Scheduler: un sistema che accetta, rifiuta o riordina le operazioni richieste dalle transazioni.

Schedule seriale: le transazioni sono sequenziali e avvengono una alla volta (il commit della prima segna l’inizio della seconda.)

Schedule serializzabile: produce lo stesso risultato di un seriale sulle stesse transazioni.

**View-Serializzabilità**

Due schedule S1 ed S2 si dicono view equivalenti se:

* Lo stesso insieme di transazioni partecipa ad entrambi gli schedule ed entrambi includono le stesse operazioni su queste transazioni.



* Se in S1 per ogni operazione di lettura di una transizione il valore letto è stato scritto da un’operazione di scrittura di un’altra transazione, allora in S2 per il medesimo valore vale la stessa condizione



* Se l’operaione di Wk(Y) è l’ultima a scrivere su S1 allora Wk(Y) deve essere anche l’ultima a scrivere su S2

Uno schedule si dice view\_serializzabile se è view\_equivalente ad un qualche schedule seriale, l’insieme di tutti gli schedule view\_serializzabili è chiamato **VSR.**

**Problemi della view\_serializzabilità:**

* Verificare se due schedule sono view-equivalenti ha costo polinomiale.
* Decidere se uno schedule è view\_serializzabile senza avere di fatto uno schedule seriale con cui confrontarlo è un problema NP-Completo dunque **non applicabile in pratica.**

**Conflict-Serializzabilità**

Un’azione si dice in conflitto con un’altra se operano sullo stesso oggetto ed almeno una di esse è in SCRITTURA

Due schedule sono conflict-equivalenti se includono le stesse operazioni ed ogni coppia di operazioni che sono in conflitto tra di loro appaiono nello stesso ordine in entrambi gli schedule

Uno schedule si dice conflict-serializzabile se è conflict-equivalente ad un qualche schedule seriale, l’insieme di tutti gli schedule conflict-serializzabile si dice **CSR**

La cardinalità di CSR è più vicina alla cardinalità dell’insieme degli schedule seriali rispetto a quella di VSR.

**Costo della verifica di Conflict-Serializzabilità**

Come verificare la conflict-serializzabilità?

Per mezzo del grafo dei conflitti:

un grafo con un nodo per ogni transazione dello scheduler ed un arco orientato tra due nodi se c’è almeno un conflitto tra le operazioni di quelle transazioni.

**TEOREMA**

Uno schedule è in CSR se e solo se il grafo è **ACICLICO**

Purtroppo, anche se l’algoritmo di verifica può essere molto rapido la conflict-serializzabilità è inutilizzabile in pratica poiché non si conosce la struttura del grafo dall’inizio, infatti lo scheduler opera incrementalmente

Cioà ad ogni richiesta di operazione esso decide se eseguirla subito oppure fare qualcos’altro. Non è praticabile mantenere il grafo aggiornarlo e applicare l’argoritmo di verifica ad ogni richiesta di operazione.

Dunque queste tecniche non sono praticabili.

Una tecnica che sembra fattibile è l’utilizzo dei lock sostituendo dunque lo scheduler con un gestore dei lock. (capitolo 19 libro).

**2PL BASE:** Quello descritto dalla teoria

**2PL STATICO**: una transazione deve bloccare tutti i lock prima dell’esecuzione della transazione, andando a dichiarare i read set ed i write set in anticipo

**2PL STRETTO:** la transazione non rilascia nessuno dei lock esclusivi fin quando non avviene il commit o l’abort

**2PL RIGOROSO:** come lo stretto ma la transazione non rilascia nessun lock, ne condiviso ne esclusivo.

**Controllo della concorrenza basato su Timestamp**

È un’alternativa al Locking a 2 fasi (2PL) e si basa, appunto sul timestamp ossia un identificatore che definisce un ordinamento totale sugli eventi di un sistema.

L’idea base è che ogni Transazione ha un timpestamp (un identificativo) che rappresenta l’istante preciso in cui la transazione inizia (come se timbrasse un cartellino); uno schedule è accettato solo se rispetta l’ordinamento seriale delle transazioni indotto dal timestamp

Ogni record del DB ha 2 contatori: un timestamp per l’ultima lettura del record (RTM) e uno per l’ultima scrittura (WTM).

Di seguito il funzionamento:

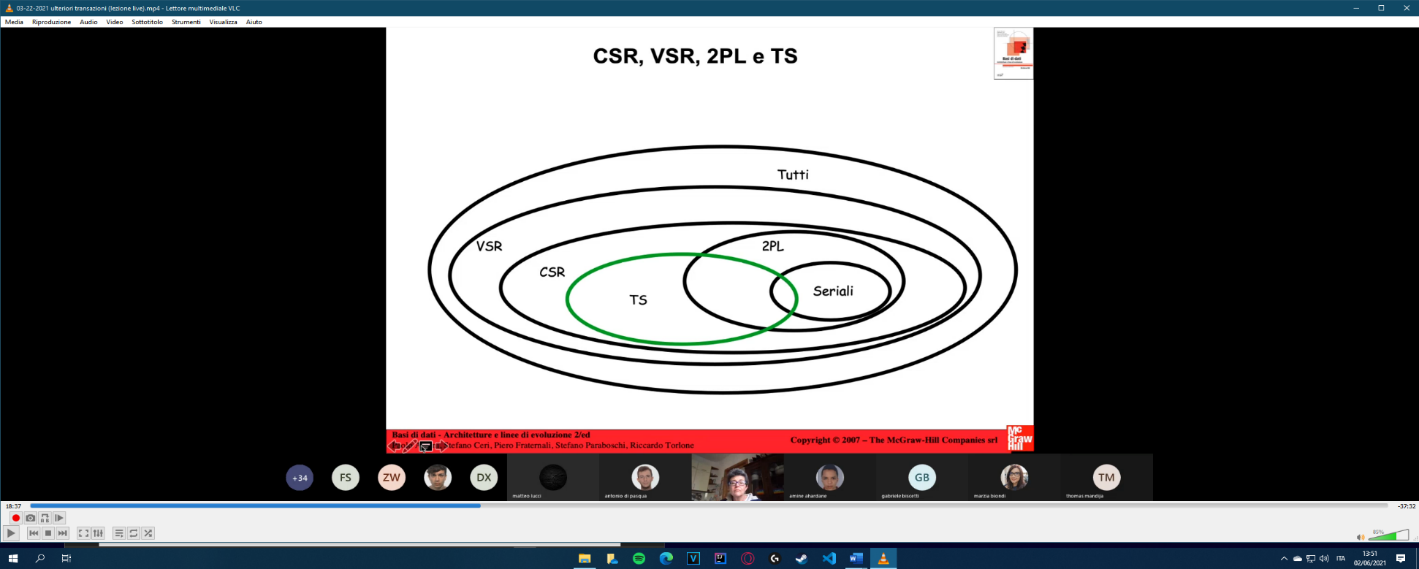
1)Transazione Read(x) con timestamp ts:

* se **ts < WTM** allora la richiesta è respinta e la transazione uccisa, questo perché il sistema sta cercando di leggere un oggetto scritto dopo che la transazione è già iniziata quindi è possibile che il record sia stato già letto con valore diverso e ciò implicherebbe un problema.
* Altrimenti la richiesta viene accolta e RTM viene posto uguale al maggiore tra RTM e ts.

2) Transazione Write(x) con timestamp ts:

* Se **ts< RTM(x) o ts<WTM(x**) allora la richiesta è respina e la transazione uccisa, poiché sto cercando di modificare un record che è stato già letto e già scritto ad una transazione “futura” quindi è come se stessi facendo un de-update del record.
* Altrimenti la richiesta viene accolta e WTM viene posto uguale a ts

Differenze tra Uso del Timestamp e 2PL

Controllando la concorrenza tramite timestamp vengono uccise molte transazioni: non vengono messe in attesa come nel 2pl.

Far ripartire da zero una transizione (che è stata uccisa nell’algoritmo timestamp) è molto più costoso di un resume a seguito di una messa in blocco in 2pl però il 2pl può causare situazioni di stallo (DEADLOCK)

Dunque, i pro e i contro sono tanti.

Un’evoluzione del timestamp è il:

**Multi Versioning Concurrency Control (MVCC)**

Funziona più o meno come il timestamp ma vengono uccise molte meno transazioni

1. Se viene ricevuta una transazione read(x) con timestamp ts:
   1. La lettura è sempre accettata e viene letto un valore x tale che : WTMi(X)<ts<WTMi2(X);
   2. RTM(x) è posto uguale al maggiore fra RTM(x) e ts.
2. Se viene ricevuta una transazione write(x) con timestamp ts:
   1. Se ts<RTM(x) allora la richiesta è respinta e la transazione uccisa
   2. Altrimenti viene scritta una nuova versione di x con WTM(x) uguale a ts.

Il concetto è che il MVCC offre diverse versioni di database a chi sta leggendo in base al momento in cui la richiesta di lettura viene effettuata. Anche se poi la versione (originale) del database è diversa poiché è in atto una modifica.

**Granularità dei lock (lock gerarchico)**

**Lock a livello di:**

DB -> Tabelle -> Tuple -> Tupla -> Campi -> Campo.

Viene scelto un lock a granularità fine per modifiche localizzate su limitati oggetti

Viene scelto un lock a granularità grande per accessi a grande mole di dati

Un livello troppo grossolano limita il parallelismo e la concorrenza

Un livello troppo fine implica troppo lavoro per il Lock Manager (vedi anche livelli di isolamento pag 544)

**UNDO E REDO**

Undo di un’azione su un’oggetto O:

* update,delete: copiare il valore dello stato before nell’oggetto O
* Eliminare O

Redo di un’azzione su O:

* insert, update: copiare il valore dell’after state nell’oggetto O
* reinserire O

Idempotenza di undo e redo:

* undo(undo(a)) = undo(a)
* redo(redo(a)) = redo(a)

**Regole Fondamentali per il log:**

Write Ahead Log:

si scrive il log prima del database

Commit-Precedenza

si scrive il giornale prima del commit

**Processo di Restart: (recovery)**

L’obiettivo del recovery è rendere di nuovo operativo un DB usando il log

Bisogna classificare le transazioni in:

* Completate (già in memoria)
* in fase di commit ma non completate (da fare un redo)
* senza commit (undo)

(il dump è un backup completo del database, salvato in memoria stabile. Nel log c’è anche un record che specifica quando il dump viene effettuato).

Tipi di ripresa: (per gestore dell’affidabilità)

**Ripresa a Caldo (GUASTI SOFT)**

La ripresa a caldo si compone delle seguenti fasi:

* trovare l’ultimo checkpoint (ripercorrendo il log a ritroso)
* costruire gli insiemi UNDO e REDO
* Ripercorrere il log in avanti aggiungendo ad UNDO le transazioni che iniziano e spostando da UNDO a REDO le operazioni che chiedono il commit
* Ripercorrere a ritroso il log compiendo gli UNDO quando necessario
* Ripercorrere di nuovo in avanti il log per fare i REDO

**Ripresa a freddo (GUASTI HARD)**

* Ripristino dei dati a partire dal BACKUP
* Si eseguono le operazioni registrate sul giornale fino all’istante del guasto
* si esegue una ripresa a caldo

**ANOMALIE**

Perdita di aggiornamento W-W

Lettura Sporca R-W con abort

Letture inconsistenti RW

Aggiornamento Fantasma RW

Inserimento Fantasma RW su dato nuovo

COMMIT A 4 FASI:

C’è una copia di backup del coordinatore e di tutto ciò che fa, in modo che se l’originale ha un guasto viene attivato il backup che, per prima cosa, crea un nuovo backup, e poi continua ad eseguire il protocollo di commit.

**Architetture e motori MySQL**

Il comando per vedere come sono costruite le tabelle è show engines

L’architettura MySQL un’architettura non perfetta bensì flessibile ed è un’architettura che separa i query processing dai task di memorizzazione e ricerca dati.

Si divide in più livelli

1. servizi di rete,connessione eccetera, simile ad altri dbms
2. è il cuore di mysql. implica il query parsing,l’analisi e qui si sceglie il parse tree. Qui vengono anche analizzati stored procedure e trigger
3. Il terzo livello sono i motori di storage che si occupano di memorizzazione e ripescaggio dei dati. Sono organizzati come se ognuno di essi fosse un file system. Il server comunica con essi attraverso API (contenenti funzioni a basso livello)

Quando un client si connette ad un server mysql il server verifica i privilegi del client

po mysql analizza le query per creare il parse tree

La scelta dello storage engine influisce anche sull’ottimizzazione delle query.

Come si comporta mysql rispetto al controllo della concorrenza

Viene fatto su due livelli: livello server e livello storage engine

in mysql esistono due lock, quelli shared(scrittura possibile solo da uno lettura da molti) e quelli exclusive (lettura e scrittura esclusivi su uno)

Esistono 2 tipi di granularità di lock, a livello di tabella o di tupla. La gestione dei lock avviene nello storage engine.

Per quanto riguarda le transazioni ogni motore implementa 4 livelli di isolamento:

Read uncommitted

Read committed

Repeatable read (default)

serializable

InnoDB riconosce le dipendenze circolari per riconoscere situazioni di deadlock delle transazioni (facendo poi rollback)

MySql ha 3 storage engine transazionali: InnoDB,NDBCluster e Falcon

MyIsam non è transazionale (ma è possibile fare lock e commit a mano)

Automaticamente MySql opera in autocommit mode (effettua un commit ad ogni “;” ) ma si può rimuovere questa opzione. “Select @@autocommit; set autocommit = 0;”

InnoDB, PBXT e Falcon utilizzano l’MVCC

**Storage engines**

Gli storage engines sono moduli software che si occupano della memorizzazione e del recupero delle informazioni.

MySql memorizza ciascun database come sottodirectory della sua directory di dati

Ogni storage engines memorizza dati ed indici della tabella in modo differente, ma è il server che gestisce la table definition.

**MyIsam**

Prima era il default, è un buon compromesso tra efficienza ed utilizzo ma non supporta transazioni e lock a livello di riga ( ma solo a livello di tabella)

tutte le tabelle sono memorizzati in due file: uno per i dati e uno per gli indici (sono portabili)

Tabelle fino a 256 TB di dati.

**MyIsam merge engine**

Non è un vero e proprio motore ma una combinazione di più tabelle myisam che viene vista come una singola tabella virtuale.

**InnoDB engine**

è il pià ppopolare per storage transazionale ma anche per efficienza e crash recovery automatico (warm e cold restart)

I dati vengono memorizzati in una serie di file unici detti tablespace

Usa MVCC e i 4 livelli di isolamento (anche il locking a 2 fasi)

Le tabelle sono costruite su clustered index: primary key molto veloci ma indici secondari meno veloci

non comprime gli indici

*Supporta il constraint sulle foreign key (a differenza di myIsam)*

**Memory Engine**

Tabelle heap: c’è un accesso molto veloce a delle tabelle che non cambiano.

Ad ogni restart del server quello che “sopravvive” è la struttura di una tabella ma non i suoi dati

Sono utili per tenere il risultato periodico di dati aggregati,risultati intermedi ecc.

Sono utilizzati anche per le query di MySQL

**Archive Engine**

può solo fare insert e select (non ha indici)

Supporta il row-level locking ma non è transazionale

Il log viene utilizzato tramite questo motore

**CSV (comma separeted value) Engine**

Tratta come tabelle i file con valori separati da virgole, non ha indici ed è utile per lo scambio di dati

**Federate Engine**

Le tabelle federated fanno riferimento a tabelle su un server MYSQL remoto (non ci sono dati locali) è adatto per singole richeiste sporadiche

**Black Hole Engine**

Viene utilizzato per simulare delle operazioni, poiché non ha nessun meccanismo di memorizzazione.

**NDB** **Cluster Engine**

Alta velocità (era di proprietà di sony)

Mantiene tutti i suoi dati in memoria

Ha un’architettura complessa basata sul “ non condivido nulla “

consiste in nodi di dati, di gestione e nodi sql

ciascun nodo di dati mantiene un frammento di dati, possono esistere frammenti duplicati su più nodi.

**Falcon Engine**

Progettato per processori multipli a 64 bit, usa MVCC e tenta di mantenere le transazioni tutte in memoria

non è ancora finito.

**Solid DB Engine**

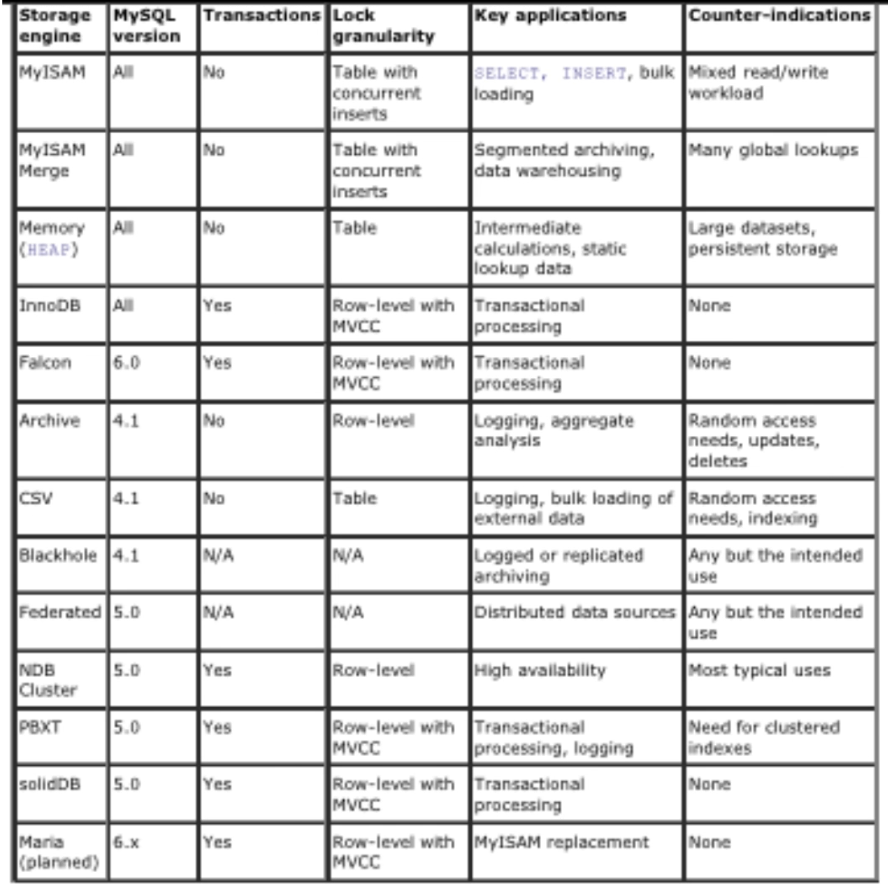
Simile all’INNODB

**PBXT**

E’ innovativo perché riduce il numero di commit

**Maria**

Vuole sostituire MyIsam

Adatto per tabelle di privilegi e tabelle temporanee

**Gis**

Definizione:

un sistema di informazione geografico è un sistema per la gestione dei dati alfanumerici e geometrici considerati nel loro contesto Geografico

vengono impiegati in aree come applicazioni ambientali, sistemi di trasporto, sistemi di pronto intervento ecc..

**A cosa servono?**

Servono per fare una mappatura di elementi geografici per una successiva analisi territoriali; per misurazioni relative a fattori ambientali; per modellizzare processi alternativi per la pianificazione territoriale; per il controllo dei cambiamenti dei processi ambientali ecce cc

Con “sistema di analisi geografico” estendiamo la capacità di consultazione di database trafizionali includendo la possibilità di analizzare i dati basandosi sulla loro ubicazione.

**Caratteristiche principali del gis**

**Geometria,Topolgia,Attributi,Georeference,Tematismi.**

Nel dettaglio:

**Geometria:** Un gis deve mantenere la rappresentazione geometrica della realtà e degli oggetti reali. in pratica per ogni oggetto il Gis deve rappresentare concetti come area ecc. deve inoltre avere la capacità di sviluppare calcoli sugli oggetti, come calcolo dell’area, del perimetro ecc.

Gli oggetti del mondo reale sono continui ma gli oggetti che disponiamo per gestire i dati sono discreti.

i formati dei dati devono tenere conto di questa cosa

i formati dei dati principali sono due: Raster e Vettoriali

* **vettoriale**: i dati geometrici sono memorizzati attraverso le coordinate dei punti significativi degli elementi stessi, un cerchio per esempio è memorizzato tramite le coordinate di centro e raggio.

Caratteristiche principali: più costoso,tempi di realizzazione lunghi, più dettagliato sul territorio, più adatto alla creazione di un database geografico.

* **raster**: prende la “figura” dell’oggetto vero e proprio e gli oggetti vengono memorizzati tramite la creazione di una griglia regolare nella quale ad ogni cella viene assegnato un valore alfanumerico che ne rappresenta un attributo (come per esempio che tonalità di grigio è un pixel in una foto in bianco e nero ecc)

Caratteristiche principali: Più adatto ad analisi territoriali mirate, richiede molto spazio in memoria, contiene una sola informazione per ogni cella.

**Topologia:** Lo studio delle relazioni spaziali tra gli oggetti.

In un GIS è importante poter definire delle regole topologiche sugli oggetti.

Una modalità per fare ciò sono gli indici spaziali, utili per le ricerche che vengono effettuate.

le interrogazioni spaziali sono diverse rispetto a quelle classiche. sono ad esempio “dammi le coordinate di, dammi tutti gli oggetti che sono alle coordinate xy, dammi tutti gli oggetti inclusi nel rettangolo che va da.. a.. ecc.

**Attributi**: le informazioni vere e proprie dei singoli oggetti reali

**Georeference:** la capacità di attribuire ad ogni elemento delle coordinate che corrispondono alle sue coordinate reali sulla terra. georeference: il software che gestisce la georeferenziazione.

**Tematismi:** tutte quelle informazioni in più che posso costruire con l’ausilio delle informazioni del GIS

**Database Spaziale** (database con coordinate geografiche)

Un database Spaziale è compost da tabelle bidimensionale, ogni tabella rappresenta una classe di dati geografici, ogni riga rappresenta un oggetto geografico identificato da un ID ed una o più colonne mantengono le coordinate.

**Un Geo database** è un database spaziale che contiene tutte le strutture per gestire un sistema GIS

**UN GIS** comprende anche tool e procedure per l’analisi dei dati spaziali.

Un problema: devo rappresentare degli oggetti geometrici su una mappa, rappresentando dunque questi oggetti su un piano

la terra però è rotonda (non proprio rotonda ma simil rotonda) dunque questo è un bel problema.

Perciò per rendere piana la terra **tramite una proiezione cartografica** devo decidere di “sacrificare” qualche tipo di proprietà geometrica.

Per esempio la mappa di mercatore perde completamente le proporzioni dei continenti ma preserva gli angoli.

**Organizzazione fisica dei dati**

**buffer =** apposita zona di memorizzazione centrale che nei dbms agisce da interazione tra memoria secondaria e principale

data location: principio di località spaziale

i dati sono organizzati in file, i file divisi in record (logici e fisici)

3 tipi di file:

**HEAP HASH, TREE (BTREE B+TREE)**

**Heap:**

* record inseriti in modo disordinato
* inserimenti sono efficienti ma cancellazioni ed aggiornamenti no
* scansione sequenziale (buono per letture sequenziali, cattivo per ricerche mirate)

Altre strutture sequenziali sono quelle ad array(clustered) e quelle ordinate (in base al valore di una chiave)

**Hash**

* La locazione fisica dipende dal valore della chiave
* i record sono ripartiti in bucket in base al valore della chiave
* ogni bucklet ha 1 o più blocchi ed è organizzato come un Heap
* i blocchi di un bucket sono collegati mediante puntatori a lista
* È utile che la funzione hash sia buona (ripartisce uniformemente i record nei bucket)

formula per trovare il numero di bucket B:

B = T/(fxF)

T numero di tuple, f = frazione di spazio fisico da utilizzare, F = fattore di blocco

In conclusione i file hash sono molto efficienti per l’accesso diretto basato sui valori della chiiave. non è efficiente per ricerche basate su intervelli.

**Alberi (E FILE A INDICE)**

* possono essere sia primari che secondari
* L’indice secondario di un fle è un file contenente il valore della chiave ed il relativo indirizzo fisico
* L’indice primario è un file che contiene la chiave primaria ed i dati.

**file a indice isamISAM = file con indice sparso**

* Record ordinati in crescenza
* File formato da coppie
* metodi di scansione: ricerca binaria o per interpolazione

**Btree**

* generallizzazione di ISAM
* il concetto dell’albero evita lo spreco di spazio((v,b) con b che può puntare ai blocchi di un altro file)
* sono strutture dinamiche e quindi anche efficienti per gli aggiornamenti.

ogni nodo coincide con un blocco ed ha piu successori, gli alberi devono essere bilanciati per funzionare bene. Ogni nodo contiene valori delle chiavi e puntatori a nodi figli. Sono altamente efficienti sulle ricerche, sugli aggioramenti ed inserimenti ma poco efficienti sugli accessi per intervalli di chiavi.

**B+Tree**

Come i Btree ma i nodi foglia vengono collegati tra di loro e sequenzialmente, ciò rende efficace la ricerca su intervalli

**Rtree**

Utilizzati per interrogare i dati in base alle loro coordinate geografiche

**CARATTERISTICHE COMUNI NEI DB**

In tutti i sistemi è possibile definire indici sia primari che secondari

Alcuni sistemi prevedono la possibilità di memorizzare in modo contiguo tuple di una tabella con gli stessi valori su un certo campo:🡪 “cluster”

In base ai motori di memorizzazione che scelgo gli alberi sono memorizzati diversamente.

in innoDB le primary key sono memorizzate come clustered index, gli indici secondari no

in myisam tutte le chiavi sono non clusterizzate (primarie o secondarie).

Gli indici **cluster** ordinano e archiviano le righe di dati della tabella in base ai valori di chiave, ovvero alle colonne incluse nella **definizione** dell'indice. Per ogni tabella è disponibile un solo indice **cluster**, poiché alle righe di dati è possibile applicare un solo tipo di ordinamento

**Trigger**

Le regole attive seguono il paradigma ECA (Event-Condition-Action).

I trigger sono un tipo di regole attive nei dbms come mysql.

i trigger dunque si attivano a seguito di un determinato evento, verificano se una determinata condizione è o meno verificata ed eventualmente svolge una serie di azioni, qualora ci dovessero essere le condizioni.

I trigger possono essere a livello di riga o a livello di singola istruzione. (For each row)

**Cursor**

Quando sono in un programma applicativo ed aziono una query, questa restituisce un insieme di record. Se avessi bisogno di “prelevare” un valore alla volta tra i record che mi vengono restituiti dovrei utilizzare un **Cursore** ossia uno strumento che permette ad un programma di accedere alle righe di una tabella, una alla volta.

**Stored procedure**

Insieme di istruzioni SQL memorizzate sul server, richiamate al bisogno (a differenza dei trigger possono presentare anche dei parametri)

**Index tuning**

Fare un accesso per indice equivale a random disk reads, altrimenti un full table scan è un sequential disk read

Effettuare delle random disk reads è notevolmente più lento che effettuare delle sequential disk reads, a causa del seek time.

Per tabelle di dimensioni ridotte o per query che ritornano un gran numero di righe è generalmente più efficace un FULL TABLE SCAN, ad eccezione di query sulla primary key o su tabelle innodb.

Un accesso per indice è preferibile se la quantità dei dati selezionati è inferiore al 10% del totale.

La percentuale varia a seconda dell’engine, per myisam è conveniente un accesso per indice fino al 30% delle righe selzionate.

Per tabelle piccole è preferibile un FTSCAN

Un indice multicolonna definito sulle colonne (t1..tn) può essere utilizzato per query che hanno una condizione di where su un qualsiasi prefisso delle colonne indicizzate

**HINT**

È un suggerimento all’ottimizzatore per guidarlo verso la scelta di un piano di esecuzione

non è uno standard SQL, permettono se corretti di avere un tempo di esecuzione ottimale

hint per gli indici: Use/Ignore/Force Index. (forzano l’ottimizzatore a valutare/Ignorare/usare per forza l’indice)

Se devo fare un sort su disco perdo davvero tanto in termini di performance

come distinct, union, group by, order by.

Creare un indice btree (che è una struttura ordinata) può andare a migliorare le performance

**Gestore delle interrogazioni**:

Quando giunge una query viene effettuata un’analisi lessicale, semantica ed algebrica usando un “dizionario dei dati". La query viene poi tradotta in algebra relazionale, dopodiché viene fatta un’ottimizzazione basata sui costi.

Il sistema si avvale dei profili delle relazioni (meta-tabelle usate per valutare i costi) per ottimizzare la query. Le informazioni contenute in queste meta-tabelle sono, ad esempio: Cardinalità di ciascuna relazione, dimensioni delle tuple, dimensioni dei valori eccetera.

In sostanza dunque una query viene analizzata e tradotta in una serie di “Strategie” per arrivare ad una soluzione, il gestore utilizza i profili delle relazioni per confrontare tutte le “Strategie” e utilizza poi quella meno costosa.

**Rappresentazione interna delle interrogazioni:**

Alberi: Le foglie sono i dati (relazioni/file) e i nodi intermedi sono gli operatori.

EX sel A=10(R1JoinR2) =

Sel a=10

|

Join

/ \

R1 R2

**Operazioni:**

Scansione (Full Table Scan)

Accesso diretto (Chiave primaria o Indice Secondario) (Btree B+Tree Hash)

Ordinamento

Join (operazione più costosa, i metodi usati sono: nested loop, merge-scan, hash based)

**Nested loop**: Faccio una scansione su una tabella esterna, prelevo l’attributo su cui faccio il join e si cercano le tuple che corrispondono a quel valore nella tabella interna.

**Merge-Scan: (SOLO SE HO TABELLE ORDINATE)** vengono effettuate due scansioni parallele e mi fermo quando i due scan incrociano uno stesso valore, quando c’è uno stop unisco le tabelle

**HASH JOIN:** Viene scelta una funzione hash per memorizzare una copia delle due tabelle. Queste due tabelle vengono divise in bucket per valore su cui sto facendo il join. Ecc…

Il problema dell’ottimizzazione dei costi dunque deve scegliere:

Le operazioni da eseguire, L’ordine delle stesse ed i dettagli (ad esempio quale metodo di Join)

Viene creato dunque un albero di decisione con le varie alternative, ossia diversi piani di esecuzione. Si valuta in modo approssimativo il costo di ogni piano e viene eseguito il piano meno costoso

L’ottimizzatore trova sempre una buona soluzione che però non è necessariamente **OTTIMA**

**Query Performance Optimization**

Per ottimizzare le query bisogna capire come sono eseguite

Quello che bisogna fare è verificare:

Che i dati su cui si fa la query siano tutti necessari, se i dati con cui il server lavora sono più del dovuto, il numero di righe esaminate, il numero di righe tornate.

Il client fa una query, data in pasto prima alla query cache, poi al parser e poi al preprocessor, che genera l’albero dei piani di esecuzione ed il query optimizer poi sceglie il piano da usare. Il query execution engine, che in base allo storage engine scelto per costruire la tabella svolge la query e poi da al client il risultato.

Nella query cache vengono memorizzati risultati di query già ottenuti, quando viene effettuata una query la prima cosa è cercare con metodo hash se il risultato è già presente all’interno della cache.

Nel parser si verificano in primis i privilegi, poi la query viene spezzata in token di algebra relazionale e viene costruito il parse tree. Il preprocessor effettua poi controlli semantici non risolti dal parser

Nel query optimizer potrebbero esserci dei malfunzionamenti, le ragioni sono:

Statistiche sbagliate, Non considerare la concorrenza di altre query, oppure alcuni piani di esecuzioni non vengono considerati.

L’execution plan non è un byte code ma un semplice piano di esecuzione.

L’execution engine usa il piano di esecuzione come struttura dati, invoca i metodi implementate dalle API e utilizza gli storage engines

**CASI IN CUI L’OTTIMIZZATORE NON FUNZIONA BENE**

Query nidificate,Union,Between,min(),max().