seguenti strategie:

- Range: se voglio allocare i miei dati in 100 nodi posso scegliere un attributo per partizionare, ad esempio il salario e poi divido gli attributi in 100 range differenti, il nodo 1 si prende il primo range e così via. In questo caso quando abbiamo una query con un range abbiamo il pro che solamente una parte dei nodi sono presi in considerazione. Se invece voglio aumentare il parallelismo e fare una parte della query in ogni nodo, il range è la peggiore perchè ho alcuni nodi che lavorano e altri che non fanno niente, in questo caso la soluzione migliore sarebbe utilizzare un hash partitioning.

Un altro problema del range partitioning è che va fissato un range all'inizio.

- Hash: in questo caso abbiamo una hash function con un range che mi indica la quantità dei nodi presenti e poi applico l'hash in modo da assegnare i dati ai vari nodi
- Random: è come un heap partitioning, ogni dato lo prende un nodo a caso, è tipo round robin
- Block: è come il caso random ma in questo caso vado di blocco in blocco e non di record in record.
- Co-located partitioning: ho una relazione R organizzata con uno dei metodi precedenti, ad esempio la tabella studenti organizzata con l'hash. Ora se voglio dividere i dati anche della tabella esami, metto in ogni nodo i dati relativi agli studenti che sono in quello specifico nodo. È un sistema che è simile al semi join.

Questo tipo di divisione dei dati è simile alle organizzazioni dei dati (hash, heap, sequential).

Quando dobbiamo distribuire i dati attualmente non si fa una distribuzione in un numero fissato e massimo di nodi, ad esempio se ho 100 nodi non sempre divido in 100, spesso si fa una scelta differente e si usano meno nodi.

È anche tipico in questa fase l'utilizzo del vertical partitioning e dell'horizontal partitioning, ogni singolo frammento viene definito con condizioni e proiezioni.

Una volta che è stato deciso come partizionare i dati possiamo anche definire come allocare i vari frammenti. Ad esempio se utilizziamo un range partitioning possiamo decidere che il frammento 1 verrà memorizzato sia sul nodo 1 che sul nodo 101, il frammento 2 sul 2 e sul 102. Possiamo in questo modo avere 3 approcci:

- Un frammento è solamente su un nodo
- Fully replication: ogni frammento è memorizzato in ciascun nodo, questo viene usato se abbiamo tabelle piccole e le query riguardano tutta la tabella, in questo modo evitiamo di mandare in

giro dati quando ho bisogno di fare la query. Questo serve per velocizzare le query.

- Possiamo avere una replication parziale, magari copio il frammento due o tre volte. Questo tipo di replication viene usato per garantire la resilient, in una situazione distribuita o parallela vogliamo essere capaci di continuare a lavorare anche se abbiamo un fallimento, con le copie dei dati possiamo continuare a lavorare anche se un nodo va offline. Tipicamente abbiamo 3 copie del dato perchè in questo modo quando abbiamo due copie che sono differenti tra loro, con la terza copia possiamo decidere quale valore sarà quello giusto.

I motivi per cui viene fatta questa divisione dei dati:

- Potremmo avere 100 cpu e vogliamo lavorare in parallelo in modo da svolgere le query 100 volte più velocemente
- Possiamo partizionare utilizzando il range partitioning in modo da utilizzare solamente i nodi su cui sono i dati per la query mentre sulle altre CPU possiamo fare altre operazioni.

Consideriamo le operazioni sui dati che possiamo fare quando ci troviamo ad operare in parallelo:

- L'operazione distinct è banale se utilizziamo l'hash per distribuire i dati sui nodi della rete. In questo caso ogni nodo va a fare il distinct eliminando i duplicati. Se però utilizziamo un altro metodo per fare la distribuzione dei dati dobbiamo fare due operazioni:
  - Per prima cosa ogni nodo fa il re hash dei dati e quindi i dati vengono distribuiti di nuovo tra tutti i nodi
  - Poi ogni nodo può svolgere il distinct localmente eliminando eventuali duplicati

Questa soluzione è più veloce della scansione del file perchè ogni nodo esegue l'hashing e se ad esempio abbiamo 10 nodi e 1000 pagine, ogni nodo leggerà solamente 100 pagine.

Questo pattern è tipico per il distinct

- Per quanto riguarda le operazioni di Unione di due tabelle, intersezione e differenza l'idea di base è la stessa.

Se le due tabelle sono hashate con la stessa funzione possiamo fare questa operazione localmente.

Ad esempio se abbiamo la tabella R e la S, con la prima che è hashata con una buona funzione e la seconda che non è hashata bene, posso hashare solamente la seconda.

Questo procedimento di re hashing viene fatto cercando di sfruttare al massimo il buffer del processore, teniamo infatti un numero di pagine in memoria pari al numero di nodi e poi quando faccio il rehash metto in ogni pagina i dati che devono finire in un nodo. Quando la pagina viene riempita allora invio la pagina piena al nodo destinatario.

- Join ( $R(x,y), S(y,z)$ ): in questo caso viene fatto il rehashing delle tuple di R e di S basandoci sull'attributo y e usando la stessa funzione hash. In questo modo abbiamo nei nodi i dati che vanno uniti nella join. Poi la join viene eseguita localmente.
- GroupBy( $R, x, \{f_1, \dots, f_n\}$ ): in questo caso l'idea è che prima distribuiamo la relazione tra tutti i nodi usando l'attributo X oppure usiamo una distribuzione con il range che dipende da X. Poi successivamente possiamo eseguire il groupBy in ognuno dei nodi
- Anche per filter e per la projection possiamo eseguire l'algoritmo localmente

## Join Algorithm

- Se R e S sono co-located allora possiamo fare una join node by node.  
Con co-located intendiamo che viene utilizzata la stessa hash function su X e su Y oppure che viene usato un approccio con il range sull'attributo X e Y oppure che R viene distribuito con un qualsiasi approccio e poi facciamo la co-location di S a seconda dell'attributo della join.  
In questo modo R e S sono co-locati in modo che si possa fare la join sull'attributo X e Y.
- Directed Join: abbiamo R e S che vengono partizionati basandoci sullo stesso principio, ad esempio lo stesso range. I dati su cui

vogliamo fare la join però potrebbero essere su due nodi differenti, quindi, se vogliamo mantenere R nel nodo in cui si trova possiamo spostare i dati di S nel nodo di R e fare la join.

- Repartitioned join: R ed S non sono distribuiti usando lo stesso metodo. Per fare la join dobbiamo ripartizionare entrambi usando lo stesso criterio.

La ripartizione viene fatta in parallelo, se la mia tabella è partizionata in 10 nodi allora 10 nodi faranno in parallelo la ripartizione e ognuno invierà agli altri nodi i dati. Il fatto che il repartitioning sia svolto in parallelo è importante per un discorso di performance.

Possiamo anche partizionare una sola tabella se ad esempio R è già partizionato in base all'attributo su cui faccio la join.

Possiamo scegliere quale tabella ripartizionare tra R ed S.

- Broadcast join: se una delle due tabelle su cui va fatta la join è piccola allora possiamo inviarla in broadcast a tutti i nodi.

Abbiamo due metodi per misurare le performance di un algoritmo:

- Elapsed time: avvio l'algoritmo, questo viene eseguito su quanti più nodi possibili, poi l'algoritmo termina e smetto di contare il tempo che è passato. Se l'obiettivo è ottimizzare questo tempo devo cercare di avere un alto parallelismo.
- Total time: consideriamo il tempo totale per cui le varie CPU eseguono calcoli.

Il modo migliore per il calcolo delle performance dipende dalla situazione in cui ci troviamo. Molto spesso siamo anche in situazioni intermedie, ad esempio potrei avere una macchina con 100 core e lavorano vari task in parallelo, vorremmo avere un elapsed time basso, se però per avere un miglioramento della speedup di due volte ho un costo 10 volte superiore però non mi conviene proseguire con un approccio di questo genere e quindi devo cercare di fare una media tra il miglioramento della speedup e i costi.

## **Se consideriamo l'operazione di Join.**

Tipicamente ho bisogno di eseguire un ripartizionamento in base ad una

certa join, questo ripartizionamento non sarà però ottimizzato per una successiva join che voglio effettuare.

Il costo del ripartizionamento: se abbiamo 100 nodi ogni nodo deve leggere la sua partizione di R e S e deve calcolare l'hash function sulla tuple. Poi ogni nodo invia le tuple agli altri nodi.

Se abbiamo un file distribuito con 1 milione di pagine leggiamo 1 milione di pagine e la maggior parte di queste pagine verranno inviate.

Il costo sarebbe di  $N_{\text{pagine}} \text{ lette} + N_{\text{pagine}} \text{ inviate}$  ma questo avviene in parallelo quindi in realtà diventa  $N_{\text{pagine}}/P$  dove P è il grado di parallelismo.

Quindi:

- Costo per leggere in ogni nodo e fare il rehashing:  $(N_{\text{Page}}(R) + N_{\text{Page}}(S))/P$

Tutti i nodi più o meno inviano agli altri nodi  $N_{\text{pages}}/P$  pagine e ricevono  $N_{\text{Pages}}/P$  pagine. Tra tutti questi tempi che consideriamo (lettura, invio e ricezione) non abbiamo una risposta generale che mi dice quale di questi tempi è il maggiore. Inoltre per l'invio e la ricezione delle pagine abbiamo anche un costo differente in base al tipo di hardware che stiamo utilizzando.

Nella vita reale possiamo considerare che potrei trovarmi in una situazione in cui l'accesso al disco è molto più lento dell'invio dei dati e quindi in questo caso faccio così:

- Leggo la prima pagina del disco
- Inizio a leggere la seconda e poi nel frattempo inizio a inviare la prima pagina
- Vado avanti così fino a che non leggo tutto

In questo modo in parallelo faccio tutto e il tempo che realmente passa è solamente quello per la lettura dal disco, tutte le altre operazioni più veloci le faccio in parallelo. Tipicamente sommiamo i tempi perchè è il modo più semplice ma non sempre avviene così in realtà.

Se l'invio dei dati ha un tempo che è simile alla lettura dal disco allora il parallelismo sarà limitato e quindi la somma va abbastanza bene.

C'è un'altra cosa da considerare, calcolo il tempo come se io leggessi i dati, li inviassi agli altri e ricevessi i dati ma tipicamente non avviene così

perchè la comunicazione è sincrona e quindi la storia è molto più complicata e solitamente ogni nodo fa un lavoro in stile round robin per prendere i dati e inviarli a qualcuno. Quindi in pratica se l'invio e la ricezione richiede 10 millisecondi, non è vero che in 10 millisecondi finisco perchè c'è un overhead.

Quando viene effettuato il repartitioning per la join abbiamo due possibilità:

- Il fragment di una delle due tabelle è abbastanza piccolo da entrare nella memoria principale del nodo su cui lavoriamo. In questo caso possiamo eseguire una hash join a costo 0. In questo senso non abbiamo un overhead ma anzi riusciamo anche a migliorare il tempo di esecuzione perchè non eseguiamo una join in 2 fasi ma ci basta una sola fase per eseguire la join.
- Non sempre è possibile il primo caso, se il fragment non entra in memoria dobbiamo memorizzarlo sul disco e poi eseguiamo una hash join o un sort merge utilizzando l'algoritmo che costa 4 volte il numero delle pagine perchè qua stiamo assumendo un approccio 1 operatore alla volta. Hash Join e le altre costano  $2^*N_{pages}$  se sono in un modello pipeline mentre è  $4^*N_{pages}$  nel caso 1 operator at a time.

Esattamente come per il column database anche in questo caso non è sempre ovvio che il 1 tuple at a time è migliore del 1 operator at a time approach. Ci sono motivi per preferire uno o l'altro.

Nei sistemi paralleli solitamente preferiamo l'approccio 1 operatore at a time perchè è più complicato utilizzare il tuple at a time approach.

Un altro possibile approccio è una variazione di questo in cui il risultato intermedio viene memorizzato nella memoria principale.

Il sistema che usa questo approccio è più veloce ma allo stesso tempo è più complicato programmarlo perchè nel codice devo anche considerare cosa succede quando i dati intermedi non entrano in memoria principale.

Nel caso della join il tempo totale che è necessario per eseguire l'operazione è pari al tempo totale nel caso sequenziale diviso per il numero di nodi che eseguono in parallelo l'operazione.

Una cosa che può fare la differenza è la dimensione delle pagine che stanno nei nodi, se queste entrano in main memory allora riusciamo ad abbassare il numero di operazioni I/O da effettuare e questo ci porta grandi miglioramenti.

Una cosa che invece può essere un problema è il modo in cui sono distribuiti i dati, se abbiamo una distribuzione che non è uniforme nei vari nodi siamo rovinati perché i nodi devono ricevere più o meno tutti la stessa quantità di dati.

## Distributed Databases

Consideriamo i sistemi distribuiti, questi possono essere simili ai sistemi paralleli in cui le varie CPU adottano una strategia share nothing.

Le differenze tra questi due sistemi sono le seguenti:

- Nel sistema parallelo i costi di comunicazione sono praticamente nulli, specialmente se svolgiamo degli accessi al disco che hanno un costo maggiore rispetto al costo di comunicazione. Nei sistemi distribuiti invece il costo di comunicazione può essere molto più alto rispetto ad un accesso al disco
- Nei sistemi distribuiti i fallimenti sono normali, abbiamo un sistema con tanti nodi e in ogni momento uno di questi nodi può andare offline. Notare che il fatto che il fallimento dei nodi è indipendente (nel senso che se cade un nodo gli altri continuano a lavorare) è un aspetto positivo perché il sistema in questo modo non si ferma mai perchè abbiamo sempre la maggior parte dei nodi che sono sempre attivi e lavorano. Se abbiamo degli algoritmi che quando un nodo fallisce sono in grado di spostare il lavoro di quel nodo su un altro nodo allora abbiamo un sistema che è failure resilient.
- Un altro problema è il partizionamento, abbiamo un problema quando ci sono problemi con la rete (network failure). Il sistema in questo modo viene suddiviso in due parti che continuano a funzionare e dopo un po' i due sistemi vengono riconnessi e devi

considerare quello che è successo nel mezzo. Non ci sono soluzioni ottimali per questo problema però ci sono molte tecniche che possono essere utilizzate per risolvere il problema.

- Alcuni sistemi distribuiti sono federati e altri sono P2P. I sistemi distribuiti federati sono quelli che abbiamo ad esempio quando due aziende si fondono, in questo caso abbiamo due database e costruiamo un database distribuito e federato in cui hai due sistemi che continuano ad adottare degli algoritmi differenti per gestire i vari problemi distinti.

Nel caso del P2P non solo abbiamo nodi differenti che possono usare politiche differenti e algoritmi ma abbiamo anche un modello differente di fiducia verso i nodi. I nodi della rete P2P possono scomparire e in alcuni casi possono anche essere malevoli e fare in modo di distruggere la coerenza del sistema.

Anche nel caso dei sistemi distribuiti c'è da capire in che modo distribuire i dati e come allocare le partizioni dei dati all'interno dei nodi della rete.

Come allocare le partizioni dei dati nei nodi dipende dalla topologia della rete e dalla distribuzione. Il partizionamento dei dati possiamo farlo in orizzontale o in verticale, ad esempio potrei avere un partizionamento sulla base della geografia e a seconda dell'attributo nazione potrei voler suddividere i dati in europa e in america.

Prima viene deciso come partizionare i dati e poi si decide come allocarli, quando si parla di allocare i dati si deve anche considerare il numero di copie che dobbiamo fare dei dati che andiamo a memorizzare nei vari nodi.

Questa necessità di replicare i dati è dovuta a due motivazioni, la prima è la resistenza ai fallimenti e la seconda è la velocità delle query che, grazie alla ridondanza diventano anche più veloci, allo stesso tempo ci troviamo con gli update del database che invece diventano molto più lenti.

Assumiamo una tabella che non aggiorna mai i dati ed è spesso letta, il costo per comunicare è alto, se però replichiamo i dati allora non

dobbiamo spostare di continuo i dati e quindi abbiamo un costo più basso per la comunicazione e query più veloci.

Durante la fase di distribuzione dei dati possiamo fare varie scelte per partizionare i dati e in alcuni casi posso fare la mia scelta anche in base a quello che poi farò durante la fase di allocazione.

Ad esempio potrei voler decidere di frammentare molto i miei dati e dividerli in 100 parti per poi andare a unire i dati in gruppi formando solamente 10 blocchi nel momento in cui faccio l'allocazione dei dati nei singoli nodi.

Ci sono due approcci possibili:

- Primary copy: in questo caso abbiamo un frammento primario che viene copiato mentre il resto dei dati sono delle copie
- P2P: in questo caso ogni frammento ha la stessa importanza degli altri

Frammentare non è difficile mentre l'allocazione è un problema di ottimizzazione che è anche difficile, si deve considerare quante copie devono essere fatte e questo andrà ad incidere sul bilanciamento tra la velocità di lettura e di update. Qualsiasi scelta che facciamo avrà anche un effetto sulle operazioni che verranno fatte in futuro.

Se sappiamo che in questo database verranno fatti pochi aggiornamenti allora abbiamo tante copie del fragment, se invece vogliamo fare tanti update allora avremo poche copie.

C'è anche un secondo criterio che possiamo considerare, se sappiamo che dei fragment vengono uniti in una join molto spesso allora cerchiamo di fare in modo che questi frammenti finiscano nello stesso nodo.

### **Join nel modello distribuito**

Nel caso in cui abbiamo un sistema distribuito stiamo assumendo che il costo dominante sia quello di comunicazione. Se vogliamo fare una join tra due tavole, R ed S ci sono due possibilità, o mando R al nodo dove sta S o viceversa.

In alternativa, se invece di avere tutta la tabella in un nodo abbiamo i dati frammentati e quindi ci troviamo ad esempio con una tabella che è divisa in 100 nodi abbiamo comunque problemi per quanto riguarda la join, abbiamo la possibilità di inviare i dati da un nodo all'altro in modo da eseguire comunque la join.

Tipicamente si tende ad inviare la tabella che è più piccola in modo da inviare meno dati.

C'è anche una terza possibilità che è chiamata semijoin reduction.

Assumiamo di voler fare la join tra la tabella studenti e la tabella esami sull'attributo studentID. In un sistema distribuito quello che si fa è proiettare il campo su cui facciamo la join fuori dalla tabella R e inviare questo campo al nodo in cui è memorizzata la tabella S.

Sul nodo dove sta la tabella S calcoliamo la semijoin tra i dati che abbiamo ricevuto e la tabella locale.

Ad esempio se vogliamo la join considerando gli studenti del primo anno e gli esami dove grade=30.

In un sistema distribuito facciamo così:

- Estraiamo il campo studentID da esami dove grade=30 e mando questo dato al nodo dove è memorizzata la tabella studenti
- Sul nodo studenti calcolo la semijoin tra la tabella che è arrivata e studenti.
- Poi rimando indietro al nodo dove sta esami non tutti gli studenti ma solamente quelli che ho selezionato utilizzando gli esami e nel nodo esami verrà fatta l'ultima join tra questo dato e la tabella esami.

Il risultato è lo stesso della join completa ma i dati che vengono inviati tra i nodi sono minori perché nella prima fase comunico solamente la proiezione dell'attributo che è stato anche filtrato con grade=30, nella seconda fase invio solamente una versione filtrata di S.

Questa è una ottimizzazione molto importante, invece di inviare tutti i dati invio una proiezione, calcolo la semijoin, invio indietro il risultato della semijoin e poi io calcolo la join.

Questo è qualcosa che in main memory non la farei mai ma quando il costo che domina è il costo di comunicazione allora è una buona idea e posso veramente avere un miglioramento delle performance.

È importante capire se effettivamente fare questa semijoin mi riduce il numero di dati che vengono inviati.

Ci troviamo in un sistema distribuito, una transazione quindi ora è un processo distribuito, ogni nodo è un database ed è in grado di garantire la serializability, la durability e l'atomicity.

Vorrei fare in modo che l'intero sistema sia serializzabile, atomico e abbia la durability.

Il problema per avere un sistema di questo genere riguarda il modo in cui vengono implementati i committ distribuiti e la serializability.

Inoltre è anche fondamentale considerare la consistenza dei dati che vengono replicati, assumiamo di avere una partizione dei dati, il sistema è diviso in due parti che vanno avanti in parallelo, possono prendere direzioni differenti e possono esserci divergenze.

Ci sono due approcci possibili:

- Posso evitare le divergenze rendendo uno dei due incapace di modificare i dati
- Posso permettere a entrambi le parti di divergere e poi di avere comunque una consistenza finale

Quindi o possiamo avere due volte una primary copy oppure diamo la possibilità di avere due copie che però sono uguali tra loro.

**Distributed commit problem:** ci troviamo in un sistema in cui abbiamo un client che invia al venditore una richiesta, entrambi mandano una richiesta alla banca per coordinare il pagamento. Questo è un sistema tipico e vorremmo essere in grado di aggiornare in modo atomico tutti questi database, sia lo stato del venditore che lo stato della banca, vogliamo evitare che per un ordine non trovo poi i soldi in banca.

Questa è una situazione tipica in un sistema federato ma possiamo averla anche in un non federato ad esempio abbiamo una banca che muove i soldi da un account ad un altro e i due account sono posizionati su due nodi differenti, vogliamo evitare che in caso di fallimento i soldi scompaiano da un nodo e poi non ricompaiano nell'altro.

Questi problemi sono simili tra loro perché in un sistema federato ho meno libertà durante lo sviluppo dell'algoritmo.

Il metodo canonico per risolvere questo problema è il canonical commit algorithm che è implementato da qualsiasi sistema distribuito.

Quando siamo in un contesto distribuito abbiamo delle transazioni distribuite e quindi c'è la necessità di avere un sistema di commit distribuito che mi garantisca l'atomicity.

Quindi ad esempio se ho tanti negozi e voglio spostare qualcosa da ogni negozio ad un unico centro io potrei fare la richiesta ad ogni negozio e poi considero che ogni negozio mi spedisce tutto quello che ho chiesto.

Se però il negozio ha meno di quanto chiedo magari vorrebbe poter rinunciare alla spedizione e in questo caso io che ho già aggiornato il conteggio dei prodotti ricevuti non ho più una consistenza e non garantisco neanche l'atomicity durante tutto il procedimento perché potrei avere effetti parziali nel database. Quindi la soluzione è implementare un protocollo di tipo 2 phase commit.

In un 2 phase commit situation abbiamo tanti nodi che sono in grado di eseguire la loro parte di transazione e poi un nodo che agisce da coordinatore, ogni nodo ha il suo log locale e tutti i messaggi che sono mandati in questo protocollo (basato sullo scambio di messaggi tra nodi) sono log.

Come funziona questo algoritmo (esempio per mettersi d'accordo su un meeting):

- C'è un coordinatore che chiede se una certa data e un posto va bene a tutti
- Se tutti i vari partecipanti dicono che quel giorno va bene e rispondono al coordinatore, per me che ho risposto positivamente quella data viene segnata come "pre-committed".
- Il coordinatore ha due possibilità o tutti hanno risposto e hanno detto che va bene, in questo caso finisce la prima fase del 2 phase commit. Ora inizia la seconda fase dell'algoritmo dopo che tutti hanno detto ok, qua il coordinatore conferma a tutti che la data e il posto sono stati scelti. Dato che tutti hanno fatto il pre-commit della risposta niente potrà andare male. Se invece uno aveva risposto no dovrà ricominciare dall'inizio e tutti torneranno allo stato iniziale e lasceranno lo stato di pre-commit. Un nodo è abbastanza per l'abort.

- Dopo che ho mandato il messaggio della seconda fase il coordinatore attende che tutti mandano un ack per dire che hanno ricevuto la conferma del giorno e del posto. Se non ricevo niente dopo un certo timeout invio di nuovo la mia risposta.

Descrizione formale dell'algoritmo:

- Il coordinatore scrive nel suo log di preparare il messaggio e inizia la fase 1
  - Il coordinatore invia a tutti i partecipanti il messaggio che ha creato e chiede "siete in grado di committare il messaggio e muovervi nello stato ready?"
  - Tutti i partecipanti devono rispondere positivamente o dicendo che non sono in grado di committare (se uno dice di no mi fermo e ricomincio). Se non sono in grado di andare avanti allora ci sono da fare due cose, prima passano nella fase di pre-commit e scrivono nel log che sono in questa fase in modo che se ci dovesse essere un restart allora avrei comunque memorizzato questa informazione. Dopo che ho scritto questo nel log rispondono al coordinatore e il coordinatore in questo modo capisce che quel nodo è nel pre-commit state.
  - Poi il coordinatore quando riceve tutte le risposte può far partire la fase 2 e passiamo quindi in uno stato di commit.  
Allo stesso modo il coordinatore potrebbe anche decidere di fare un abort di questa comunicazione e in questo caso manda un messaggio di abort a tutti i partecipanti e deve anche scrivere nel suo log che ha preso questa decisione, la decisione non è più modificabile poi.
- Quando il partecipante riceve il messaggio di abort lo deve scrivere anche nel suo log.

Assumiamo che la decisione di abort del coordinatore dovesse essere ritrattabile, in questo caso abbiamo un abort che viene inviato a tutti i partecipanti e poi il coordinatore crasha. Nel momento in cui il coordinatore torna online prende il log e decide di fare il commit, il coordinatore non sa a chi ha mandato l'abort quindi potrei avere dei nodi che ora ricevono il nuovo commit e dei

nodi che invece hanno ricevuto il commit. Quindi avrà alcuni nodi che lavorano e altri che invece, avendo ricevuto l'abort non fanno niente.

Il coordinatore quando invia la decisione finale (abort o commit) ai partecipanti non si aspetta un ack, ogni partecipante è in una situazione di attesa prima di ricevere il messaggio e ha un timer, quando scade il timeout viene inviata una richiesta al coordinatore e allora il coordinatore risponderà inviando il messaggio una seconda volta.

Il partecipante non smette di chiedere al coordinatore cosa deve fare fino a che non riceve una risposta perchè il partecipante si trova in una situazione di attesa dove non può fare il commit o l'abort da solo, è bloccato e tutte le risorse che sono state acquisite vengono tenute ferme in attesa del messaggio. Se la transazione ha qualche lock non posso nemmeno rilasciare le lock. Dato che il coordinatore sa che se il partecipante manda un messaggio se non riceve il messaggio allora non ha bisogno di ricevere un ack, in ogni caso si assume che solamente una piccolissima parte dei messaggi è persa.

Consideriamo due scenari:

- Con l'ack il coordinatore attende N ack, se ho N partecipanti abbiamo 200 messaggi in circolo, più un altro per recuperare il messaggio perso
- Se invece non ho l'ack e solamente l'1% dei messaggi viene persi allora ne ho 100 inviati per l'abort e poi ne ho 2 per recuperare gli ultimi dati che ho perso.

La versione con i 2 ack richiede 2 volte il numero dei messaggi ma allo stesso tempo è anche vero che quando un partecipante non riceve un messaggio è difficile capire che un messaggio non è stato veramente ricevuto. Il partecipante ha bisogno di un timeout più lungo perchè il coordinatore risponde a tutti solamente dopo che tutti sono nella fase di pre-commit, quindi prima di sollecitare in realtà è meglio aspettare 4 volte il timeout.

Per il coordinatore è facile capire se il partecipante non risponde, per il partecipante invece è complicato capire se il coordinatore è

online o no.

Anche i partecipanti non possono modificare la loro decisione dopo che sono passati nello stato di pre-commit.

Nel caso del commit la situazione è molto simile, assumiamo che il coordinatore ha inviato il messaggio di commit a tutti quanti i partecipanti, questa decisione non è revocabile e viene scritta nel log (quindi il coordinatore scrive due cose nel log e il partecipante una se dice che non è disponibile, altrimenti due).

Ogni partecipante poi quando riceve il commit entra in una seconda fase in cui lavora.

Noi vogliamo anche provare che ogni nodo arriva poi in una situazione in cui termina il lavoro che sta svolgendo assumendo che il nodo viene riavviato nel momento in cui crasha e che i crash sono poco frequenti.

Per quanto riguarda la possibilità di recuperare lo stato corretto del sistema dopo un crash dobbiamo considerare che il protocollo è robusto al 100% perchè indipendentemente da quando crasha un nodo, questo viene comunque riportato in vita e si troverà di nuovo nella situazione in cui era in precedenza perchè tutto viene scritto all'interno del log.

Il log contiene tutte le decisioni che non possono essere revocate e quindi mi garantisce la correttezza del protocollo perchè quando faccio il restart svolgo di nuovo le stesse operazioni e torno in uno stato corretto. C'è una cosa meno chiara riguardo al protocollo, è la proprietà di liveness.

Quando si sviluppa un protocollo distribuito abbiamo un set di regole che mi dicono in una situazioni “multi agente” quali sono le cose che ogni agente può effettuare. Solitamente diamo molta libertà ai vari agents e quindi partendo dal protocollo possiamo pensare a tante differenti implementazioni. Quando pensiamo alla correttezza del protocollo dobbiamo dimostrare non la correttezza delle singole implementazioni ma la correttezza del protocollo che quindi è più astratto.

In un protocollo distribuito quindi devo dimostrare correttezza e liveness.

La correttezza mi dice che il protocollo non produrrà una situazione inconsistente all'interno del sistema.

Provare la liveness mi indica che non è mai il caso che il protocollo continua ad andare avanti senza arrivare ad un risultato.

In alcuni casi la liveness richiede che i nodi siano in grado di eseguire un restart dopo che sono andati offline e che stiano offline solamente per un periodo limitato di tempo.

Parlando sempre di fallimenti consideriamo anche i messaggi che vengono inviati all'interno del sistema.

L'idea basilare che riguarda i messaggi è che ogni messaggio può essere duplicato, può essere perso e non è mai un grande problema perché quando sono in attesa di un messaggio posso chiedere al coordinatore di inviare il messaggio nuovamente.

Tipicamente se dopo un numero di iterazioni non ottengo risposte allora possiamo assumere che destinatario è offline e quindi facciamo quello che ci dice di fare il protocollo, questo dipende molto da chi è andato offline. Se un partecipante è offline non è un problema perché se va offline nella prima fase il coordinatore da un messaggio di abort, se va offline nella seconda fase invece il coordinatore si limita a dire che lui ha mandato abort o commit a tutti perché il coordinator sa che quando il partecipante verrà riavviato, se questo è arrivato già in una fase di abort o di commit non ci sono problemi altrimenti se il partecipante scopre che è ancora nello stato di attesa allora sarà lui a contattare il coordinator. Il coordinator a questo punto o risponde con un abort/commit o in alternativa può dire al partecipante che sta attendendo ancora la risposta da un altro partecipante e quindi aspetta.

Quindi per il coordinatore il fatto che un partecipante è offline non è importante. Per quanto riguarda il contrario la situazione è diversa, se il partecipante è ancora nella prima fase e il coordinatore è offline allora il partecipante può inviare un messaggio dicendo che vuole fare un abort, il problema ce l'abbiamo quando il coordinatore è down dopo che il

partecipante ha detto di essere pronto (quindi è nel ready state) ma prima di aver ricevuto un messaggio di abort o di commit.

Il restart dopo un crash è sempre guidato dai log, cosa succede dopo un crash:

- Assumiamo che un partecipante si riavvia dopo un crash, se il precedente record nel log è un commit o un abort allora è semplice, con un abort facciamo un undo delle operazioni e se è un commit allora facciamo un redo delle operazioni della transazione.
- Se non trovo record alla fine della transazione, o ready o commit o abort, lo considero come un abort, così come succede nel classico protocollo undo-redo
- L'ultimo caso possibile è quello in cui trovo dopo il restart il log ready non ho altre possibilità che contattare il coordinatore.

In alternativa, se chiedo al coordinatore e lui non risponde perchè è offline allora potrei chiedere agli altri partecipanti. Quindi quando sono veramente bloccato è quando non solo il coordinatore è andato offline ma anche i partecipanti non rispondono perchè non hanno ricevuto un messaggio dal coordinatore ma sono comunque in stato ready (oppure contatto nodi che sono nella mia parte di rete che non hanno ricevuto il messaggio ma altri invece sanno la risposta e io comunque devo attendere fino a che la rete non si riconnette).

Riportare online il coordinatore è meno problematico e abbiamo tre possibilità, se non siamo in nessuno di questi tre casi invece non faccio niente.

- Se l'ultimo record che trovo nel log è prepare abbiamo varie possibilità, potrei ad esempio aver inviato in precedenza i messaggi a tutti e non aver ricevuto risposta, oppure potrei aver inviato la metà dei messaggi o magari ho inviato messaggi e ricevuto qualche risposta. Quello che posso fare è o rimandare il messaggio di prepare e tutti quanti i partecipanti invieranno

nuovamente la risposta, in alternativa (scelta più comune) il coordinatore può entrare nello stato abort mandare a tutti un abort.

- Se l'ultimo record del log è abort allora abbiamo due possibilità o rimando il messaggio di abort a tutti i partecipanti o non faccio niente. Non è un problema non fare niente perchè comunque i partecipanti mi contattano se dopo un po' non hanno notizie e sono bloccati.
- Anche se l'ultimo record è un commit ho comunque due possibilità o invio di nuovo la decisione a tutti o non faccio niente. Posso non fare niente perchè comunque vengo sollecitato se non ci sono messaggi che arrivano dal coordinatore al partecipante.

Se non faccio niente dopo il restart devo comunque essere preparato a ricevere un messaggio dai partecipanti e quindi devo essere in grado di andare velocemente nel log per capire l'ultimo record.

Se sono un coordinator e vado offline e torno online e trovo che l'ultimo record è un commit, cosa devo fare?

Posso inviare di nuovo il messaggio a tutti quanti oppure posso non fare niente e attendere che il partecipante mi contatti e chieda cosa deve fare perchè è bloccato.

Non è una situazione semplice da considerare, non so bene quanti messaggi sono stati inviati prima del crash e quindi non posso dire subito quale dei due metodi è migliore.

In termini di messaggi inviati è meglio una situazione in cui non invio nulla.

Potrei essere però in una situazione intermedia, potrei dire che per l'ultimo commit invio i messaggi ma per tutti gli altri no oppure potrei controllare quali altri log record ho dopo il log del commit.

Quando considero questo problema sono importanti da considerare anche le lock che vengono prese dalle varie transazioni, la migliore situazione in questo caso è il resend dei commit in modo che chi ha la lock possa subito ricominciare a lavorare.

Per decidere la soluzione migliore mi servirebbe:

- Un cost model per i messaggi che devo mandare e per le lock

- Un modello matematico che mi indica la probabilità che un messaggio sia stato inviato prima del crash.

Nella maggior parte dei casi si segue l'istinto durante l'implementazione e si sceglie di non fare niente perchè l'implementazione è più semplice.

Riassumendo possiamo dire che la 2 phase commit è un sistema interessante ma può funzionare solamente se sono sicuri al 100% che il coordinatore va offline e torna online in un tempo che è molto breve. Un altro modo per risolvere questo problema è utilizzare nella rete tre coordinatori che lavorano sincronizzati.

Per implementare questo c'è la necessità di essere sicuri che i vari coordinatori sono sincronizzati tra loro.

Possiamo rilassare questa richiesta chiedendo che di tre coordinatori la maggioranza sia sincronizzata, poi ci sono anche tante altre variazioni per implementare questo protocollo.

Implementare la transazione vuol dire implementare la durability, la resistenza ai fallimenti e il controllo della concorrenza.

Per la resistenza ai fallimenti partiamo da nodi che implementano questa feature, per l'atomicity locale e la durability sono utilizzati i classici database e poi queste feature sono estese all'intero sistema.

Questo 2 Phase commit è implementato in qualsiasi DBMS e ognuno ha un'interfaccia per ricevere messaggi e per partecipare alle transazioni distribuite.

Il problema in questo protocollo è che se il coordinatore va offline dopo aver preso la decisione ma prima di aver mandato la risposta ogni partecipante è fermo con la lock e nessuno può andare avanti, neanche spegnendo e riaccendendo il partecipante risolve il problema perchè dopo il restart sono di nuovo nella stessa situazione.

È un problema anche eleggere un nuovo coordinatore perchè se mentre ho scelto il nuovo coordinatore torna online il secondo potrei avere due coordinatori separati e magari uno fa una scelta uno un'altra e iniziano a mandare in giro due messaggi diversi. Allora se devo scegliere il nuovo coordinatore devo anche fare in modo che il passaggio dal vecchio coordinatore al nuovo venga effettuato in ogni nodo in modo atomico.

## Distributed Locking

Se ci troviamo in un sistema distribuito vogliamo poter effettuare delle operazioni sui dati garantendo la consistenza, specialmente se i dati sono replicati su vari nodi. Quindi quando il dato è replicato in vari nodi facciamo una differenziazione tra l'elemento logico X che vogliamo modificare e la copia fisica di X che abbiamo all'interno dei singoli nodi. Abbiamo vari sistemi per gestire le lock all'interno di un sistema distribuito:

- Il primo metodo è un metodo centralizzato in cui abbiamo un nodo che funziona da lock manager e mantiene una tabella delle lock per i vari elementi logici che abbiamo nel sistema, sia se poi questa copia sta anche nel lock manager sia se non sta nel nodo del lock manager.

Quando una transazione vuole la lock su un elemento logico X viene inviata una richiesta di lock al lock manager che può garantire la lock logica sull'elemento X oppure può rifiutare la richiesta. Quindi qua il costo è di tre messaggi per ogni lock.

L'utilizzo di un sistema di questo genere e di una sola lock logica per ogni elemento logico che abbiamo nel sistema può essere un problema perché abbiamo un single point of failure e se quel nodo cade risulta impossibile prendere le lock.

Soltamente quando si considerano le lock in questo sistema distribuito si va a calcolare un cost model che dipende da quanti messaggi vengono scambiati durante il tentativo di acquisire la lock. Se abbiamo un lock manager per ogni lock abbiamo al più 3 messaggi (richiesta, risposta, rilascio). Se invece utilizziamo un sistema diverso in cui assumiamo di non avere repliche, le lock locali al mio nodo le prendo senza inviare messaggi mentre invece le lock che sono all'interno di altri nodi mi costano ognuna 3 messaggi, quindi ho un numero di messaggi pari al numero di lock che devono essere acquisite \* 3.

Quando abbiamo un sistema in cui lo stesso dato è replicato varie volte su tanti nodi allora dobbiamo stare attenti a come interpretiamo la lock di X. Supponiamo di avere due copie di X, abbiamo X1 su un nodo e X2 su un altro nodo. Con una transazione accediamo a X1 e la leggo e con una transazione accediamo a X2 e la modifichiamo. Quindi ci troviamo in una situazione in cui due elementi che dovrebbero essere uguali in realtà hanno un valore differente, quindi non è garantita la consistenza e inoltre non è nemmeno garantita (in alcune situazioni) la serializability.

Quindi in questi casi l'unica cosa da fare è una distinzione tra una lock da prendere sul dato logico che appunto è logico e non esiste fisicamente e una lock da prendere sul dato fisico nel nodo vero e proprio. Un elemento logico X può avere 0 lock esclusiva e 0 lock shared oppure 0 lock esclusive e tante shared.

Un primo metodo che migliora l'approccio centralizzato è il Primary-Copy Locking:

- In questo caso abbandoniamo l'idea di avere un lock manager centralizzato e distribuiamo il lock manager. Mi rimane invece l'idea che un certo dato abbia una lock logica, una volta presa quella lock logica sono sicuro di poter lavorare su tutte le repliche di quel dato.

Quindi per ogni dato X consideriamo la sua copia logica, ognuno di questi dati ha una copia primaria che si trova su uno dei nodi (non ho un single point of failure), quando ho bisogno della lock prendo questa lock primaria e ho finito.

Un altro approccio ci permette di utilizzare le lock locali senza però andare a identificare una delle lock come primaria, tutte sono simmetriche e quindi posso richiedere la lock ad una delle tante copie.

La chiave del funzionamento di questa procedura è che lo schema di locking deve essere globale e quindi avremo un certo numero di transazioni che saranno necessarie per prendere una lock globale su X. Supponiamo che un elemento A ha n copie, consideriamo due numeri in questo caso:

- s è il numero delle lock shared che devo prendere sui dati ripetuti per fare in modo di avere la lock globale in shared mode

- $x$  è il numero delle lock esclusive che devo prendere sui dati ripetuti per fare in modo di avere la lock esclusiva

È fondamentale che  $x+s>n$  e che  $2x>n$  nel senso che il numero delle lock esclusive deve essere maggiore rispetto al numero delle lock shared perchè altrimenti ci troviamo ad avere nello stesso momento la possibilità di prendere la lock esclusiva e la lock shared e questo non va bene. Il numero di messaggi che vengono inviati può essere alto quando consideriamo la lock esclusiva perchè avremmo da inviare tanti messaggi per ottenere la lock, invece sarà basso quando dobbiamo prendere la lock shared.

- Se consideriamo la soluzione Read-Lock-One e Write-Lock-All, in questo caso ci troviamo con la lock della lettura che deve essere presa 1 volta, quindi scambiando al più tre messaggi mentre la lock della scrittura va presa su tutti i dati ripetuti quindi è molto più costosa.

Il problema di questo approccio è che se un nodo che ha la lock va offline io prendo la lock di tutti gli altri nodi ma non di quel nodo e quindi il suo dato rimarrà non aggiornato.

- Un'alternativa è avere una “Majority Locking”, in questo caso abbiamo che in entrambi i casi viene preso un numero di lock che è pari alla metà dei dati ripetuti, quindi in entrambi i casi il numero di messaggi che vengono inviati è comunque alto.

In questo modo sono sicuro che lettura e scrittura non vengono eseguite in contemporanea perchè devo prendere il 51% delle lock per scrivere ma anche per leggere. Quindi è impossibile che due transazioni abbiano legalmente una scrittura nello stesso momento o una lettura mentre un'altra scrive. Per fare l'aggiornamento di un dato questo metodo è ottimo ma non è ottimo quando devo fare la lettura perchè comunque devo prendere il 51% delle lock e questo comporta che devo leggere 51 valori (se ho 100 dati) quando me ne basterebbe solamente 1.

- Esiste un protocollo intermedio, potrei dire che nel caso di una scrittura mi serve il 99% delle lock mentre per la lettura mi basta

leggere solamente 2 copie del dato. Qua utilizziamo un quorum approach in cui il quorum esclusivo deve essere più grande e maggiore del 51% mentre lo shared può anche essere 2 (viene calcolato come  $N-X+1$ )

Ora la domanda è quanto è grande N, N è il numero delle repliche dei dati, non è il numero dei nodi che abbiamo nella rete, se copio i dati su tutti i nodi allora abbiamo che N è uguale al numero di nodi.

Più aumenta la ridondanza dei dati e meglio diventa il procedimento di lettura, allo stesso tempo però peggiora la scrittura perchè ho tanti dati da aggiornare e costeranno di più.

Tipicamente ho due situazioni:

- Le letture sono molto comuni e le scritture sono poco comuni, in questo caso copiamo i dati su tutti i nodi e usiamo l'approccio con due lock per la lettura e  $S = 2$  e  $X = N-1$  per la scrittura
- Se invece abbiamo una situazione con il numero di scritture e letture bilanciate allora tengo 2 o 3 copie dei dati e preferisco avere un numero di lock che è lo stesso per le letture e per la scrittura, quindi  $S = X = (n+1)/2$

Consideriamo anche che potrebbe non piacermi molto l'idea di leggere due copie dei dati ogni volta perchè, sappiamo che gli update sono comuni ma magari non sono troppo comuni e quindi mi trovo a leggere due volte lo stesso dato e a pagare un prezzo alto per cercare di evitare un evento che accade non troppo di frequente. Se sto solamente eseguendo delle operazioni statistiche potrebbe non essere troppo interessante se leggo un dato più vecchio e quindi in questo caso potrebbe anche andarmi bene una singola copia.

Se vogliamo transazioni e serializability questo è quello che dobbiamo fare, in alternativa se non serve la serializability possiamo anche leggere senza usare le lock. Questo è importante perchè in sistemi distribuiti succede spesso che leggo senza lock e scrivo senza interessarmi della consistenza.

## Gestione delle deadlock

Per gestire le deadlock in un approccio distribuito funzionano tutti gli algoritmi che funzionavano anche in precedenza con gli altri metodi. Quindi possiamo avere un timeout o un wait for graph o una deadlock prevention.

In tutti i casi esiste una versione distribuita, in generale si utilizza il timeout.

## Lezione 15: BigData & NoSQL

Storia dei DBMS:

- Inizialmente database creati appositamente in alcune situazioni particolari
- Poi sono nati i database relazionali e poi è arrivato il concetto di client-server con un database su entrambi i lati
- Poi sono arrivati gli Object Oriented Database che sono stati proposti per cercare di risolvere i mismatch a livello di implementazione. Ad esempio se viene scritto un codice che lavora con oggetti complicati e vogliamo memorizzare questo oggetto nel database dovremmo dividere l'oggetto in tante parti più piccole, questo problema si chiama impedance mismatch e vuol dire che lavoriamo in un modo in Java e in un modo differente con SQL.

Alla fine questo approccio è scomparso del tutto.

- Poi è nato il concetto di Big Data che è un insieme di tre concetti differenti tra loro ovvero:
  - Volume: abbiamo una grande quantità di dati da analizzare e quindi da leggere, visto che nel modello tradizionale dei DBMS ogni tabella esegue un table scan all'inizio un

- approccio di questo genere non va bene perchè ci metterei troppo tempo
- Velocity: nel modello tradizionale perdiamo del tempo anche a fare uno schema di questo database all'inizio. Se vogliamo fare una analisi veloce o capire magari la correlazione tra due attributi non mi conviene fare così perchè perdo troppo tempo, quindi servirebbe un approccio differente
  - Variety: ci sono situazioni in cui i dati del database arrivano da varie fonti, ognuna di queste fonti memorizza i dati in modo differente e quindi potrebbe essere necessaria una forma differente di analisi, se noi fissiamo uno schema fin dall'inizio questo potrebbe essere un problema per memorizzare altri dati che arrivano da altre fonti.

Per gestire il problema dei Big Data Google, che doveva analizzare petabyte di dati, ha sviluppato dei data center appositi in cui non si installa Oracle o MySQL, hanno inventato un nuovo modo per lavorare con i dati.

Google ha sviluppato il Google File System che è un metodo differente e ridondante per lavorare con queste grandi quantità di dati. Se abbiamo 100 macchine collegate e vogliamo eseguire un codice Java su queste 100 macchine ci serve un pattern completamente differente da utilizzare. Quindi hanno inventato il pattern chiamato "Map Reduce" che permette di eseguire la stessa operazione in parallelo su 100 macchine.

Successivamente hanno implementato BigTable che non è un database completo ma un database molto semplice che perde tante feature dei classici database, ad esempio non possiamo fare la join tra le varie tavole.

Questa stessa architettura è stata poi clonata in un progetto Open Source chiamato Hadoop.

Per gestire meglio i Big Data sono stati sviluppati i sistemi NoSQL che garantiscono la Velocity e sono adatti anche quando abbiamo dati che sono molto differenti tra loro. Sono adeguati per contenere grandi quantità di dati, dove i normali DBMS hanno delle difficoltà.

Non sono la soluzione per tutti i problemi, altrimenti i sistemi SQL non esisterebbero più.

Ci sono alcune caratteristiche dei sistemi NoSQL:

- Non implementano le transazioni ACID, solitamente implementano una forma più blanda di ACID e quindi questo aiuta le applicazioni in cui dobbiamo eseguire solamente delle query. Il problema delle transazioni che rispettano ACID è ancora più difficile da risolvere quando siamo in un contesto distribuito quindi in questo caso usare un sistema NoSQL può essere una buona idea.
- La seconda idea interessante è l'idea che nei classici DBMS partiamo da uno schema per poi inserire i dati. Nei sistemi NoSQL abbiamo un grande risparmio di tempo nella prima fase del lavoro perchè non abbiamo la necessità di definire uno schema preciso come quello dei classici DBMS, possiamo avere dati che non sono omogenei tra loro e metterli comunque all'interno del database senza modificare il sistema.
- Inoltre nei sistemi NoSQL non viene utilizzata la forma normale e quindi rinunciamo anche a fare un gran numero di Join.

I sistemi NoSQL sono differenti dai sistemi NewSQL, questi ultimi infatti sono sistemi che usano SQL con un nuovo approccio, usano infatti dei column database, in NoSQL noi lasciamo completamente SQL.

Esistono varie tipologie di database NoSQL:

- Abbiamo un database che mi permette di memorizzare coppie <key, value>, abbiamo due colonne nel database, una con un ID e una per un oggetto che mi permette di oscurare i dati che sono contenuti all'interno dell'oggetto. Questo tipo di storage è anche semplice da parallelizzare su 100 macchine perchè ci limitiamo a tagliare i dati in 100 blocchi e poi a distribuirli.
- Poi abbiamo i Document Databases, MongoDB è quello maggiormente conosciuto da questo punto di vista, sono database dove memorizziamo oggetti XML o JSON.  
Ci sono anche forme di restrizione che possiamo utilizzare su questi database perchè il database qua è in grado di andare all'interno dei dati che sono contenuti al suo interno e per ogni json

object ad esempio può lavorare come una classica query filtrando i dati.

Tipicamente in questi database abbiamo anche l'idea di key, alcuni database di questo genere lavorano con un approccio key,value in altri casi invece abbiamo un campo specifico per ogni chiave.

Non abbiamo bisogno di definire uno schema, inseriamo Json o XML senza indicare il formato.

- Nell'approccio Sparse Table database lo schema è una grande tabella, possiamo avere un grande numero di colonne, ad esempio una colonna per memorizzare i dati di uno studente poi altre colonne per i dati di un esame. È anche possibile aggiungere colonne dinamicamente, l'accesso alle colonne viene effettuato con una chiave di accesso ai dati.
- Le prime tre soluzioni sono per gestire i Big Data, c'è una quarta soluzione che è chiamata Graph database, qua i dati sono delle triple <subject, predicate, object> e possiamo rappresentare ogni database come un grafo. Questa soluzione è stata sviluppata per lavorare con i dati dei social network dove tutto è un grafo.

### Perchè utilizzare NoSQL?

Quando definiamo un classico database è necessario definire anche lo schema del database, se non si conoscono i dati già prima di cominciare (ad esempio se stiamo prendendo i dati da pagine web) può essere complicato creare lo schema, allo stesso tempo si perde del tempo per creare lo schema.

Se poi si considera anche l'ambito enterprise abbiamo che si è sempre pensato che ogni azienda dovrebbe avere un singolo database che contiene tutti i dati che la riguardano, in questo modo abbiamo molti vantaggi:

- Abbiamo alcune persone che si occupano del data management e tutta la gestione dei dati viene effettuata su un singolo database, quindi c'è solo un database da controllare e se devo fare una query la faccio velocemente perchè ho un solo database.  
L'idea di un database unico e centralizzato è stata spesso invocata ma poi in realtà non si è realizzata e si è passati all'idea del

federated database, abbiamo tanti database e usiamo un software che mi permette di eseguire le query decomponendola e andando quindi a fare le query sui singoli database. Poi il risultato viene unito in un unico risultato e viene presentato.

Questa idea non era praticabile fino a 5 anni fa poi sono state inventate le service distributed application e quindi ora è fattibile perchè abbiamo una interfaccia che è il protocollo REST, ogni servizio mi espone una web interface che è universale e quindi poi è possibile avere dei sistemi che dialogano.

Un altro motivo per cui si utilizza NoSQL è il fatto che possiamo utilizzare una distributed architecture con cluster.

Inoltre usare MongoDB, ad esempio, è più semplice di usare un database SQL perchè è più facile eseguire il setup e anche utilizzarlo. È più semplice perchè comunque mancano alcune feature che sono tipiche dei sistemi SQL, se servono cose molto complesse conviene utilizzare un sistema differente dal NoSQL, ad esempio se vogliamo supportare tante query differenti.

Cerchiamo di dare una definizione di NoSQL, in realtà non è proprio una definizione perchè NoSQL è più un termine generico che può essere utilizzato se siamo in casi d'uso di vario tipo:

- In teoria è un sistema che non supporta SQL, in realtà abbiamo alcuni sistemi NoSQL che utilizzano anche SQL
- Solitamente sono progetti open source
- Sono progetti orientati ai cluster, non tutti ma tipicamente funzionano in questo modo e non supportano transazioni che rispettano ACID
- Sono progetti recenti
- Non abbiamo uno schema o tipicamente non abbiamo uno schema prima di iniziare a vedere effettivamente come sono organizzati i dati
- Sono una buona soluzione se sappiamo su cosa focalizzare l'attenzione, sono sistemi che vengono creati pensando all'idea del data warehouse

- Non esiste una tecnologia NoSQL ma esiste l'idea di NoSQL, è più un movimento ideologico
- Sono in realtà sistemi che si basano su un'idea molto vecchia, opposti rispetto all'idea di forma normale

L'idea dei database NoSQL si basa su due modelli differenti:

- Aggregate Data Model: quando vogliamo memorizzare dei dati all'interno del database, invece di dividerli in una serie di tabelle dove ogni tabella memorizza dei dati, possiamo creare un solo database e memorizzare tutto qua.

Se ad esempio abbiamo un codice scritto in java e un approccio tradizionale e vogliamo fare una selezione dei dati della tabella avremo da fare tante select in tante tabelle, stesso discorso per gli aggiornamenti, questo è anche parte del problema dell'impedent match con una rappresentazione dell'oggetto che è differente tra quella che abbiamo nel codice e quella che abbiamo nel database. L'aggregate data model ha alcuni vantaggi e alcuni problemi.

Vantaggi:

- Quando prendiamo i dati possiamo memorizzarli nella stessa forma in cui si trovano, quando vogliamo fare una query, se tutte le query sono su una certa tabella allora posso subito fare la query senza dover fare una join.

Problemi:

- Ridondanza, se abbiamo un gran numero di ordini per ogni cliente, memorizziamo per ogni ordine tutti i dati del cliente e quindi se poi dovessi fare una modifica per quel cliente dovrei andare a modificare i dati del cliente ogni volta che lo troviamo nel database. Se un eventuale aggiornamento riguarda solamente i dati futuri non ci sono problemi, ad esempio se devo modificare un prezzo non voglio che anche gli ordini vecchi vengano modificati, solamente i nuovi. In questo caso la ridondanza non è un problema.
- Dobbiamo decidere come aggregare i dati già dall'inizio, se voglio fare una query con una differente aggregazione dei dati devo fare un gran lavoro per fare questa modifica.

L'aggregazione quindi è ottima se abbiamo delle query che sono sempre le stesse e che vanno bene per delle query, non posso fare troppe modifiche perchè altrimenti ogni volta devo fare le join.

Se utilizziamo un aggregate data model abbiamo un comportamento differente a seconda del tipo di database che abbiamo, ad esempio con un key,value store abbiamo un database in cui memorizziamo per ogni chiave un oggetto, in un document database invece memorizziamo tutto come XML o JSON e in un column database utilizziamo una colonna come chiave e altre colonne per memorizzare le informazioni, a patto che però sia tutto su una stessa riga.

Il key,value store è implementato distribuendo i dati in un numero grande di macchine parallele, abbiamo dei nodi che agiscono da coordinatori e dicono agli altri cosa devono fare, quando un nodo è down allora il coordinatore alloca il task in un altro nodo. Le query grandi poi vengono suddivise in task più piccoli e distribuiti.

- Graph Data Model

Tutti questi sistemi NoSQL sono senza schema, alcuni completamente senza schema, altri hanno un minimo di schema che viene creato nel momento in cui conosciamo i dati con cui lavoreremo.

In alcuni casi c'è un'inferenza dello schema, inseriamo i dati e poi lo schema viene capito.

Materialised View: l'aggregated data model dice che devi scegliere un modo per aggregare i dati. Se stiamo scegliendo il tipo di aggregazione per un sistema OLAP e scegliamo due tipi di aggregazioni non è un problema perchè comunque guadagnamo sul lavoro successivo andando ad eseguire solamente una volta ora la join.

Questo funziona solamente in un approccio OLAP

In un key, value database indichiamo anche che tipo di distribuzione vogliamo per le coppie di dati, dobbiamo scegliere come frammentare ovvero lo shading e anche come allocare i dati ovvero la replication.

Per ogni tipo di distribuzione che scegliamo dobbiamo anche decidere in

che modo allocare i dati, quindi dobbiamo trovare un replication model che sia adeguato al mio caso d'uso:

- Alcuni sistemi utilizzano un sistema Master-Slave, per ogni dato c'è una master copy e un set di slave copy. Quando aggiorniamo i dati, l'aggiornamento avviene sulla master copy, poi il sistema propaga i dati negli slaves. Quando vogliamo leggere i dati possiamo leggere dalla slave copy se non siamo interessati alla consistenza, altrimenti dalla master copy.  
Quando un master va offline allora uno degli slave viene promosso a master. È importante che ogni volta che viene scelto un nuovo master tutta la rete sia d'accordo e soprattutto che tutti conoscano il nuovo master e questo è difficile, per un master election avremmo bisogno di un 2 phase commit protocol.
- Un altro tipo di replication è il P2P, niente di nuovo rispetto al modo in cui si devono coordinare le transazioni. Quello che è nuovo è il punto di vista, in un database distribuito partiamo dall'assunzione che quello che voglio fare è implementare una transazione e inizio a chiedermi quali sono i protocolli che posso usare per implementare la transazione. Qui invece abbiamo un punto di vista completamente opposto, mi chiedo quale protocollo vorrei avere, ad esempio voglio avere un master slave replication, ora dopo che ho creato il protocollo mi chiedo se è consistente, in un DBMS invece partiamo col dire che io voglio il database consistente. Qua mi chiedo come fare a ottenere la consistenza se di base non è garantita e non ce l'ho.

In un database NoSQL partiamo dicendo che vogliamo la velocity e quindi iniziamo a diventare troppo lenti se vogliamo implementare la consistenza.

Fin dall'inizio in questi sistemi inizi a pensare a vari livelli di consistenza, siamo disposti a perdere alcuni aggiornamenti o vogliamo essere sicuri di non perdere mai un update al database?

Siamo disposti a leggere dati che sono consistenti ma che non sono aggiornati oppure vogliamo sempre l'ultimo dato che è stato aggiornato?

Questo problema in questo sistema è più importante perché ci sono tanti applicazioni (Big Data applications) per questo sistema in cui non ci interessa della consistenza ma ci interessa solamente della velocity.

Vantaggi essenziali di NoSQL:

- Schemaless
- Supporto ai cluster
- La consistenza è flessibile

Svantaggi di NoSQL:

- Non hanno supporto alle transazioni ACID, quindi ad esempio una banca non può utilizzare NoSQL
- È difficile decidere il livello corretto di consistenza
- Non hai SQL e quindi devi decidere in che modo aggregare i dati, se tutte le query riguardano lo stesso argomento va bene altrimenti no
- Ci sono delle implementazioni di SQL su Hadoop o su Spark ma non hanno lo stesso livello di ottimizzazione delle soluzioni native.

Trend architetturali:

- Data Lake: invece di preparare un sistema data warehouse con tutta la perdita di tempo iniziale del data cleaning si preferisce prendere i dati e utilizzarli direttamente per l'analisi inserendoli in questo "lake". Si prova ad eseguire algoritmi di machine learning e reti neurali sperando di trovare dei pattern interessanti. Questa non è un'alternativa al data warehouse è un'altra possibilità.
- Polyglot system: combinano DBMS, DSS, NoSQL nello stesso centro IT, abbiamo vantaggi da alcuni punti di vista ma ci sono anche svantaggi perché sono più complicati e sono difficili da mantenere. Se vogliamo eseguire una query distribuita poi abbiamo bisogno di usare un framework.  
Ci sono anche problemi di sicurezza perché abbiamo un numero

maggiori di punti di attacco per il sistema però comunque molte aziende usano questa soluzione.

**Cap Theorem:** non è possibile avere consistenza, availability e partition tolerance tutte e tre insieme. Se vogliamo avere l'availability allora dobbiamo rinunciare alla consistenza e se vogliamo la consistenza invece rinunciamo all'availability e dobbiamo essere preparati ad avere anche una latenza nelle risposte e negli aggiornamenti.

L'availability mi dice che io posso modificare e accedere ai dati in qualsiasi momento ma questo mi può causare problemi con la consistenza.

In alcuni casi possiamo rilassare la consistenza e anche la durability. Molti sistemi big data permettono di avere una consistenza rilassata, cosa che non puoi avere in un classico database che invece necessita di quel tipo di consistenza.

Ad esempio Amazon ha una consistenza rilassata quando facciamo un acquisto e solamente quando si arriva all'ultima pagina, quando clicchi sul pulsante “ok compra” allora viene fatta una operazione 2 phase commit che è robusta e che mi garantisce la consistenza.

Quando si usano questi sistemi dobbiamo essere in grado di capire cosa è importante per noi, se vogliamo che sia garantita la consistenza o no.

Per quanto riguarda la consistenza, nei sistemi che supportano l'aggregate data model solitamente non viene supportata l'atomicity ed è supportata la consistenza 1 documento alla volta, si tratta di una single entity atomicity.

Non è semplice implementare la consistenza se abbiamo più copie dello stesso oggetto, tipicamente si cerca di implementare un approccio con il quorum in cui si devono prendere abbastanza conferme per poter scrivere o leggere.

Un altro approccio è quello ottimistico che è simile a quello ottimistico dello snapshot isolation, quando dobbiamo leggere dei dati abbiamo un timestamp sul dato e noi abbiamo successo solamente se nessun altro aggiorna il dato. L'approccio ottimistico è semplice e non è molto

costoso, l'unico problema è che non è semplice mantenere un timestamp coerente e distribuito.

Una versione tipica del timestamp è quella in cui si utilizza un counter, quando aggiorniamo il dato abbiamo la versione 1, la 2, la 3.... questo non è un buon metodo quando abbiamo due processi che in contemporanea modificano la stessa versione e ottengono ad esempio entrambi la versione 3, diventa difficile capire chi è stato l'ultimo ad aggiornare.

Un'altra tecnica utilizza il globally unique identifier, generiamo un numero random di 128 bit che viene aggiornato quando vogliamo aggiornare i dati. Il problema in questo caso ce l'abbiamo quando dobbiamo fare un aggiornamento perchè vogliamo fare in modo di mantenere quello che si aggiora per ultimo ma in questo senso il GUID non ci aiuta.

Un'altra tecnica che ci aiuta è la vector clock, ogni versione dei dati ha un clock che non è solamente un counter ma è un vector counter che mi dice che, per il nodo A quella è la versione 7 del file, per il nodo B è la versione 8. Quando siamo nel nodo B e aggiorniamo l'elemento allora aumentiamo solamente il contatore nel mio nodo e non l'altro, se qualcuno in parallelo incrementa quello stesso valore, modifica il valore nel suo nodo e non negli altri. Quando devo considerare le copie potrei avere ad esempio A,5,9 e B,6,9 e in questo modo ho una time relation tra le due versioni, queste sono in parallelo e nessuna viene prima o dopo l'altra, se invece abbiamo A,5,9 e B,5,10 allora sappiamo che la seconda versione viene dopo la prima e in questo modo abbiamo la relazione temporale.

**Map-Reduce:** l'idea della Map-reduce è che il programmatore dovrebbe essere in grado di eseguire il codice indipendentemente dal numero di nodi che ha nella rete. Quando una macchina va offline non devo nemmeno accorgermene e devo continuare l'esecuzione del codice. La map-reduce cerca proprio di fare questo e prova a sfruttare il fatto che abbiamo tante macchine a disposizione.

Il protocollo funziona in questo modo:

- Abbiamo una prima fase dove la stessa funzione viene applicata a tutti gli oggetti della nostra collezione distribuiti nei vari nodi. Questa prima fase è chiamata map phase e abbiamo una funzione map che mi indica cosa deve essere fatto su ogni elemento della collezione. Ad esempio se abbiamo un libro e sono state distribuite le pagine in tanti nodi la nostra funzione map potrebbe essere il conteggio delle parole nella pagina che chi è stata assegnata. Quindi la funzione map che svolgo prende ogni parola che trova nel testo e poi emette per ognuna una coppia `<key, value>` dove la chiave è la parola e value è il numero di occorrenze.
- Una seconda fase poi distribuisce i dati: consideriamo tutte le coppie generate nella prima fase e le distribuiamo nei vari nodi in base alla chiave, questo viene fatto dal sistema che quindi ridistribuisce nei nodi le varie coppie.  
Il programmatore deve specificare una funzione reduce che, una volta ricevuti i risultati della map phase calcola per ogni key la somma di tutti i valori ricevuti.

La parte interessante è che è semplice applicare questo metodo in modo che sia resistente ai fallimenti, abbiamo un set di nodi e il programmatore chiede di applicare la funzione ad un milione di documenti dividendo il carico di lavoro nei vari nodi.

Se il nodo a cui è stato assegnato il lavoro va offline e quindi non risponde in un tempo decente allora possiamo assegnare quel lavoro ad un altro nodo, non abbiamo bisogno di fermare il sistema e ricominciare dall'inizio.

L'implementazione di questo protocollo è differente in Hadoop e in Spark, in Hadoop abbiamo che l'input e l'output di ogni fase sono memorizzati nel disco usando un file system distribuito e ridondante. In Spark, definito successivamente, abbiamo la possibilità di mantenere il risultato di ogni fase in memoria principale e questo mi permette di eseguire le operazioni più velocemente.

Una fase importante di questo processo è la scelta della dimensione del task, se creiamo un task troppo grande abbiamo tanto lavoro da fare

quindi magari ci troviamo con un nodo più lento che ci mette troppo tempo rispetto agli altri. L'idea basilare è che il coordinatore crea dei task abbastanza piccoli e poi li distribuisce ai vari nodi.

La divisione del lavoro in task che siano piccoli è estremamente importante per un sistema che deve essere realmente veloce.

# Physical Design

## Strutture dati da usare:

- **Heap organization:** usata quando abbiamo pochi dati oppure quando dobbiamo leggere la maggior parte dei dati che abbiamo memorizzato;
- **Sequential organization:** usata quando abbiamo la necessità di ordinare i nostri dati in base ad un certo attributo;
- **Hash organization:** usata quando l'operazione più frequente che svolgiamo è la ricerca all'interno del set di dati tramite l'utilizzo di una chiave;
- **Index sequential organization:** usata quando vogliamo mantenere i dati ordinati in base ad una chiave permettendo allo stesso tempo di eseguire inserimenti all'interno del database. Inoltre l'operazione che svolgiamo maggiormente è la ricerca di record tramite una chiave oppure tramite un range di chiavi.

## Utilizzo degli indici

Possiamo avere dei benefici nell'utilizzo di indici:

- Indici su chiavi primarie e chiavi esterne
- Quando va fatto un range search sull'attributo su cui viene organizzato l'indice
- Quando abbiamo una condizione in AND
- Quando dobbiamo poi svolgere una operazione in cui i dati servono ordinati (order by, group By, distinct) e abbiamo che la chiave che usiamo nell'indice è la stessa che viene usata per creare l'indice
- Quando grazie all'indice possiamo svolgere delle operazioni Index Only

In alcuni casi due indici possono essere uniti in un solo indice, non usiamo gli indici quando abbiamo la condizione in OR.

## Soluzioni

- Ammettiamo di avere una relazione R(A,B) dove la coppia A,B è una chiave primaria per R. Se facciamo una selezione del tipo R.B=0 all'interno di R, vuol dire che automaticamente A diventa chiave primaria per la tabella perchè abbiamo che necessariamente i valori di A per cui B=0 saranno differenti tra loro.  
Questo comporta che nel caso di un merge join non avremo problemi perchè essendo A chiave primaria non ho il problema di dover tornare indietro nella tabella.
- Se abbiamo la relazione R(A,B) con la coppia A,B chiave primaria e un numero complessivo di record pari a Nrec, cosa possiamo dire del numero di chiavi di A e di B? Possiamo solamente dire che il numero di chiavi sarà compreso tra 1 e Nrec e poi possiamo dire che  $Nkey(Totale) < Nkey(A)*Nkey(B)$ .
- Il distinct lo possiamo eliminare:
  - Se non ho una group by devo partire con la closure in cui metto solamente gli elementi della select poi aggiungo tutti gli altri che posso aggiungere. Se tutti gli attributi delle due

- tabelle su cui faccio la join sono nella closure allora il distinct lo posso eliminare
- Se ho una group by creo la closure come sopra, la differenza sta nel fatto che ora nella closure devono essere presenti gli attributi del group By.
  - L'index nested loop preserva l'ordinamento della parte sinistra ma non quello della parte destra. Se ad esempio a sinistra facciamo una index filter su un indice R.A clusterizzato allora il risultato dell'index nested loop sarà ordinato in base a R.A
  - Se abbiamo la condizione T.idt = R.idt allora devo considerare il numero delle chiavi differenti di T perchè idt è chiave primaria di T, quindi il numero massimo di chiavi differenti tra T e R sarà comunque il numero di chiavi di T.
  - Perchè viene creato un indice sulla primary key e sulla foreign key?  
Viene creato un indice sulla foreign key perchè questo può comportare dei miglioramenti per le performance nel momento in cui svolgiamo una join sugli attributi PK e FK di due tabelle.  
Inoltre può migliorare le performance anche quando un record viene cancellato o aggiornato.  
Creiamo in automatico un indice sulla chiave primaria in modo da garantire che i vincoli vengano rispettati.
  - Se faccio una filter e voglio capire quanti record rimangono dopo la filter e quante pagine posso calcolare  $E_{rec} = N_{rec}(O) * sf$  e  $N_{pag}(\text{dopo filter}) = N_{pag} * sf$ .
  - Dopo il merge join il risultato sarà ordinato in base agli attributi su cui è stato fatto il precedente ordinamento, ad esempio se ho la tabella studenti.ID e esami.IDStudente nella join allora il risultato verrà ordinato in base a studenti.ID. Se metto anche un altro attributo nel sort, quindi ad esempio esami.IDStudente lo ordino anche in base al voto che è stato preso, la soluzione sarà ordinata anche in base al voto.
  - Nell'Hash Join il numero di passi che vengono fatti dipende dalla dimensione della B e dalla relazione esterna. La formula per capire quanti passi si fanno è  $\log_b(N_{page}(\text{outer relation}))$ . Per

calcolarlo consideriamo le potenze della B, quindi B, B al quadrato e B al cubo.

- In generale quando dobbiamo fare un merge Sort tra due tabelle e poi il risultato va mergiato con un'altra tabella possiamo considerare come tabella più a destra, quindi come tabella da mergiare per ultima, quella che comporta una Join espansiva e che mi va ad aumentare la dimensione della tabella.
- Tra MergeJoin e HashJoin preferiamo il Merge Join perchè il costo dipende dal Sorting mentre nell'hash join dipende dal calcolo dell'hash.
- Se dobbiamo fare la join tra una tabella piccola (anche una tupla) e una tabella grande (esempio studente esame) non ha senso fare il sort merge join ma ha senso invece fare un index nested loop perchè in questo modo evitiamo di portare in memoria tanti dati inutili.

## Domande ricevimento

Differenza tra primary e secondary organization:

La primary organization mi indica in che modo vengono organizzati i vari record sul disco, quando ne arriva uno nuovo mi permette di scegliere in che modo inserirlo sul disco.

Ne abbiamo 4 di primary organization:

- Abbiamo la più semplice che è l'hash, quando arriva un nuovo hash lo mettiamo in fondo all'ultima pagina che abbiamo creato
- La sequenziale mi mantiene un ordine su uno degli attributi che abbiamo all'interno della tabella
- L'hash mi prende il record, calcola l'hash e quindi la pagina in cui finisce quello specifico record
- Con i Btree abbiamo una organizzazione ad albero.

La secondary organization invece la utilizzo per l'accesso successivo ai dati, in particolare per accedere ad esempio con una query. Questa secondary organization corrisponde agli indici, data una tabella possiamo avere vari indici sui vari attributi della tabella. Gli indici non sono necessari, possiamo anche non averli. L'organizzazione fisica degli indici però si fa utilizzando i Btree, quindi usando una primary organization.

## **Appunti dal libro Database System: The Complete Book (CAPITOLO 20 SU DATABASE PARALLELI E DISTRIBUITI)**

La join tra R e S distribuita la possiamo fare banalmente inviando nel nodo dove sta S tutta la tabella di R e poi possiamo fare la join.

Se però la comunicazione tra i due nodi è molto costosa abbiamo la possibilità di sfruttare la semi join.

- Prendiamo la tabella più piccola ed estraiamo solamente la colonna su cui facciamo la join. Eliminiamo anche i duplicati. Se è minore di S allora la inviamo al nodo in cui c'è R.
- Nel nodo dove sta R consideriamo la semi join tra R e quello che ho ricevuto da S. In questo modo quello che ho in R dovrebbe diminuire di dimensione. Se diminuisce mandiamo il risultato della semijoin nel nodo di S. Il risultato della semi join contiene quei record che poi dovranno finire anche nella join finale.

- Nel nodo S, una volta ricevuta la semijoin di R possiamo fare la join completa.

## Confronto tra i vari tipi di organization:

Npag = numero delle pagine

Nrec = numero dei record

Lr = lunghezza record

Dpag = dimensione pagina

Sf =  $(k_2 - k_1) / (k_{max} - k_{min})$

h = altezza albero

File Type	Memory	Equality Search	Range Search	Insert	Delete	Update
<b>Heap</b>	$N_{pag} = N_{rec} * (L_r / D_{pag})$	$C_s = (N_{pag}/2)$ se la chiave esiste $C_s = N_{pag}$ se non esiste	Npag perchè accedo tutto il file	2 perchè viene inserito in fondo al file	$C_s + 1$ perchè cerco e poi 1 per la modifica.	$C_s + 1$ perchè cerco e poi 1 per la modifica.
<b>Sequentia I</b>	$N_{pag} = N_{rec} * (L_r / D_{pag})$	$\lg(N_{pag})$	$C_s = \log(N_{pag}) + (s_f * N_{pag}) - 1$ Perchè prima facciamo la ricerca per k1 e troviamo k2.	$C_s + N_{pag} + 1$ Consideriamo la ricerca più lo spostamento delle pagine che ho dopo il punto in cui inserisco	$C_s + 1$	$C_s + 1$
<b>Hash Statico</b>		1 solo accesso perchè uso l'hash	Npag perchè devo leggere tutte le pagine	Costo 1 fino a quando non ho overflow. Con l'overflow devo fare la	Costo della ricerca + 1 per update	Costo della ricerca + 1 per

				riorganizzazione e questo costa molto (4*Npage)	della pagina	update della pagina
<b>Hash Dinamico</b>		1 solo accesso perchè uso l'hash.	Npag perchè devo leggere tutte le pagine	Costa 1. Nel caso del linear hash potrei avere un costo maggiore perchè si può creare una sequenza.	Costo della ricerca + 1 per update della pagina	Costo della ricerca + 1 per update della pagina
<b>B Tree</b>		al più $h = \text{altezza albero}$	$C = s^h * Nnodes$	$h \text{ letture} + 2h-1 \text{ scritture}$	Se va bene $h+1$ . Se va male $2h-1 \text{ letture} + h+1 \text{ scritture}$	