Università degli Studi di Padova
Dipartimento di Scienze Statistiche
Corso di Laurea Triennale in
STATISTICA E TECNOLOGIE INFORMATICHE

RELAZIONE FINALE

TECNICHE DI OTTIMIZZAZIONE DI QUERY SQL

SQL QUERY'S OPTIMIZATION TECHNIQUES

Relatore Prof. Loris Nanni
Dipartimento di Ingegneria dell'Informazione

Laureando: Pietro Di Bianca
Matricola N 1010241

Anno Accademico 2012/2013

A mia zia Emanuela

INDICE

INTRODUZIONE

Nei sistemi di gestione di basi di dati l'utente formula generalmente le proprie interrogazioni utilizzando un linguaggio ad alto livello (come SQL (Structured Query Language)) di tipo dichiarativo: l'utente specifica cosa vuole piuttosto che come calcolare il risultato dell'interrogazione. Pertanto è lasciato al sistema il compito di individuare una strategia efficiente per calcolare il risultato in relazione agli obiettivi prefissati e nel rispetto dei vincoli di sistema, cioè di eseguire quel processo che va sotto il nome di ottimizzazione delle interrogazioni. Il termine "ottimizzazione" è, in realtà, impreciso in quanto il sistema non individua in generale la strategia ottima, ma una strategia ragionevolmente efficiente per eseguire l'interrogazione. Infatti per trovare la strategia ottima può essere necessario impiegare un tempo eccessivo (se non nel caso di interrogazioni molto semplici) e disporre di informazioni su come i file sono implementati e sul contenuto stesso della base di dati, e queste informazioni potrebbero non essere disponibili nel catalogo del sistema.

Ci sono due tecniche principali di ottimizzazione: la prima utilizza regole euristiche, basate su proprietà degli operatori dell'algebra relazionale, per effettuare una trasformazione di una interrogazione in una equivalente (cioè che produce lo stesso risultato) ma più efficiente (che richiede un tempo di esecuzione non superiore a quello richiesto dall'interrogazione iniziale); la seconda effettua una stima dei costi di esecuzione delle diverse strategie e sceglie quella con costo minore.

I sistemi utilizzano in genere una combinazione delle due strategie.

L'ottimizzazione delle prestazioni è l'insieme delle attività il cui scopo è quello di massimizzare le performance di un sistema in relazione agli obiettivi prefissati e nel rispetto dei vincoli di sistema.

Gli obiettivi da raggiungere variano a seconda del tipo di sistema considerato:

- Sistemi OLTP (OnLine Transaction Processing) [4]: si occupano della massimizzazione del throughput (la quantità di lavoro svolto in un'unità di tempo). Questi sistemi, quindi, devono garantire transazioni in tempi brevi (pochi secondi), un rapido aggiornamento del database e una buona tolleranza ai guasti. L'obiettivo principale è quello della garanzia di integrità e sicurezza della transazioni.

- Sistemi OLAP (OnLine Analytical Processing) [8][16]: si occupano della minimizzazione del response time (il tempo di risposta all'utente). Questi sistemi designano un insieme di tecniche software per l'analisi interattiva e veloce di grandi quantità di dati, che è possibile esaminare in modalità piuttosto complesse. L'obiettivo principale è quello di raggiungere la miglior performance nella ricerca.

L'ottimizzazione delle prestazioni è un fattore di cui bisogna sempre tenere conto in tutte le fasi della progettazione e dell'implementazione di una base di dati:

- I progettisti delle basi di dati ne devono tener conto attraverso una corretta modellazione;
- I progettisti delle applicazioni stabiliscono il giusto compromesso tra complessità ed efficienza delle funzioni;
- Gli implementatori delle applicazioni attraverso un attento utilizzo delle risorse di sistema;
- Gli amministratori di database, che devono continuamente verificare le necessità mutate degli utenti e identificare eventuali colli di bottiglia non individuabili durante la fase di realizzazione.

Per poter svolgere il proprio compito, l'amministratore deve conoscere al meglio le modalità di funzionamento del DBMS (DataBase Management System): quali indici usare; quali piani di accesso; di quante risorse necessita una certa operazione; su quali componenti si può intervenire, ecc.

Nel capitolo 1 verrà descritto che cos'è e a cosa serve l'ottimizzazione delle query SQL; si illustreranno le varie tecniche di ottimizzazione delle query SQL, quali: ottimizzazione algebrica e relative regole di equivalenza; ottimizzazione sintattica e semantica; ottimizzazione euristica con un esempio di algoritmo euristico; ottimizzazione cost-based; si descriveranno i tipi di indici di cui si può far uso nei vari sistemi di ottimizzazione (indici b-tree, bitmap, clustered, unclustered, primary, secondary, multi-level, sparsi); si illustreranno i tre algoritmi di join più diffusi (nested loops, sort merge, hash).
Nel capitolo 2 si mostreranno tre esempi pratici di query SQL ottimizzate riscrivendo solamente il plan di esecuzione della query. In questo capitolo, inoltre, verrà illustrata una versione semplificata del database utilizzato nell'attività di stage all'IZSVe (Istituto Zooprofilattico Sperimentale delle Venezie).
Nel capitolo 3, infine, sono presenti le conclusioni.

CAPITOLO I - PROBLEMA AFFRONTATO

In questo capitolo verranno illustrati i seguenti argomenti: perché si usa e perché serve ottimizzare le query SQL; quali sono le varie tecniche di ottimizzazione (euristica, algebrica, sintattica, ecc.); quali sono gli algoritmi di join più diffusi ed utilizzati; quali sono le tipologie di indici.

1.1 DESCRIZIONE DEL PROBLEMA

L'ottimizzazione SQL prevede una trasformazione delle espressioni SQL che produca esattamente lo stesso risultato delle query di partenza (non ottimizzata), ma solitamente in tempi più brevi.

Quando una query viene mandata al DBMS (DataBase Management System) [10][11], questo effettua diversi passaggi per trasformare la query in sequenza di record.

Quando la query viene ricevuta dal DBMS, il primo passaggio a cui è sottoposta è il "parsing", ossia la separazione della query nelle sue componenti.

Il processo di parsing ha principalmente due funzioni:

1. Verificare la correttezza lessicale e sintattica della query;
2. Identificare tutte le parti che compongono la query.

Ogni parte identificata dal parser viene memorizzata in una struttura interna al DBMS, solitamente nella forma di un albero di esecuzione (query tree), cioè una rappresentazione che può essere facilmente manipolata dal sistema interno, aggiungendo, rimuovendo o spostando le sue componenti.

A questo punto l'albero viene passato all'ottimizzatore che sceglierà quali algoritmi usare per implementare gli operatori di algebra relazionale presenti nell'albero di esecuzione.

Lo scopo della fase di ottimizzazione della query è quello di produrre un piano di esecuzione il più efficiente possibile, basandosi su quanto è specificato dall'albero di esecuzione della query.

Un "ottimizzatore" teoricamente può produrre un piano di esecuzione "ottimale" per ogni query, in realtà questo finirà per produrre un piano solamente accettabile per la maggioranza delle query. Questo perché il numero di combinazioni possibili in una Join aumenta geometricamente e nello stesso modo aumenta la complessità della query.
Senza l'utilizzo di tecniche di "pruning" o altri metodi euristici per limitare il numero di combinazioni valutate, il tempo richiesto per ottenere una reale ottimizzazione della query risulta assolutamente inaccettabile. In molti casi l'ottimizzatore sceglie una query meno efficiente perché la selezione di una query più efficiente richiede più tempo che l'esecuzione della query inefficiente.
I vari database utilizzano di solito differenti tecniche di ottimizzazione per ottenere una certa efficienza nel piano di esecuzione.

1.2 APPLICAZIONI

L'ottimizzazione delle query è una funzione di molti sistemi di gestione di database relazionali, soprattutto se si ha a che fare con database di medio-grandi dimensioni.
Se si cercano di estrarre dei dati (mediante query SQL) da un database di piccole dimensioni, i benefici dell'ottimizzazione, soprattutto in termini di tempo di risposta dei risultati, non si notano in quanto le query forniscono i record richiesti in meno di 1 secondo.
Se, invece, si lavora con database di medio-grandi dimensioni, allora l'ottimizzazione gioca un ruolo fondamentale nel tempo di risposta dei risultati, in quanto query che richiedono anche parecchi minuti per essere eseguite, ottimizzandole, possono velocizzarsi anche del 90-95% (vedi i tre esempi di query ottimizzate nel Capitolo II - Risultati sperimentali). Questo perché le strutture dei database sono, in moltissimi casi, complesse e, per le query non semplici, i dati richiesti da una query possono essere raccolti accedendo in diversi modi attraverso differenti strutture dati e in diversi ordini. Ogni modo diverso, in generale, richiede un tempo di elaborazione diverso, quindi la stessa query può essere eseguita in pochi secondi o in diversi giorni.
L'obiettivo, quindi, è quello di trovare il modo per eseguire una query nel minor tempo possibile. Poiché SQL è un linguaggio dichiarativo (si specifica cosa deve essere estratto ma non in che modo), ci sono tipicamente un gran numero di modi alternativi per

eseguire una interrogazione. Data questa gran vastità di possibilità per eseguire una query, trovare il modo ottimale per eseguirla è molto complesso, richiede molto tempo, può essere molto costoso e spesso è praticamente impossibile. A questo punto entra in gioco Query Optimizer (presente in ogni programma di gestione di database) [23] che analizza alcuni dei diversi piani di esecuzione, corregge eventuali piani prestabiliti per eseguire la query e restituisce ciò che considera la migliore alternativa.
Spesso questi piani restituiti dal Query Optimizer sono imperfetti, quindi gli utenti e gli amministratori del database devono esaminare manualmente i piani ed ottimizzarli ulteriormente per ottenere prestazioni migliori.

1.3 STATO DELL'ARTE

Al giorno d'oggi, l'efficienza di un'applicazione dipende molto dall'efficienza del database sottostante. Per far sì che un database sia efficiente, bisogna che le query siano le più efficienti possibili.
Per rendere le query le più efficienti possibili (cioè ottimizzarle), ci sono varie tecniche, le cui principali saranno discusse nei prossimi paragrafi.

1.3.1 TIPOLOGIE DI OTTIMIZZAZIONE

Ottimizzazione algebrica

L'ottimizzazione algebrica [7][11] ha lo scopo di trovare un'espressione che sia equivalente all'espressione data e possa essere eseguita in modo più efficiente. Per passare da un'espressione dell'algebra relazionale all'altra si sfruttano le regole di equivalenza degli operatori:

1) Cascata di selezioni: una condizione di selezione congiuntiva può essere spezzata in una sequenza di singole operazioni σ

$$\sigma_{c1 \text{ AND } c2 \text{ AND } \ldots \text{AND } cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$$

2) Commutatività della selezione:

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$$

3) Cascata di proiezioni: in presenza di una sequenza di operatori π si possono ignorare tutte le proiezioni fuorchè l'ultima

$$\pi_{Lista1}(\pi_{Lista2}(...(\pi_{Listan}(R))...)) \equiv \pi_{Lista1}$$

4) Commutatività di σ rispetto a π: se la condizione di selezione c coinvolge solo gli attributi appartenenti alla lista di proiezione A1,…,An, le due operazioni possono essere commutate

$$\pi_{A1,A2,\ldots,An}(\sigma_c(R)) \equiv \sigma_c(\pi_{A1,A2,\ldots,An}(R))$$

5) Commutatività di ►◄:

$$R \blacktriangleright\blacktriangleleft_c S \equiv S \blacktriangleright\blacktriangleleft_c R$$

6) Commutatività di ►◄ rispetto a σ: se tutti gli attributi della condizione di selezione c coinvolgono solamente gli attributi di una delle relazioni su cui viene eseguito il join, le due operazioni possono essere commutate

$$\sigma_c (R \blacktriangleright\blacktriangleleft S) \equiv \sigma_c (R) \blacktriangleright\blacktriangleleft S$$

Alternativamente se la condizione c può essere scritta, sfruttando la regola 1, come c1 AND c2 dove c1 e c2 operano rispettivamente su attributi di R e S, allora è possibile effettuare la seguente trasformazione

$$\sigma_c (R \blacktriangleright\blacktriangleleft S) \equiv \sigma_{c1} (R) \blacktriangleright\blacktriangleleft \sigma_{c2} (S)$$

7) Commutatività di ►◄ rispetto a π: sia la lista di proiezione L = A1,…,An,B1,…,Bn dove A1,…,An appartengono ad R e B1,…,Bn appartengono a S, se la condizione di join c coinvolge solamente attributi in L, allora le due operazioni possono essere commutate

$$\pi_L (R \blacktriangleright\blacktriangleleft_c S) \equiv \pi_{A1,\ldots,An} (R) \blacktriangleright\blacktriangleleft_c \pi_{B1,\ldots,Bn} (S)$$

Se la condizione di join include oltre a quelli in L ulteriori attributi di R e S, ossia An+1,…,An+k,Bn+1,…Bn+k, allora questi vanno mantenuti sino al join e quindi è necessaria un'ulteriore operazione di proiezione

$$\pi_L (R \blacktriangleright\blacktriangleleft_c S) \equiv \pi_{A1,\ldots,An,An+1,\ldots,An+k} (R) \blacktriangleright\blacktriangleleft_c \pi_{B1,\ldots,Bn,Bn+1,\ldots,Bn+k} (S)$$

8) Associatività di ►◄:

$$(R \blacktriangleright\blacktriangleleft_{c1} S) \blacktriangleright\blacktriangleleft_{c2} T \equiv R \blacktriangleright\blacktriangleleft_{c1} (S \blacktriangleright\blacktriangleleft_{c2} T)$$

9) Composizione di X e σ: se la condizione c di un'operazione di selezione che segue un'operazione prodotto cartesiano rappresenta una condizione di join, allora

$$\sigma_c (R \times S) \equiv R \blacktriangleright\blacktriangleleft_c S$$

Ottimizzazione euristica

L'ottimizzazione euristica [21] è un meccanismo basato su regole specifiche per produrre un piano di esecuzione efficiente. Dato che la query ricevuta è una struttura definita, ogni nodo dell'albero viene mappato direttamente in un'espressione algebrico-relazionale. La funzione euristica è quindi applicata per ridurre l'espressione ai suoi termini di base, ottenendo quindi una rappresentazione più efficiente. Usando un'espressione algebrica, si assicura anche che nessuna delle necessarie informazioni richieste per esaminare i dati verrà persa durante il processo.

Un esempio molto diffuso di algoritmo euristico di ottimizzazione basato su regole è il seguente:

1. Scomporre i predicati di selezione congiuntiva utilizzando la regola (1) di equivalenza tra espressioni di algebra relazionale. In questo modo si garantisce un maggior grado di libertà per lo spostamento delle operazioni di selezione in modo da anticiparle quanto più possibile
2. Commutare le operazioni di selezione rispetto alle altre operazioni (regole (2)(4) e (6)), in modo da anticiparle quanto più possibile conformemente a quanto permesso dagli attributi coinvolti nelle selezioni (push-down selezione)
3. Usando la regola (9) si sostituiscano con operazioni di join le operazioni di prodotto cartesiano seguite da operazioni di selezione
4. Usando le regole (5) e (8) si modifichi la sequenza di esecuzione delle operazioni di join in modo da anticipare il join su relazioni in cui insistono operazioni di selezione più restrittive
5. Sfruttando le regole (3)(4)(7) si anticipino quanto più possibile le liste degli attributi di proiezione creando, quando necessario nuove operazioni di proiezione. Dopo ogni operazione di proiezione, dovrebbero essere mantenuti

solamente gli attributi necessari alle operazioni successive e alla generazione del risultato dell'interrogazione (push-down proiezioni)

Ottimizzazione sintattica

L'ottimizzazione sintattica [11] si appoggia pesantemente sulla comprensione dell'utente sia del database, sia della distribuzione dei dati tra le varie tabelle. In parole povere, si fa affidamento sul fatto che l'utente ha già fatto delle scelte in base alle proprie conoscenze. L'ottimizzatore cerca di migliorare l'efficienza della query scegliendo gli indici appositi tra quelli disponibili per ogni singola tabella.

Questo tipo di ottimizzazione è estremamente efficiente quando si accede a dati in un ambiente sostanzialmente statico e quando l'utente sa quello che sta facendo. Se il database ed i suoi contenuti cambiano in maniera molto varia o se l'utente non sa esattamente cosa richiedere (la query non è già ottimizzata di suo), allora si possono avere risultati pessimi.

Ottimizzazione semantica

Questo tipo di ottimizzazione [12] non e' ancora entrata nel novero delle ottimizzazioni "standard", ma e' oggetto di ricerche. Questo metodo si basa sulla conoscenza della struttura del sottostante database per ignorare o eliminare parti della query che non ritornerebbero risultato o non ritornerebbero risultati utili.

Ottimizzazione "cost-based"

Per eseguire questo tipo di ottimizzazione [19], l'ottimizzatore richiede informazioni specifiche relative alle informazioni del database stesso. Queste informazioni sono strettamente dipendenti dal sistema e possono includere la dimensione e la strutture dei file, la disponibilità di indici, la percentuale di record da recuperare da ogni tabella, e così via.

Dato che lo scopo di ogni ottimizzazione è quello di ridurre al minimo il tempo di estrazione dei record, l'ottimizzazione basata sui costi utilizza le informazioni sulla struttura del database e la distribuzione dei dati per assegnare un "costo" stimato, misurato in termini di tempo, numero di record da estrarre da ogni tabella e numero di accessi dalla memoria centrale alla memoria di massa (e viceversa) per ogni

operazione. Valutando la somma totale di questi "costi" è possibile selezionare la sequenza più efficiente di estrazione dei dati.
Ovviamente, i "costi" assegnati saranno più o meno validi a seconda delle informazioni che il sistema mantiene sulla composizione delle tabelle, dei file ecc. Mantenere aggiornate queste informazioni occupa tempo e risorse, quindi ogni sistema memorizza un blocco di informazioni e poi lo aggiorna (ricostruendolo) di tanto in tanto. Se sul database vengono effettuate molte operazioni che coinvolgono la distruzione totale e la ricostruzione da zero di svariate tabelle, l'efficienza di questo metodo di ottimizzazione e' seriamente compromessa.
Per valutare i costi delle varie strategie di esecuzione occorre tenere traccia di alcune informazioni necessarie alle funzioni di costo utilizzate dai DBMS.
Per le tabelle, queste informazioni sono:

- Numero di tuple della tabella
- Dimensione delle tuple
- Numero di blocchi di disco utilizzati per memorizzare la tabella

Per le colonne, queste informazioni sono:

- Valore minimo
- Valore massimo
- Numero di valori distinti
- Numero di valori nulli
- Lunghezza media
- Distribuzione dei dati (istogrammi)

Per gli indici, infine, queste informazioni sono:

- Numero di foglie
- Profondità

Nelle informazioni di cui bisogna tenere traccia per le colonne si sono nominati gli istogrammi. Un istogramma [11][20] fornisce una rappresentazione semplificata della distribuzione dei valori per gli attributi delle relazioni di una specifica istanza di database. La rappresentazione è ottenuta suddividendo lo spazio dei valori in più raggruppamenti o intervalli (bucce) e stimando, per ognuno di essi, la frequenza media con cui i valori al loro interno si presentano nel database. Ci sono principalmente due modi per poter suddividere questo spazio di valori:

- Equi-width: si suddivide lo spazio dei valori in intervalli di egual dimensione;
- Equi-height: la suddivisione è tale che la somma delle frequenze dei valori degli attributi associati a ciascun raggruppamento sia uguale (indipendentemente dal numero di valori che saranno contenuti nell'intervallo).

1.3.2 INDICI

Un indice [9] è una struttura dati accessoria per accedere ai dati delle relazioni sulla base di una chiave di ricerca. L'idea di base è quella di associare ai dati una tabella nella quale l'i-esima tupla memorizza una coppia del tipo (k_i,p_i), dove:

- K_i è un valore della chiave di ricerca su cui l'indice è costruito
- P_i è un riferimento al record con valore di chiave k_i

I DBMS dispongono di più tipologie di indici ognuna adatta ad un particolare tipo di dato o ad un particolare tipo di accesso:

- Indici B^+tree
- Indici Bitmap

Un indice B^+tree [2] per un attributo c di una tabella R è un albero bilanciato che consente accessi associativi alle tuple di R in base ai valori della chiave c. Le foglie dell'albero, collegate tra loro in sequenza, contengono i puntatori ai record (RID = Row IDentifier), mentre i nodi interni costituiscono una mappa per consentire una rapida localizzazione delle chiavi.

L'indice B^+tree è una struttura bilanciata, ossia garantisce che l'altezza h dell'albero sia sempre costante per tutti i percorsi dalla radice alle foglie. A tal fine vengono utilizzate delle procedure di bilanciamento che si innescano a fronte di cancellazioni ed inserimenti.

La dimensione di un B^+Tree dipende da molteplici fattori (lunghezza dei separatori, lunghezza delle RID, dimensione della pagina di disco) e può variare a causa delle operazioni di bilanciamento. Una stima approssimativa della dimensione può essere fornita dal numero delle foglie dell'albero che è pari a:

$$NL = [NK * len(k) + NR * len(p)]/(D*u)$$

dove NK è il numero di valori distinti di chiave, NR è il numero delle tuple da indicizzare, len(k) e len(p) rappresentano rispettivamente la lunghezza delle chiavi

delle tuple e dei puntatori ai blocchi dati, infine u rappresenta il fattore di riempimento dei nodi.

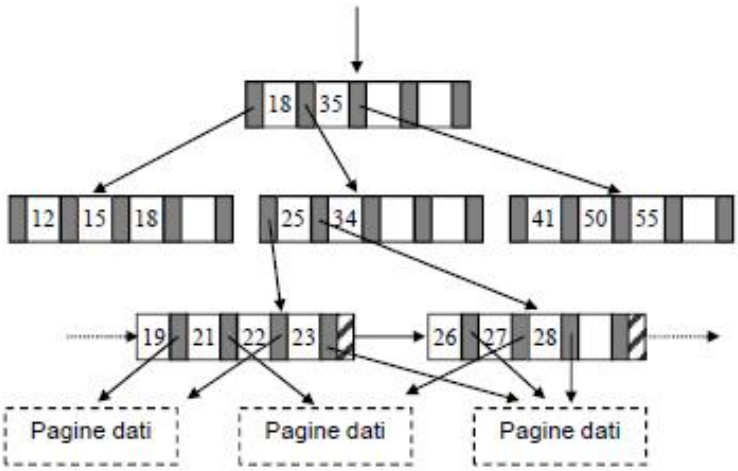

Figura 1: rappresentazione indice B$^+$tree

Un'altra categoria molto importante di indici è rappresentata dai bitmap [5]. Un indice bitmap su un attributo è composto da una matrice di bit contenente:

- Tante righe quante sono le tuple della relazione
- Tante colonne quanti sono i valori distinti di chiave dell'attributo

Il bitmap (i,j) è posto a TRUE se nella tupla i-esima è presente il valore j-esimo.
I vantaggi di questo tipo di indice sono i seguenti:

- Lo spazio richiesto su disco può essere molto ridotto
- I/O molto basso perché vengono letti solo i vettori di bit necessari
- Sono ottimi per interrogazioni che non richiedono l'accesso ai dati
- Permettono l'utilizzo di operatori binari per l'elaborazione dei predicati
- Sono adatti ad attributi con una ridotta cardinalità poiché ogni nuovo valore distinto di chiave richiede un ulteriore vettore di bit

Un indice può essere utilizzato per eseguire un'interrogazione SQL se l'attributo su cui è costruito:

- Compare nella clausola WHERE
- È contenuto in un fattore booleano
- Il fattore booleano è argomento di ricerca attraverso indice
- Compare in un ORDER BY o GROUP BY

Gli indici [5], quindi, sono delle tabelle speciali associate alle tabelle dati, che vengono poi utilizzate durante le operazioni che agiscono su queste ultime. Contrariamente a molti linguaggi gestionali mirati al trattamento dei file, SQL permette di creare più indici su una stessa tabella. Tuttavia quando si crea un indice, SQL memorizza, oltre ai dati della tabella, anche quelli dell'indice. Quindi ogni variazione alla tabella comporta una variazione agli opportuni puntatori alle righe della tabella e non è detto che ciò sia sempre conveniente. Ad esempio se una tabella cambia spesso dati, allora la presenza di molti indici rallenta il lavoro di aggiornamento. Di seguito è fornita una lista che aiuta a valutare quando è opportuno usare gli indici:

- Gli indici occupano spazio su disco.
- Possiamo ottimizzare le query SQL, tramite l'uso di indici, se queste forniscono modeste quantità di dati (non più del 23%). In caso contrario, allora gli indici non migliorano la velocità di lettura delle query.
- Gli indici di piccole tabelle non migliorano le prestazioni.
- I migliori risultati si ottengono quando le colonne su cui sono stati costruiti gli indici contengono grandi quantità di dati o tanti valori NULL.
- Gli indici rallentano le operazioni di modifica dei dati. Di questo bisogna tenerne conto quando si effettuano molti aggiornamenti. Infatti prima di un massiccio aggiornamento del database sarebbe meglio distruggere tutti gli indici e poi ricrearli.
- Se la condizione delle query riguarda un solo campo allora è opportuno usare un indice composto da quella sola colonna. Se la condizione delle query riguardano la combinazione di più campi allora è opportuno creare un indice contenente quei campi.

Esistono diverse tipologie di indici; la prima distinzione è tra

- Indici ordinati: i valori di chiave vengono ordinati, in modo tale da poter essere reperiti più efficientemente
- Indici hash: si usa una funzione hash per determinare la posizione dei valori di chiave; questi indici tuttavia non forniscono prestazioni soddisfacenti per ricerche di intervallo

Gli indici ordinati [22], a loro volta, si suddividono in:

- Clustered o unclustered
- Primary o secondary
- Dense o sparse
- Single-level o multi-level

Gli indici clustered sono costruiti sul campo su cui i record nel file dati sono mantenuti ordinati, altrimenti sono detti unclustered. Per ogni relazione si possono costruire un numero arbitrario di indici unclustered, mentre si può costruire al massimo un indice clustered.

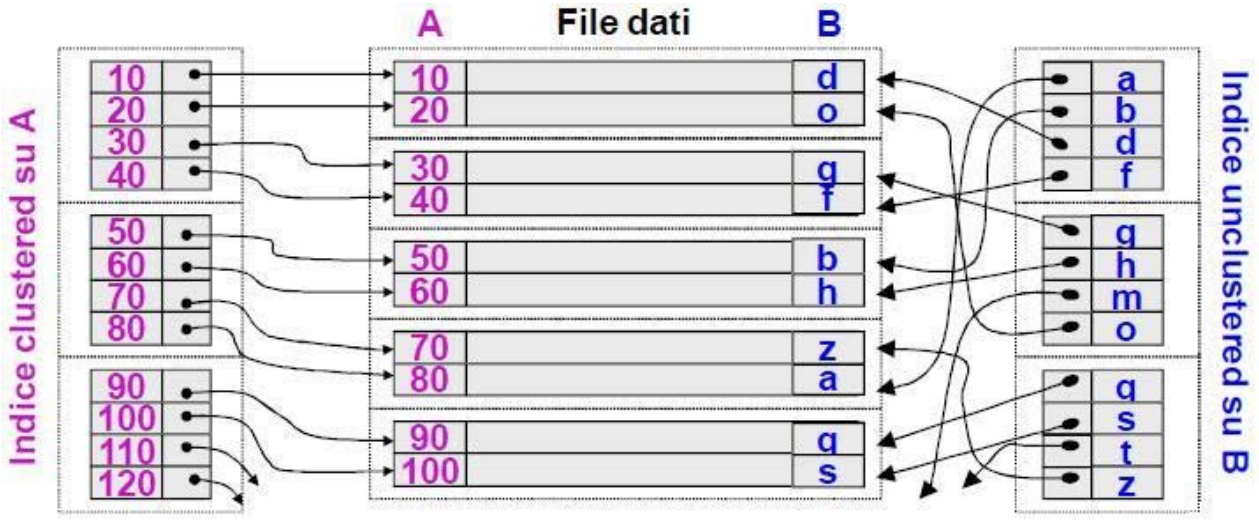

Figura 2: rappresentazione indice clustered ed indice unclustered

Un indice è detto primary (primario) se è costruito su un campo a valori non ripetuti (chiave relazionale), altrimenti è detto secondary (secondario) [18].

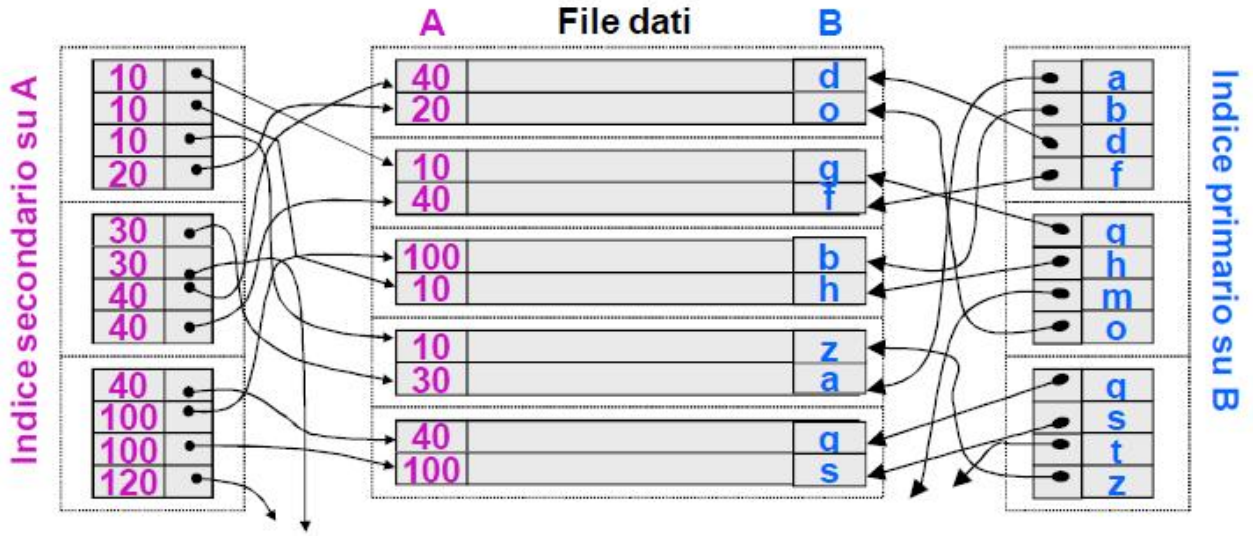

Figura 3: rappresentazione indice primary ed indice secondary

Per evitare di replicare inutilmente i valori di chiave, la soluzione più comunemente adottata per gli indici secondari consiste nel raggruppare tutte le coppie con lo stesso valore di chiave in una lista di puntatori.

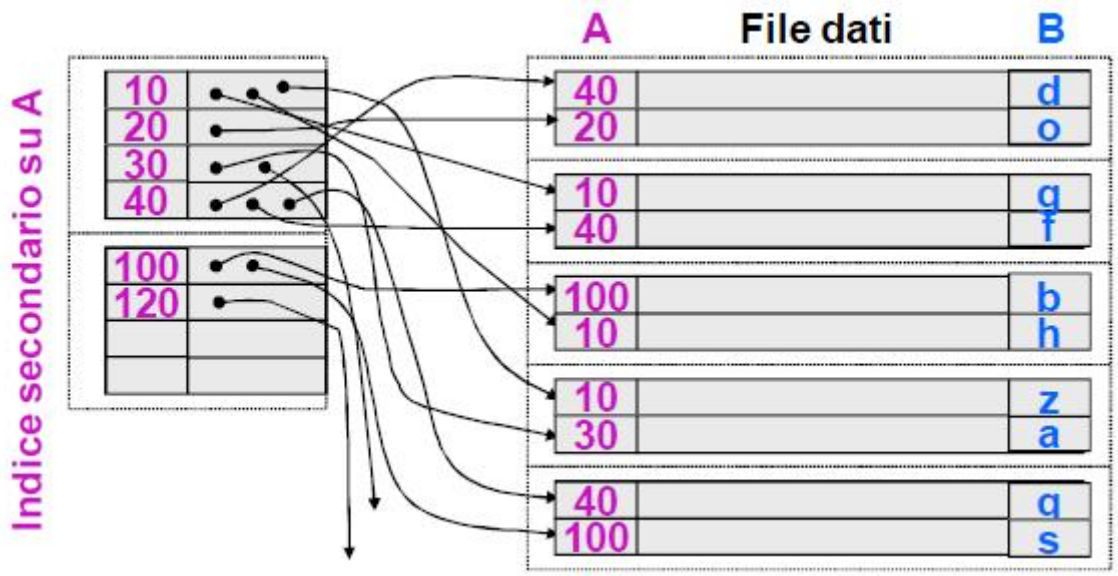

Figura 4: rappresentazione indice secondary raggruppato per chiave

Negli indici dense (densi) il numero di puntatori è uguale al numero di record del file dati; negli indice sparse (sparsi) è minore (tipicamente uno per pagina dati).
Gli indici appena descritti sono tutti single-level (o flat). È tuttavia possibile indicizzare l'indice usando un indice sparso (e così via in modo ricorsivo) in modo da creare una struttura multi-level.

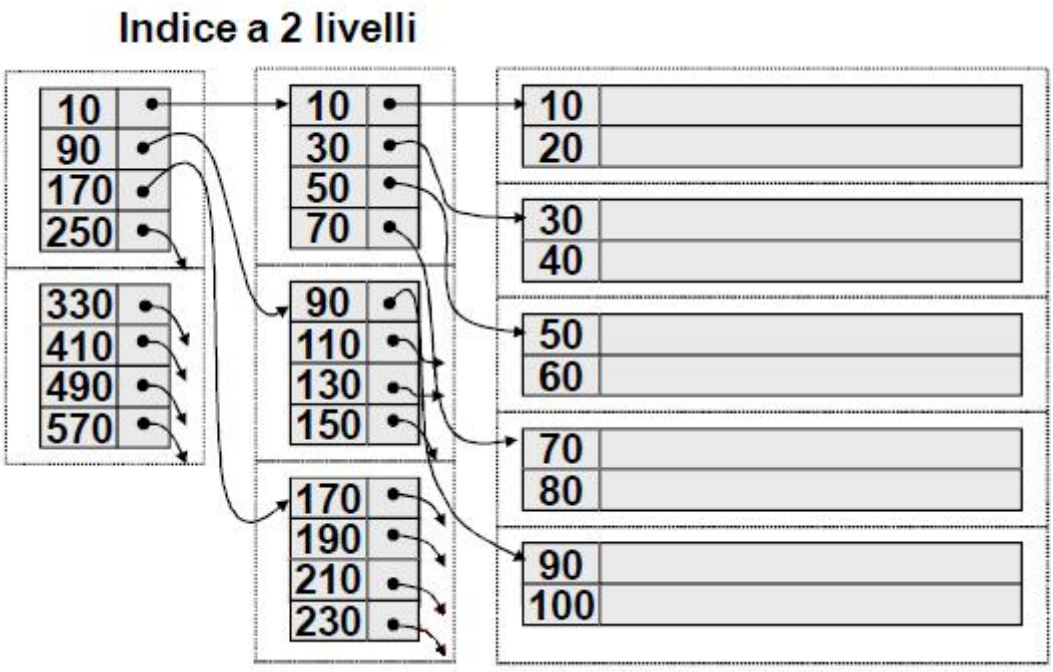

Figura 5: rappresentazione della struttura di un indice multi-level

1.3.3 ALGORITMI DI JOIN

L'operatore di join [1][6][15], nella sua versione base, ha come input due insiemi di tuple (due relazioni) e come output un insieme in cui ogni tupla è ottenuta combinando, sulla base di uno o più predicati di join, una tupla del primo insieme con una tupla del secondo insieme.

Nell'implementazione più semplice il join di due relazioni R e S prevede il confronto di ogni tupla di R con ogni tupla di S, con complessità $O(NT_R \text{ X } NT_S)$.

Esistono moltissime implementazioni del join che mirano a sfruttare al meglio le risorse del sistema e le (eventuali) proprietà degli insiemi di tuple in ingresso per evitare di eseguire tutti i possibili NR(R)*NS(S) confronti. Le implementazioni più diffuse si riconducono ai seguenti operatori fisici:

- Nested Loops Join
- Sort Merge Join
- Hash Join

Si noti che, benché dal punto di vista logico il join è un'operazione commutativa, fisicamente invece c'è una chiara distinzione, che influenza anche le prestazioni, tra operando sinistro (o esterno, outer) e operando destro (o interno, inner).

Nested Loop Join

È l'algoritmo di join [17] più semplice e deriva direttamente dalla definizione dell'operazione. Una delle due relazioni coinvolte è designata come esterna e l'altra come interna. Supponendo di operare usando due relazioni R e S (R esterna e S interna) e di essere in presenza di due predicati locali Fr e Fs (rispettivamente su R e S), l'algoritmo procede ricercando per ogni tupla di R che soddisfa Fr, tutte le tuple di S che soddisfano il predicato Fs ed il predicato di join Fj.

Il costo di esecuzione si esprime, in generale, come:

$$C_a(R) + ET_R \times C_a(S)$$

dove

- $C_a(R)$ rappresenta il costo di accesso alla relazione R;
- $ET_R = f_R \times NT_R$ è il numero atteso di tuple residue di R;
- $C_a(S)$ è il costo di accesso alla relazione S, per ogni tupla residua di R.

Alcuni casi notevoli di costo di esecuzione sono:

1. In assenza di predicati locali sulla relazione esterna: $C_a(R) + NT_R \times C_a(S)$
2. Facendo uso di scan sequenziali: $NP_R + ET_R \times NP_S$
3. Se valgono sia 1 sia 2, allora il costo diventa: $NP_R + NT_R \times NP_S$

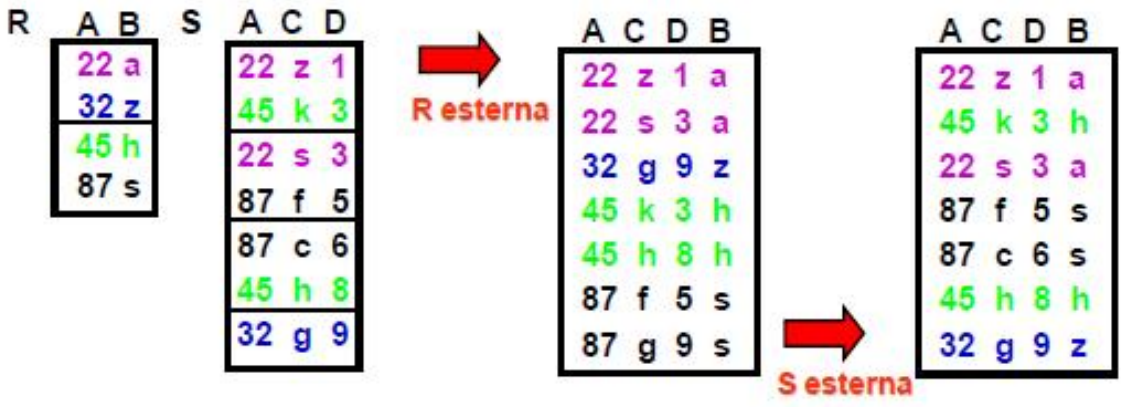

Figura 6: rappresentazione del funzionamento del Nested Loop Join

Sort Merge Join

Il Sort Merge Join è applicabile quando entrambi gli insiemi di tuple in input sono ordinati sugli attributi di join.

Per due tabelle R ed S ciò è possibile se, oltre ad essere fisicamente ordinate sugli attributi di join, esiste un indice per ogni tabella costruito sugli attributi di join.

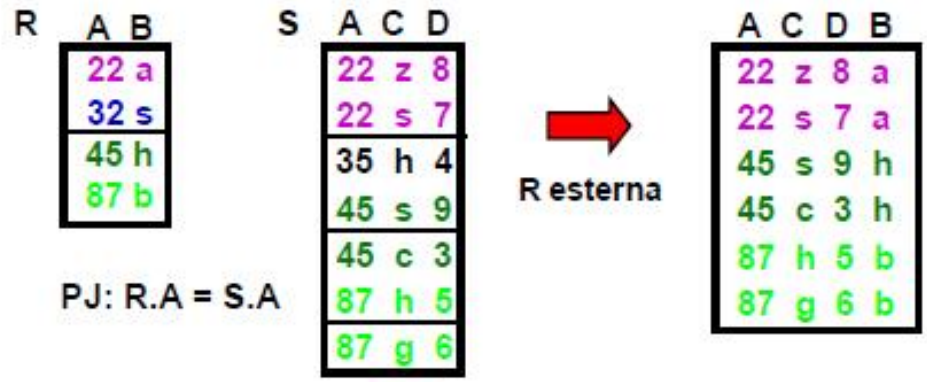

Figura 7: rappresentazione del funzionamento del Sort Merge Join

Questo tipo di join [13] sfrutta l'ordinamento delle tuple rispetto all'attributo di join per ridurre il numero di confronti. Le due tabelle R ed S devono, quindi, essere ordinate su tale attributo in modo tale da scorrerle parallelamente alla ricerca di tuple con lo stesso valore.

Se le due tabelle sono già ordinate, il vantaggio computazionale nell'utilizzo del Sort Merge rispetto al Nested Loop è evidente poiché le tabelle vengono scandite una sola volta. Il costo di esecuzione, allora, è pari a: $NP_R + NP_S$

Nel caso, invece, che le due tabelle non siano ordinate, è necessario considerare il trade-off dato dal costo di ordinamento delle due tabelle. Il costo di esecuzione, quindi, è: $Sort(R) + Sort(S) + NP_R + NP_S$

Hash Join

I metodi che fanno uso di tecniche hash [13][14] mirano ad ottenere lo stesso effetto del Sort Merge senza ricorrere all'ordinamento delle tabelle e all'uso di indici sugli attributi di join.

L'algoritmo di Simple-Hash Join è uno tra i più semplici della famiglia degli Hash Join.

Questo algoritmo è formato da due passi:

1. Build: si applica una funzione hash H ai valori degli attributi di join di una delle due tabelle (S, detta interna) generando una hash table H_s. Si utilizza un vettore V di bit per tener traccia di quali bucket di H_s sono vuoti o meno.
2. Probe: si scorre la seconda tabella (R, detta esterna) e, per ogni tupla, si accede a H_s tramite la stessa funzione hash H, solo se il bucket interessato non è vuoto.

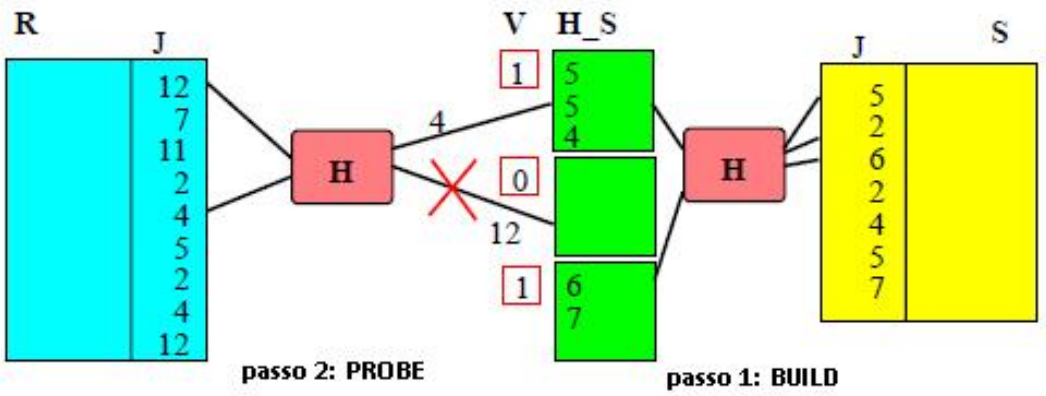

Figura 8: rappresentazione del funzionamento del Simple-Hash Join

Una valutazione di prima approssimazione del costo è: NPS + 2 NTS + NPR + NTR

dove NPS + 2 NTS è il costo del passo 1 (build), ossia lettura di S e costruzione di H_s, mentre NPR + NTR rappresenta il costo del passo 2 (probe), cioè lettura di R ed accesso a H_s.

Solitamente la tabella utilizzata per costruire l'hash table è quella con minor cardinalità (minor numero di tuple), mentre l'efficienza dell'algoritmo decresce all'aumentare delle collisioni, in quanto aumenta il numero di accessi a confronti inutili.

CAPITOLO II - RISULTATI SPERIMENTALI

In questa sezione si parlerà del software utilizzato per scrivere ed ottimizzare le query sql, del database su cui si fanno le interrogazioni sql e si vedranno tre esempi di query ottimizzate.

Ho svolto lo stage presso l'Istituto Zooprofilattico delle Venezie a Legnaro (IZSVe), che è un ente sanitario di diritto pubblico con autonomia gestionale ed amministrativa, facente parte del Servizio Sanitario Nazionale, quale strumento tecnico ed operativo per la sanità animale, il controllo della salute e qualità degli alimenti di origine animale, l'igiene degli allevamenti ed attività correlate. L'IZSVe, insieme agli altri nove IZS sparsi nel territorio italiano, è sottoposto alla vigilanza del Ministero della Salute.

Durante il periodo di stage, il mio compito principale è stato quello di scrivere ed eseguire query sql in modo da ottenere dei report per estrazioni dati richiesti dai vari uffici dell'Istituto.

Per scrivere ed eseguire le query ho utilizzato il programma IBExpert [24], che è un ambiente professionale di sviluppo integrato per la gestione di database InterBase e Firebird. IBExpert include molti strumenti, tra i quali:

- Editor visuali per tutti gli oggetti dei database
- Editor SQL e Script Executive
- Debugger per stored procedures e trigger

Il database che ho utilizzato durante il periodo di stage è un database Firebird.

Il DBMS Firebird (sviluppato e supportato dalla Firebird Foundation) [3][25] è open source ed è disponibile per Windows, Linux, Apple Macintosh OS/X, FreeBSD e altre piattaforme UNIX.

I database di Firebird sono composti da sequenze di pagine a dimensione fissa. Ogni pagina possiede un header che contiene un identificatore del tipo di pagina, un checksum e altre informazioni necessarie al DBMS per la corretta interpretazione.

A causa della dimensione fissa delle pagine, nascono delle limitazioni su vari aspetti:

- Numero massimo di tabelle: 32000
- Dimensione massima di una tabella: 32 TiB (tebibyte = 2^{40}, circa uguale al terabyte)
- Dimensione massima di un record: 64 KiB (kibibyte = 2^{10}, circa uguale al kilobyte)

Per quanto riguarda i tipi di dato memorizzabili, Firebird ammette i tipi standard (char, smallInt, integer, bigInt, float, double), i tipi data e ora (timestamp occupa 64 bit, date occupa 32 bit, time occupa 32 bit).

Firebird utilizza indici rappresentati come B^+Tree, supporta indici primary e secondary, indici multi-attributo, ascending o descending.

Il costo di accesso ad un indice clustered o unclustered è identico: durante l'accesso agli indici, è raccolta una bitmap di riferimenti ai record. Una volta terminata la ricerca, si accede alla pagina in un certo ordine evitando caricamenti ripetuti della stessa pagina.

Il Query Optimizer di Firebird è un ottimizzatore cost-based.

Per quanto riguarda gli algoritmi di join, Firebird considera la cardinalità e la selettività per decidere quale algoritmo usare tra (l'HASH JOIN non è stato ancora implementato):

- NESTED LOOP JOIN
- SORT MERGE JOIN

2.1 DATABASE IZSVe

Nell'immagine seguente è rappresentata una versione semplificata del database utilizzato all'IZSVe (nella versione completa sono presenti oltre 200 tabelle ed alcune di queste possono raggiungere più di 20 campi).

In questa versione semplificata sono rappresentati tutti i campi e tabelle usati nel prossimo paragrafo, dove saranno presentate tre query che sono state ottimizzate durante l'attività di stage.

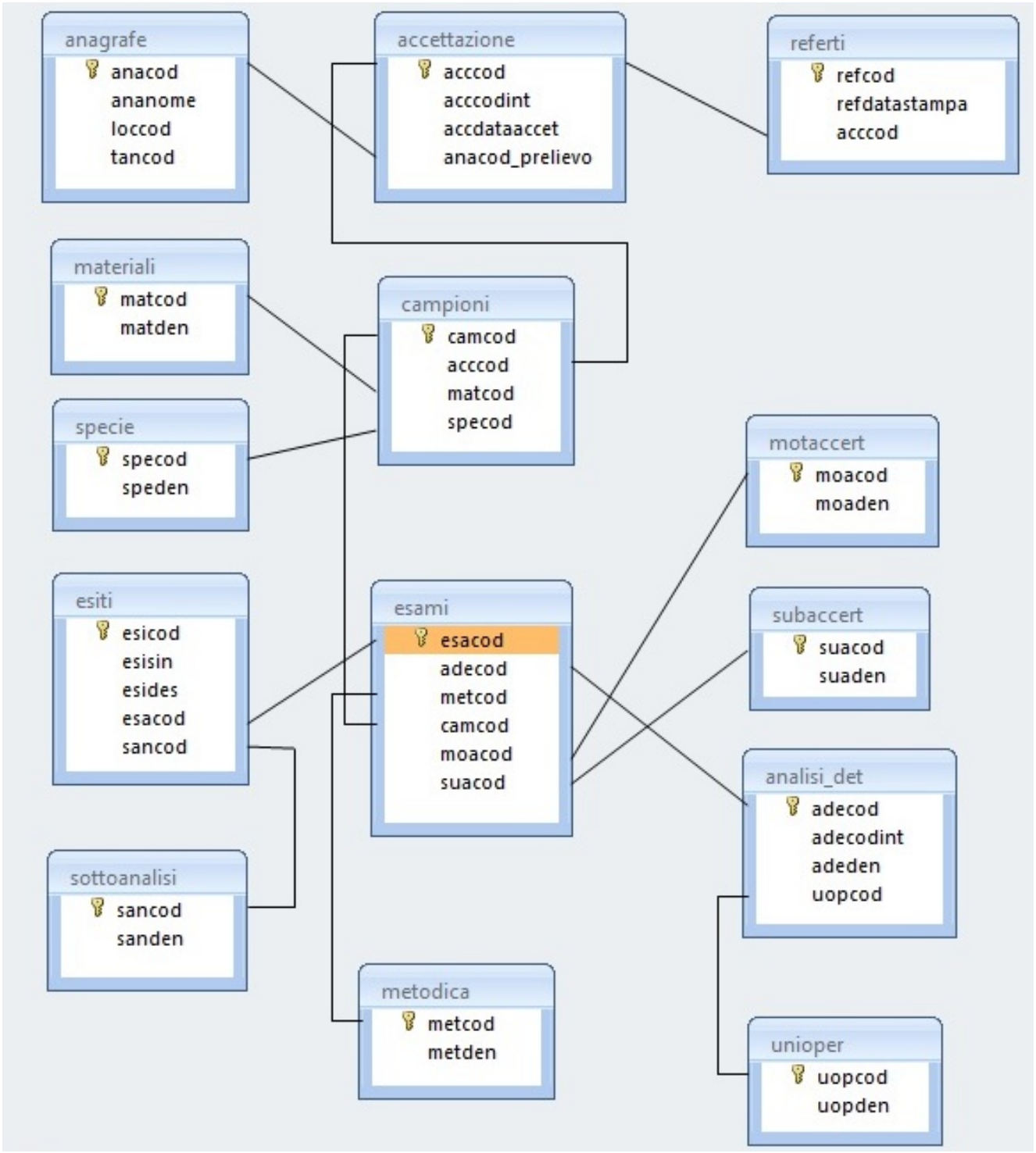

Figura 9: rappresentazione della versione semplificata del database utilizzato all'IZSVe

Tutte le chiavi primarie delle tabelle sotto rappresentate sono dei contatori, quindi di tipo integer.

Nella tabella ACCETTAZIONE, il campo ACCCODINT (varchar), è UNIQUE e rappresenta il numero di registro (anno + numero progressivo); il campo ACCDATAACCET (date) rappresenta la data di accettazione; il campo ANACOD_PRELIEVO (integer) è chiave esterna alla tabella ANAGRAFE e rappresenta il codice provenienza del luogo di prelievo del campione accettato.

Nella tabella ANAGRAFE, il campo ANANOME (varchar) rappresenta il nome o la ragione sociale del luogo di prelievo (allevamento, macello, stabilimento); il campo LOCCOD (varchar) rappresenta il codice della città del luogo di prelievo; il campo TANCOD (varchar) rappresenta la tipologia del luogo di prelievo (allevamento, macello, asl, ecc.).

Nella tabella REFERTI, REFDATASTAMPA (date) rappresenta la data di stampa del referto; ACCCOD (integer) è chiave esterna alla tabella ACCETTAZIONE.

Nella tabella CAMPIONI, i campi ACCCOD, MATCOD, SPECOD (tutti e tre di tipo integer) sono chiavi esterne alle tabella ACCETTAZIONE, MATERIALI, SPECIE.

Nella tabella MATERIALI è presente il campo MATDEN (varchar) che rappresenta la denominazione del tipo di materiale del campione (ad es. sangue, carcassa, organi interni, ecc.).

Nella tabella SPECIE, SPEDEN (varchar) rappresenta la denominazione della specie animale a cui fa riferimento il campione.

Nella tabella ESAMI, i campi ADECOD, METCOD, CAMCOD, MOACOD e SUACOD (tutti integer) sono chiavi esterne alle tabelle ANALISI_DET, METODICA, CAMPIONI, MOTACCERT e SUBACCERT.

Nella tabella MOTACCERT (che rappresenta una specifica ulteriore del motivo), MOADEN (varchar) rappresenta la denominazione del motivo di accertamento (ad es. ricerca, emergenza epidemica, controllo ufficiale, ecc.).

Nella tabella SUBACCERT (che rappresenta un ulteriore motivo per cui è stato eseguito un esame oltre al motivo accertamento), il campo SUADEN (varchar) è la denominazione del sub accertamento (ad es. piano di ricerca, sorveglianza, monitoraggio, ecc.).

Nella tabella METODICA, il campo METDEN (varchar) è la denominazione del metodo utilizzato per eseguire l'esame su un certo campione (ad es. cromatografia, antibiogramma, esame al microscopio, ecc.).
Nella tabella ESITI, il campo ESISIN (char) rappresenta una descrizione sintetica dell'esito (P = positivo, N = negativo, D = dubbio, I = inadatto, E = effettuato, L = nei limiti); il campo ESIDES (varchar) rappresenta, invece, la descrizione più dettagliata dell'esito; i campi ESACOD e SANCOD sono chiavi esterne alle tabelle ESAMI e SOTTOANALISI.
Nella tabella SOTTOANALISI, il campo SANDEN rappresenta la denominazione delle sottoanalisi per ogni esame (ad es. listeria, salmonella, ecc.).
Nella tabella ANALISI_DET, ADECODINT (varchar) rappresenta un'abbreviazione dell'analisi effettuata; ADEDEN (varchar) è la denominazione dell'analisi svolta; UOPCOD è chiave esterna alla tabella UNIOPER.
Nella tabella UNIOPER, il campo UOPDEN (varchar) rappresenta la descrizione dell'unità operativa che ha svolto le analisi (ad es. alimenti, sierologia, diagnostica, ecc.).

2.2 ESEMPIO QUERY 1:

```
SELECT anagrafe.ananome as Nome_azienda,
        accettazione.accdataaccet as Data_accettazione,
        anagrafe.loccod as ID_città,
        anagrafe.tancod as ID_tipologia_azienda,
        campioni.camcod as ID_campione,
        motaccert.moaden as Motivo_accertamento,
        subaccert.suaden as Subaccertamento,
        metodica.metden as Metodica,
        analisi_det.adecodint as ID_analisi_effettuata,
        esiti.esisin as Esito

FROM   accettazione INNER JOIN campioni on (accettazione.acccod=campioni.acccod)
        INNER JOIN anagrafe on (accettazione.anacod_prelievo=anagrafe.anacod)
        INNER JOIN esami on (campioni.camcod=esami.camcod)
        INNER JOIN motaccert on (esami.moacod=motaccert.moacod)
        INNER JOIN subaccert ON (esami.suacod=subaccert.suacod)
        INNER JOIN esiti on (esami.esacod=esiti.esacod)
        INNER JOIN materiali on (campioni.matcod=materiali.matcod)
        INNER JOIN analisi_det on (esami.adecod=analisi_det.adecod)
        INNER JOIN metodica on (esami.metcod=metodica.metcod)
        INNER JOIN referti on (accettazione.acccod=referti.acccod)

WHERE  accettazione.accdataaccet>='2012-10-01' AND accettazione.accdataaccet<='2012-12-31'
        AND referti.refdatastampa is NOT NULL
        AND anagrafe.tancod != 6                    --Allevamento
        AND analisi_det.adecodint LIKE 'M1%'        --analisi effettuata: Aflatossina%
        AND analisi_det.uopcod=33                   --PD chimica alimenti

GROUP BY anagrafe.ananome,
        accettazione.accdataaccet,
        anagrafe.loccod,
        anagrafe.tancod,
        campioni.camcod,
        motaccert.moaden,
        subaccert.suaden,
        motodica.metden,
        analisi_det.adecodint,
        esiti.esisin
```

In questa query sono state richieste alcune informazioni (campi estratti nel SELECT) riguardo accettazioni nel periodo ottobre 2010-dicembre 2012 in cui sono stati eseguiti degli esami su animali non appartenenti ad alcun allevamento (codice tipologia ente = 6) per verificare la presenza di Aflatossina nel latte (codice analisi effettuata = M1). Queste analisi sono state eseguite dal laboratorio di Padova - Chimica Alimenti (codice unità operativa = 33) il cui referto aveva una data di stampa (alcuni referti nel database non possiedono la data di stampa).
Dopo aver posto le condizioni della clausola WHERE nell'ordine migliore (ossia quello che riduce il più possibile la dimensione dei risultati intermedi) ed aver eseguito la query, IBExpert ha generato il seguente plan di esecuzione:

```
PLAN SORT (JOIN (ACCETTAZIONE INDEX (ACC_IX_DATAACCET),
        REFERTI INDEX (REF_FK_ACC),
        ANAGRAFE INDEX (ANA_PK_COD),
        CAMPIONI INDEX (CAM_FK_ACC),
        MATERIALI INDEX (PK_MATERIALI),
        ESAMI INDEX (ESA_FK_CAM),
        ESITI INDEX (ESI_FK_ESA),
        ANALISI_DET INDEX (ADE_PK_COD),
        MOTACCERT INDEX (MOA_PF_COD),
        METODICA INDEX (MET_PK_COD),
        SUBACCERT INDEX (SUA_PK_COD)))
```

Il tempo di esecuzione della query utilizzando il plan di IBExpert: 1m 14s 106ms

La seconda fase consiste nella riscrittura del plan di esecuzione della query, in cui ho anticipato tutte le tabelle contenute nella clausola WHERE, ossia:

```
PLAN SORT (JOIN (ACCETTAZIONE INDEX (ACC_IX_DATAACCET),
        REFERTI INDEX (REF_IX_ACCCOD),
        ANAGRAFE INDEX (ANA_PK_COD),
        CAMPIONI INDEX (CAM_FK_ACC),
        ESAMI INDEX (ESA_FK_CAM),
        ANALISI_DET INDEX (ADE_PK_COD)
        MATERIALI INDEX (PK_MATERIALI),
        MOTACCERT INDEX (MOA_PF_COD),
        SUBACCERT INDEX (SUA_PK_COD),
        METODICA INDEX (MET_PK_COD),
        ESITI INDEX (ESI_FK_ESA)))
```

Tempo di esecuzione della query utilizzando il plan riscritto: 0m 05s 072ms

Ottimizzando la query riscrivendo il plan di esecuzione, il tempo di risposta della query si è ridotto di circa il 93%.

2.3 ESEMPIO QUERY 2:

```
SELECT substring (accettazione.accdataaccet from 6 for 2) as Mese_Accettazione,
        unioper.uopden as Unità_Operativa_esecutrice,
        analisi_det.adeden as Analisi_effettuata,
        count (esiti.esicod) as Numero_esami

FROM  accettazione INNER JOIN campioni on(campioni.acccod=accettazione.acccod)
        INNER JOIN esami on(campioni.camcod=esami.camcod)
        INNER JOIN esiti on (esami.esacod=esiti.esacod)
        INNER JOIN analisi_det on (esami.adecod=analisi_det.adecod)
        INNER JOIN unioper on (analisi_det.uopcod=unioper.uopcod)

WHERE accettazione.accdataaccet>='01.01.2012' and accettazione.accdataaccet<='31.12.2012'
        AND  unioper.uopcod=12                  --PD Microbiologia Alimentare
        AND (analisi_det.adeden like '%SALMONELLA%'
              OR  analisi_det.adeden like '%LISTERIA MONO%')

GROUP BY  substring (accettazione.accdataaccet from 6 for 2),
        unioper.uopden,
        analisi_det.adeden
```

In questa query, l'obiettivo era quello di estrarre informazioni (campi SELECT) riguardo accettazioni dell'anno 2012 in cui sono state svolti degli esami dal laboratorio di Padova – Microbiologia Alimentare (codice unità operativa = 12) per verificare la presenza di batteri di Salmonella o Listeria Monocytogenes.
Dopo aver posto le condizioni della clausola WHERE nell'ordine migliore (ossia quello che riduce il più possibile la dimensione dei risultati intermedi), si è eseguita la query.
Il plan fornito da IBExpert è il seguente:

```
PLAN SORT (JOIN (ACCETTAZIONE INDEX (ACC_IX_DATAACCET),
        CAMPIONI INDEX (CAM_FK_ACC),
        ESAMI INDEX (ESA_FK_CAM),
        ESITI INDEX (ESI_FK_ESA),
        ANALISI_DET INDEX (ADE_PK_COD),
        UNIOPER INDEX (UOP_PK_COD)))
```

Tempo di esecuzione della query utilizzando il plan di IBExpert: 9m 25s 831ms

Si è proceduto riordinando il piano di accesso alle tabelle mediante indici, e si è ottenuto:

```
PLAN SORT (JOIN (ACCETTAZIONE INDEX (ACC_IX_DATAACCET),
        UNIOPER INDEX (UOP_PK_COD),
        CAMPIONI INDEX (CAM_FK_ACC),
        ESAMI INDEX (ESA_FK_CAM),
        ESITI INDEX (ESI_FK_ESA),
        ANALISI_DET INDEX (ADE_PK_COD)))
```

Tempo di esecuzione della query utilizzando il plan riscritto: 0m 52s 698ms

In questo caso, il tempo di esecuzione si è ridotto di circa il 90%.

2.4 ESEMPIO QUERY 3:

```
SELECT extract (year from accettazione.accdataaccet) as Anno,
        accettazione.acccodint as Codice_Unico_accettazione,
        motaccert.moaden as Motivo_accertamento,
        subaccert.suaden as Subaccertamento,
        unioper.uopden as Unità_Operativa_esecutrice,
        specie.speden as Specie,
        materiali.matden as Matrice,
        metodica.metden as Metodo,
        analisi_det.adeden as Tipo_Analisi,
        sottoanalisi.sanden as Sottoanalisi,
        campioni.camcod as Codice_campione,
        esiti.esicod as Codice_esito,
        esiti.esisin as Esito,
        esiti.esides as Valore

FROM accettazione INNER JOIN campioni on (accettazione.acccod=campioni.acccod)
            INNER JOIN esami on (esami.camcod=campioni.camcod)
            INNER JOIN esiti on (esiti.esacod=esami.esacod)
            INNER JOIN sottoanalisi on (sottoanalisi.sancod=esiti.sancod)
            INNER JOIN anagrafe on (accettazione.anacod_prelievo=anagrafe.anacod)
            INNER JOIN analisi_det on (analisi_det.adecod=esami.adecod)
            INNER JOIN unioper on (analisi_det.uopcod=unioper.uopcod)
            INNER JOIN motaccert on (motaccert.moacod=esami.moacod)
            INNER JOIN subaccert on (subaccert.suacod=esami.suacod)
            INNER JOIN specie on (specie.specod=campioni.specod)
            INNER JOIN materiali on (materiali.matcod=campioni.matcod)
            INNER JOIN metodica on (metodica.metcod=esami.metcod)

WHERE accettazione.accdataaccet>='2011-01-01' and accettazione.accdataaccet<='2011-12-31'
        AND (specie.specod=624 OR specie.specod=613)    -- specie:  Cane e Bovino
        AND analisi_det.adeden like 'LEPTO%'            -- analisi effettuata: Leptospira
        AND metodica.metden like 'MICROAGGL%'           -- tecnica di prova: Microagglutinazione
        AND motaccert.moaden NOT LIKE '%RICERCA%'
```

```
GROUP BY  extract (year from accettazione.accdataaccet),
        accettazione.acccodint,
        motaccert.moaden,
        subaccert.suaden,
        unioper.uopden,
        specie.speden,
        materiali.matden,
        metodica.metden,
        analisi_det.adeden,
        sottoanalisi.sanden,
        campioni.camcod,
        esiti.esicod,
        esiti.esisin,
        esiti.esides
```

In quest'ultima query erano richiesti i campi nel SELECT di accettazioni dell'anno 2011 in cui sono stati fatti degli esami per verificare la presenza di leptospira in campioni di cane (codice specie = 624) o bovino (codice specie = 613) mediante l'uso della tecnica di microagglutinazione (esami non eseguiti per la ricerca).
Per ottenere questa serie di informazioni, IBExpert ha fornito il seguente plan:

```
PLAN SORT (JOIN (ACCETTAZIONE INDEX (ACC_IX_DATAACCET),
        ANAGRAFE INDEX (ANA_PK_COD),
        CAMPIONI INDEX (CAM_FK_ACC),
        SPECIE INDEX (PK_SPECIE),
        MATERIALI INDEX (PK_MATERIALI),
        ESAMI INDEX (ESA_FK_CAM),
        ESITI INDEX (ESI_FK_ESA),
        ANALISI_DET INDEX (ADE_PK_COD),
        SOTTOANALISI INDEX (SAN_PK_COD),
        UNIOPER INDEX (UOP_PK_COD),
        METODICA INDEX (MET_PK_COD),
        MOTACCERT INDEX (MOA_PF_COD),
        SUBACCERT INDEX (SUA_PK_COD)))
```

Tempo di esecuzione della query utilizzando il plan di IBExpert: 5m 32s 704ms

Riordinando il plan di accesso alle tabelle in base alla clausola WHERE, si è ottenuto:

```
PLAN SORT (JOIN (ACCETTAZIONE INDEX (ACC_IX_DATAACCET),
        SPECIE INDEX (PK_SPECIE),
        CAMPIONI INDEX (CAM_FK_ACC),
        ESAMI INDEX (ESA_FK_CAM),
        ANALISI_DET INDEX (ADE_PK_COD),
        METODICA INDEX (MET_PK_COD),
        MOTACCERT INDEX (MOA_PF_COD),
        SUBACCERT INDEX (SUA_PK_COD),
        ANAGRAFE INDEX (ANA_PK_COD),
        ESITI INDEX (ESI_FK_ESA),
        MATERIALI INDEX (PK_MATERIALI),
        SOTTOANALISI INDEX (SAN_PK_COD),
        UNIOPER INDEX (UOP_PK_COD)))
```

Tempo di esecuzione della query utilizzando il plan riscritto: 0m 15s 350ms

Riscrivendo il plan di esecuzione della query, il tempo si è ridotto di circa il 95% rispetto al tempo di esecuzione iniziale utilizzando il plan fornito da IBExpert.

2.5 Considerazioni finali

Nella tabella seguente è riportato un quadro riassuntivo delle tre query sopra ottimizzate riscrivendo il plan di esecuzione.

Query	Tempo di esecuzione prima	Tempo di esecuzione dopo	Miglioramento percentuale
Query 1	1 m 14 s 106 ms	0 m 05 s 072 ms	**93,25%**
Query 2	9 m 25 s 831 ms	0 m 52 s 698 ms	**90,70%**
Query 3	5 m 32 s 704 ms	0 m 15 s 350 ms	**95,40%**

Nella prima e nella seconda colonna sono rappresentati i tempi di esecuzione delle query prima e dopo che queste vengano ottimizzate.
Nella terza colonna vengono rappresentati, in percentuale, di quanto sono migliorati i tempi di esecuzione delle query dopo averle ottimizzate. Per calcolare queste percentuali si utilizza la seguente formula:

*100 – ("Tempo di esecuzione dopo" * 100 / "Tempo di esecuzione prima")*

Si nota, quindi, che riscrivendo il plan di esecuzione delle tre query SQL esaminate si ottiene un miglioramento percentuale che varia dal 90% al 95%, abbattendo di fatto i tempi di esecuzione.

CAPITOLO III - CONCLUSIONI

Si è visto che quando si scrive una query SQL bisogna tener conto di alcune semplici "regole" per velocizzare il più possibile il tempo di risposta, quali: nella clausola WHERE, inserire le condizioni nell'ordine migliore, ossia trovare l'ordine che riduce il più possibile la dimensione dei risultati intermedi; evitare di usare, se non in casi in cui non se ne può fare a meno, subquery.

Quando si esegue la query, il Query Optimizer analizza alcuni dei possibili piani di esecuzione dell'interrogazione e fornisce il piano che considera migliore. La maggior parte delle volte, però, questo piano di esecuzione è imperfetto e tocca agli utenti ed agli amministratori del database riscriverlo per migliorarlo e velocizzare, quindi, il tempo di risposta della query SQL.

Abbiamo visto anche che ci sono più tecniche per ottimizzare le query: l'ottimizzazione algebrica, che di solito è la prima che viene effettuata; l'ottimizzazione sintattica e semantica; l'ottimizzazione euristica e l'ottimizzazione basata sui costi.

Per velocizzare il tempo di risposta delle query, bisogna anche tener conto degli indici e dei join. Per quanto riguarda gli indici, le strutture più utilizzate sono quelle B^+tree e i Bitmap. Per i join esistono principalmente tre algoritmi: Nested Loop, Sort Merge e Simple-Hash.

Nelle query ottimizzate durante l'attività di stage all'IZSVe, si è notato che il Query Optimizer genera piani di esecuzione che non sono molto efficienti. Riscrivendo questi plan, si sono ottenuti dei miglioramenti, dal punto di vista del tempo, anche del 95%, ossia si è riusciti a passare da alcuni minuti di esecuzione della query a pochi secondi prima di ottenere i risultati richiesti.

L'ottimizzazione, quindi, è parte fondamentale quando si fanno delle interrogazioni ai database in quanto permette di risparmiare notevoli quantità di tempo e risorse.

BIBLIOGRAFIA

[1] Atzeni, Ceri, Fraternali, Paraboschi, Torlone, Basi di Dati Vol.2

[2] Bayer, Rudolf; McCreight, E. (July 1970), Organization and Maintenance of Large Ordered Indices, Mathematical and Information Sciences Report No. 20, Boeing Scientific Research Laboratories.

[3] Bellavista, Mella, Firebird DBMS, Università di Bologna, 2011

[4] Bernstein PA & Newcomer E (1997) *Principles of Transaction Processing For the System Professional*, Morgan Kaufmann, San Mateo CA

[5] C. Favre, F. Bentayeb, Bitmap Index-based Decision Trees, 15th International Symposium on Methodologies for Intelligent Systems (ISMIS 05), New York, USA, May 2005 ; LNAI, Vol. 3488, 65-73

[6] Chaudhuri, Surajit (1998). "An Overview of Query Optimization in Relational Systems". Proceedings of the ACM Symposium on Principles of Database Systems. pp. pages 34–43.

[7] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, Sumin Song, RankSQL: Query Algebra and Optimization for Relational Top-k Queries, In *Proceedings of the 2005 ACM SIGMOD Conference (SIGMOD)*, pages 131-142, Baltimore, Maryland, USA, June 2005

[8] Codd E.F., Codd S.B., and Salley C.T. (1993). "Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate". Codd & Date, Inc. Retrieved 2008-03-05

[9] Elmasri, Navathe, Sistemi di basi di dati - Fondamenti, Pearson - Addison-Wesley, 2007

[10] Garcia Molina, Ullman, Widom, Database Systems – The complete book, Prentice Hall, 2002

[11] Ioannidis, Yannis (March 1996). "Query optimization". ACM Computing Surveys 28 (1): 121–123

[12] J.J.King. Quist: A system for semantic query optimization in relational databases. In Proc, of the 7th Int. VLDB Conf., pages 510-1517, Cannes France, August 1981

[13] Kim, Kaldewey, Lee, Sedlar, Nguyen, Satish, Chhugani, Di Blas, Dubey (2009). Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs

[14] Kitsuregawa, M.; Tanaka, H.; Moto-Oka, T. (1983). "Application of Hash to Data Base Machine and Its Architecture". New Generation Computing 1 (1): 63–74

[15] M.Jarke, J.Koch. Query Optimization in Database Systems. ACM Computing Surveys, 16(2), June 1984

[16] "OLAP Council White Paper" (PDF). OLAP Council. 1997. Retrieved 2008-03-18

[17] S. Sarawagi and M. Stonebraker. Single query optimization in tertiary memory. Technical Report Sequoia 2000, S2k-94-45, University of California at Berkeley, 1994

[18] Sunil Choenni, Henk M. Blanken, Thiel Chang, Index Selection in Relational Databases, ICCI, 1993

[19] Surajit Chaudhuri, Kyuseok Shim: An Overview of Cost-based Optimization of Queries with Aggregates. Data Engineering Bulletin 18(3): 3-9 (1995)

[20] V. Poosala, Y.Ioannidis, P.Haas and E.Shekita. Improved histograms for selectivity estimations of range predicates. In ACM SIGMOD Conf. On the Management of Data, 1996

[21] Zhiyuan Chen, Johannes Gehrke, and Flip Korn, Query Optimization in Compressed Database Systems. ACM SIGMOD International Conference on Management of Data, 2001: p. 271-282

[22] http://en.wikipedia.org/wiki/Database_index

[23] http://en.wikipedia.org/wiki/Query_optimizer

[24] http://ibexpert.net

[25] http://www.firebirdsql.org