# Ottimizzazione della Memoria per Query non Ricorsive

Giuseppe Di Battista e Alberto Marchetti-Spaccamela

Dipartimento di Informatica e Sistemistica
Universita' di Roma "La Sapienza"
Via Buonarroti, 12 - 00185 Roma

**Sommario**

Nelle basi di dati tradizionali e' piuttosto raro che si debbano eseguire query con un alto numero di operazioni di join, al contrario, nelle basi di dati deduttive, e' frequente il caso di dover eseguire query che coinvolgono centinaia di join. In questo lavoro si considera il problema della ottimizzazione di sequenze di join, concentrando la attenzione sulla occupazione di memoria. La complessita' computazionale del problema viene caratterizzata per varie classi di query ed inoltre si indaga sulle relazioni intercorrenti tra tempo di risposta e impegno di memoria.

## 1. Introduzione

Nelle basi di dati deduttive e' frequente il problema di dover effettuare query che, tradotte in termini di espressioni dell'algebra relazionale, coinvolgono centinaia o migliaia di join [KBZ]. Per questa ragione un problema cruciale per la realizzazione di basi di dati deduttive e' la disponibilita' di algoritmi di ottimizzazione che minimizzino i costi di calcolo delle risposte.

Ci sono molti possibili modi di eseguire una query, essenzialmente dipendenti dall'ordine di esecuzione delle operazioni e dalla scelta dei metodi di implementazione della singola operazione. Lo scopo di questo lavoro e' quello di individuare una strategia e degli algoritmi per minimizzare i costi di calcolo della risposta.

Il problema e' stato gia' esaminato da vari autori (si vedano ad esempio [S], [W],

[IK], [KZ] e [KBZ]); in particolare in [KBZ] si descrivono un modello ed una precisa formulazione del problema di ottimizzazione, con lo scopo di evidenziare i vari parametri che debbono influenzare la scelta di una strategia ottimale. Comunque in tutti i lavori elencati il problema e' stato affrontato dal punto di vista della ottimizzazione del tempo di risposta, senza considerare l'analogo problema di ottimizzazione della memoria; in realta' in tutti i casi si fa l'ipotesi che non ci siano limiti alla memoria centrale utilizzabile e quindi l'ipotesi che tutte le relazioni temporanee utilizzate durante il calcolo possano essere contemporaneamente memorizzate.

Si osservi che non e' sempre vero che la memoria centrale e' sufficiente a memorizzare tutte le relazioni di partenza ed i risultati intermedi; in queto senso quello che occorrerebbe progettare e' un algoritmo che ottimizzi il tempo di risposta, con il vincolo di un valore massimo per la memoria a disposizione. La complessita' di questo problema e' notevole, per questa ragione nel seguito si assumeranno alcune ipotesi semplificatrici. In particolare concentreremo la nostra attenzione sul problema della occupazione di memoria ed ottimizzando la memoria ci aspettiamo di ottenere una buona soluzione approssimata anche dal punto di vista del tempo di risposta.

Nella sezione 2 si danno le definizioni di base e si introduce la terminologia utilizzata. Il problema di ottimizzazione e' definito come problema su un grafo (grafo di join in [KBZ])) e si mostra come il problema di ottimizzare la memoria sia in generale NP-completo. Nella sezione 3 si esamina il caso in cui il grafo di join sia aciclico. In particolare si esamina un problema lasciato aperto in [KBZ]: quello di utilizzare piu' di un risultato temporaneo contemporaneamente in memoria principale, in modo da considerare un maggior numero di strategie di esecuzione (usando la terminologia usata in [KBZ] l'intero spazio di processamento) e di rendere possibile un approccio parallelo. Il problema e' risolto completamente per una certa classe di grafi di join, inoltre si propone una euristica semplice ed efficiente per ottenere una stima pessimistica dei requisiti di memoria di un di un qualunque grafo di join aciclico. Nella sezione 4 si propone una soluzione euristica per il caso di un grafo di join qualunque. Una trattazione teorica completa esula dagli scopi del presente lavoro, quindi le prove di alcuni teoremi vengono omesse.



## 2. Definizioni

Data una query Q, siano $R_1,..,R_n$, le relazioni coinvolte. Supponiamo che Q abbia la forma seguente:

$$Q = \pi_{TL} [ \sigma_{QL} (R_1 \times .... \times R_n) ],$$

dove $\sigma,\pi$, e x indicano rispettivamente selezione, proiezione e prodotto cartesiano. QL e' la cogiunzione di predicati di join e di selezione, e TL e' la lista degli attributi target. Noi restringeremo la attenzione a query che sono formate esclusivamente da join: questa ipotesi e' giustificata dal fatto che comunque le operazioni di join sono le piu' costose (si veda anche [S]). In corrispondenza a Q si definisce il grafo di join $G_Q(V,E)$, che rappresenta la query: le relazioni sono in corrispondenza con i nodi del grafo ($V=\{R_1...R_n\}$), mentre ogni arco rappresenta un join ( $<R_i, R_j>$ appartiene a E se c'e' qualche predicato del tipo $R_i.a = R_j.b$) .

La <u>dimensione</u> (in bytes) di una relazione $R_i$ sara' indicata con $|R_i|$. Il <u>fattore di selettivita'</u> $s_{ij}$, rispetto al join tra $R_i$ e $R_j$ e' definito in modo tale che:

$$|R_i|\ |R_j|\ s_{ij}$$

sia il valore atteso della dimensione del risultato.

Si fanno le usuali ipotesi [S] riguardanti la uniformita' di distribuzione dei valori degli attributi e la loro mutua indipendenza. Ne segue che la dimensione del risultato di due join (ad esempio di $R_1$ con $R_2$ e di $R_2$ con $R_3$) e'

$$|R_1|\ |R_2|\ |R_3|\ s_{12}\ s_{23}.$$

Si noti come questa sia chiaramente una approssimazione, comunque questo approccio e' largamente accettato sia per la sua semplicita' che per il suo buon comportamento nella pratica.

Come e' stato osservato nell'introduzione, ci sono vari modi per eseguire una query, in dipendenza dall'ordine di esecuzione delle interrogazioni e dalla scelta dei metodi per eseguirle. Un *piano di esecuzione* per calcolare la risposta a una query e' la specifica di:

1. L'algoritmo utilizzato per implementare ogni operazione;
2. Le decisioni che concernono le relazioni che devono essere materializzate ed eventualmente spostate in memoria secondaria per far posto in memoria principale

3. L'ordine di esecuzione delle operazioni.

L'obiettivo e' quello di determinare un piano di esecuzione che minimizzi il costo di calcolare la risposta.

## 2.1 Scelta di un metodo per calcolare i join

E' stato osservato in [D] che, per basi di dati residenti in memoria, il metodo di join piu' conveniente e' l'hash join. Nel seguito si fara' riferimento in prevalenza a questo metodo; comunque tutte le considerazioni ed i risultati restano validi anche per altri metodi di join.

Rimandiamo a [D] per una descrizione dell'hash join, ai nostri fini e' solo necessario ricordare che, se $R_i$ e' la relazione hash (residente in memoria centrale) e $R_j$ e' in memoria secondaria allora l'hash join richiede memoria pari a:

$$c\,|R_i| + s_{ij}\,|R_i|\,|R_j| + t$$

dove c e' una costante maggiore di 1 che puo' essere scelta in modo appropriato in modo da minimizzare il costo dell'operazione e t e' la dimensione del template utilizzato per portare in memoria porzioni di $R_j$.

Se anche $R_j$ e' in memoria principale, perche' e' il risultato di una relazione temporanea calcolata precedentemente, la memoria necessaria e':

$$c\,|R_i| + s_{ij}\,|R_i|\,|R_j| + |R_j|$$

Dato che i valori di c e t non sono importanti per cio' che segue essi verranno d'ora in poi omessi.

## 2.2 Scelta delle materializzazioni

Il numero e la dimensione delle relazioni materializzate che sono contemporaneamente in memoria centrale dipende dal piano di esecuzione scelto.

In [S] [IK] [KZ] [KBZ] si considera solo il caso in cui in memoria centrale sia presente una sola relazione materializzata. L'esempio seguente mostra come, in alcuni casi, il piano di esecuzione ottimo richieda la presenza contemporanea in memoria di piu' di una materializzazione. Sia Q la query:

$$Q = \pi_{TL}\,[\,s_{QL}\,(R_1 \times R_2 \times R_3 \times R_4)\,]$$

dove QL e' del tipo



$$R_1.a = R_2.b \text{ and } R_2.c = R_3.d \text{ and } R_3.e = R_4.f$$

Il grafo di join corrispondente e' mostrato in figura 1. Le etichette sugli archi rappresentano i fattori di selettivita'; le etichette sui vertici rappresentano le dimensioni delle corrispondenti relazioni.

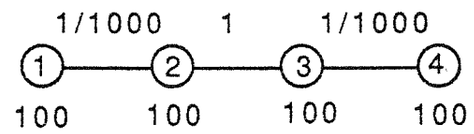


Fig. 1 Un esempio di grafo di join.

Se e' possibile materializzare una sola relazione, allora il migliore piano di esecuzione esegue i join nell'ordine seguente:

- per prima cosa $R_1$ viene trasferita in memoria principale e viene eseguito il join con $R_2$;
- poi si effettua il join tra la relazione temporanea ottenuta ed $R_3$;
- l'ultimo join e' tra la relazione temporanea ed $R_4$.

La memoria necessaria e' 1010.

Se scegliamo invece il piano di esecuzione seguente:

- prima $R_1$ e' trasferita in memoria centrale e si esegue il join con $R_2$;
- poi $R_3$ e' trasferita in memoria principale e si esegue il join con $R_4$;
- alla fine si fa il join tra le due relazioni temporanee;

e' facile verificare come la memoria necessaria sia 120. Considerazioni analoghe possono essere fatte anche per il tempo di esecuzione.

L' idea di base e' che l'ottimizzatore non dovrebbe occuparsi del fatto di dover mettere in memoria secondaria le relazioni temporanee per fare spazio in memoria principale. In tal senso si suggerisce il seguente approccio lazy: l'ottimizzatore tenta di

tenere al valore minimo possibile l'occupazione di memoria centrale e memorizza tutte le relazioni temporanee in memoria centrale fino a quando cio' e' possibile. Quando la memoria centrale diviene insufficiente per continuare il piano di esecuzione, l'ottimizzatore sceglie in modo euristico la relazione (o le relazioni) da portare temporaneamente in memoria secondaria. Dato che l'occupazione di memoria centrale viene tenuta il piu' bassa possibile, ci si attende un limitato numero di trasferimenti.

### 2.3 Scelta della sequenza di relazioni

Come conseguenza delle considerazioni dei punti 2.1 e 2.2 il nostro scopo ora e' quello di trovare, tra tutte le possibili permutazioni delle operazioni, l'ordine di esecuzione che minimizzi la memoria massima richiesta per rispondere alla query.

In questo senso la prima cosa da osservare e' il fatto che il problema di minimizzare la memoria necessaria e' in generale NP-completo.

**Teorema** Data una query Q il problema di trovare un piano di esecuzione che minimizzi la memoria necessaria a risolvere Q e' NP-completo.

## 3. Il caso di grafi di join aciclici

In questa sezione ci occupiamo del caso in cui il grafo di join sia un albero. In particolare nel paragrafo 3.1 si considera il problema della ottimizzazione della memoria nel caso in cui una sola relazione sia mantenuta in memoria principale. In questo caso si propone un algoritmo che trova il piano di esecuzione ottimo in tempo $O(n^2)$. Nei paragrafi 3.2 e 3.3 ci si occupa del problema di mantenere in memoria pu' di una relazione: in 3.2 il problema viene risolto per una particolare classe di alberi di join, in 3.3 si presenta una semplice euristica per valutare la ocupazione di memoria per un qualunque albero di join. Nel paragrafo 3.4 si mostrano alcune relazioni tra tempo di esecuzione e memoria richiesta da una query.



### 3.1 Il caso di una sola relazione temporanea

In questo paragrafo si fa l'ipotesi di mantenere in memoria, ad ogni istante, una sola relazione: il risultato parziale del processamento dei join. Gli altri join vengono effettuati ad uno ad uno tra le altre relazioni ed il risultato parziale (con riferimento alla terminologia usata in [KBZ] si esaminano Binary Linear Processing Trees). In accordo con questa strategia l'albero di join pu' essere pensato come un albero orientato, la cui radice e' la prima relazione della sequenza di join. Cio' semplifica la notazione per i fattori di selettivita': nel seguito verranno rappresentati con un solo pedice, in modo tale che $s_i$ rappresenti il fattore di selettivita' nel join tra $R_i$ e suo padre nell'albero oriemntato di join.

La memoria necessaria per calcolare il join tra due relazioni $R_1$ ed $R_2$, con radice $R_1$ e' espressa da $|R_1| + s_2|R_1||R_2|$, infatti, prima $R_1$ e' trasferita in memoria centrale, poi si calcola il join con il metodo hash, creando una tupla per volta la nuova relazione, la cui dimensione e' chiaramente $s_2|R_1||R_2|$.

Per una certa sequenza S di join, la memoria necessaria e':

$$C(S) = \max [ (|R_1|s_1+|R_1||R_2|s_1s_2), ( |R_1||R_2|s_1s_2+|R_1||R_2||R_3|s_1s_2s_3), \ldots, (|R_1|\ldots|R_{n-1}|s_1\ldots s_{n-1}+|R_1|\ldots|R_n|s_1\ldots s_n) ] =$$

$$= \max_{k=1\ldots n-1} \left( \prod_{i=1}^{k} |R_i|s_i + \prod_{i=1}^{k+1} |R_i|s_i \right) = \max_{k=1\ldots n-1} \left[ \left(\prod_{i=1}^{k} |R_i|s_i\right) ( 1 + |R_{k+1}|s_{k+1}) \right],$$

dove $s_1=1$ (fattore di selettivita' dummy) ed n e' il numero di relazioni coinvolte nella query.

La funzione di costo puo' essere riscritta nel modo seguente:

$C(R_i) = s_i|R_i|$

$C(S_1S_2) = \max [ C(S_1), (1 + first(S_2))T(S_1), C(S_2)T(S_1) ]$

per ogni coppia di sottosequenze non nulle, dove:

$$T(S) = \prod_{i=1}^{n} s_i|R_i|$$

e first(S) e' $s_i|R_i|$ se il primo elemento della sequenza S e' la relazione $R_i$.

Ad esempio possiamo calcolare:

$$C(R_iR_j) = \max ( s_i|R_i|, (1+ s_j|R_j|)s_i|R_i|, s_i|R_i|s_j|R_j| ) = s_i|R_i| + s_is_j|R_i||R_j|.$$

E' facile verificare che le due definizioni che abbiamo dato per C(S) sono equivalenti. Usando la seconda definizione si puo' provare il seguente teorema:

**Teorema** La funzione di costo C(S) ammette la *string interchange relation* transitiva e completa.

La presenza della string interchange relation [M] permette di confrontare i due costi C(AUVB) e C(AVUB) indipendentemente dalle sottosequenze A e B. Ne segue che possiamo decidere l'ordine delle sottosequenze U e V in modo indipendente dal resto della sequenza. Questa proprieta' permette di calcolare la memoria necessaria (per una sequenza di join che parte da una certa radice) in tempo O(n log(n)), usando la tecnica mostrata in [M].

Per poter calcolare la minima memoria richiesta e' a questo punto sufficiente iterare la stessa procedura per tutte le possibili radici, e questo pu' essere facilmente fatto in $O(n^2 \log(n))$. Il tempo di calcolo puo' essere migliorato usando i risultati parziali del calcolo come in [KBZ]. Usando questa tecnica il calcolo puo' essere effettuato in $O(n^2)$.

### 3.2 Alberi join increasing

Un problema lasciato aperto in [KBZ] e' la possibilita' di gestire piu' di un risultato temporaneo in memoria, in modo da coprire una classe piu' ampia di piani di esecuzione (usando la terminologia di [KBZ]: l'intero Processing Tree space) ed in modo da favorire un approccio parallelo al calcolo. Noi risolviamo il problema per una certa



classe di alberi di join per il nostro problema di ottimizzazione della memoria.

**Definizione.** Un join tra due relazioni $R_i$ e $R_j$ si dice essere <u>join increasing</u> se:

$$|R_i||R_j|s_{ij} \geq |R_i|+|R_j|,$$

altrimenti e' detto essere <u>join decreasing</u>. Un albero di join JT si dice essere join increasing se tutti i join di JT sono join increasing. In modo analogo si definisce un albero join decreasing.

Si noti che se un albero di join e' join increasing e si sostituisce l'arco $<R_i,R_j>$ ed i vertici $R_i$, $R_j$ con un vertice che rappresenta il loro join, l'albero di join risultante e' ancora join increasing.

Dal nostro punto di vista possiamo distinguere due tipi di join:

(1) join tra due relazioni temporanee;
(2) join tra una relazione temporanea ed una nuova relazione.

Per join di tipo 1 la memoria necessaria e' $|R_i||R_j|s_{ij}+|R_i|+|R_j|$. Per join di tipo 2 la memoria necessaria e' $|R_i||R_j|s_{ij}+|R_i|$, dove $R_i$ e' risultato parziale. A questo punto e' possibile enunciare la seguente proprieta':

**Teorema.** Per ogni piano di esecuzione di un dato albero join increasing l'ultimo join ha la massima necessita' di memoria.

In base a questa proprieta', per determinare il piano di esecuzione che richiede meno memoria e' sufficiente determinare quale join deve essere calcolato per ultimo. In altri termini occorre determinare qual'e' il join che, eseguito per ultimo, richiede meno memoria. Distinguiamo due casi: se una delle due relazioni coinvolte nell'ultimo join e' una foglia, allora il join deve essere calcolato come join di tipo 1, altrimenti e' meglio calcolarlo come join di tipo 2.

La scelta dell'ultimo join in un albero JT pu' essere facilmente compiuta in tempo $O(n^2)$. Il seguente algoritmo migliora questo limite utilizzando una visita in post-ordine di JT.

Algoritmo A

1. calcola P: prodotto di tutti i fattori di selettivita' e di tutte le dimensioni di JT
2. scegli una radice r per JT, determinando in questo modo una orientazione per JT (si noti che r puo' essere un qualunque nodo di JT che non sia una foglia)
3. visita JT in post-ordine, calcolando per ogni vertice $R_i$ il prodotto $P_i$ di tutti i fattori di selettivita' e delle dimensioni del sottoalbero che parte da $R_i$
4. min = maxvalue
5. per ogni arco diretto $<R_i, R_j>$ di JT
   - se $R_j$ non e' una foglia
     allora $v = P/(P_j\, s_{ij}) + P_j + P$
     altrimenti $v = P/(P_j\, s_{ij}) + P$
   - se v<min allora min=v e=$<R_i, R_j>$

**Teorema** L'Algoritmo A trova un piano di esecuzione ottimo in tempo lineare per un albero join increasing.

3.3 Stima euristica della occupazione di memoria

Si propone qui una euristica semplice ed efficiente per ottenere una stima pessimistica della memoria necessaria per un dato albero di join JT di tipo generale.

- per ogni join $<R_i,R_j>$ in JT
  se $<R_i,R_j>$ non e' join increasing, allora sostituisci il fattore di selettivita' $s_{ij}$ con $s_{ij}^*$ tale che $s_{ij}<s_{ij}^*$, $|R_i||R_j|s_{ij}^* \geq |R_i|+|R_j|$, e $s_{ij}^*$ sia minimo.
- stima la memoria necessaria per JT usando l'algoritmo A.

E' possibile provare la seguente diseguaglianza per la stima ottenuta:

$$o_m \leq e_r \leq 2ko_m$$

dove $o_m$ e' la memoria necessaria all'ottimo, $e_r$ e' la stima ottenuta, e k e' definita come segue:

$$k = \prod_{<R_i,R_j> \varepsilon E} s_{ij}^*/s_{ij}$$

Si puo' dire che k riassume la differenza tra l'albero di partenza e l'albero che viene



effettivamente utilizzato per la stima.

3.4 Relazioni tra tempo e memoria

Si mostrera' ora come, sotto certe ipotesi, una soluzione ottima rispetto alla memoria richiesta sia vicina alla soluzione ottima rispetto al tempo impiegato. Il risultato non e' sorprendente se si osserva che il tempo richiesto per eseguire un join aumenta con le dimensioni delle relazioni coinvolte. Cosi', se abbiamo successo nel limitare la memoria richiesta,possiamo aspettarci di ottenere una buona soluzione approssimata anche dal punto di vista del tempo di risposta.

Dato un piano di esecuzione S, sia t(S) il tempo richiesto da S (cioe' il tempo necessario per rispondere alla query seguendo S), nel caso di albero join increasing e' possibile provare il seguente teorema.

**Teorema** Siano S e T rispettivamente i piani di esecuzione ottimi rispetto alla memoria ed al tempo per un albero join increasing. Allora i tempi t(S) and t(T) soddisfano la seguente diseguaglianza: $t(S) < (1+\varepsilon)\ t(T)$, dove $\varepsilon = 1\ /\ (\min |R_i|)$

Il teoreme precedente mostra che se la dimensione delle relazioni coinvolte e' molto grande, allora la soluzione ottima dal punto di vista della memoria centrale impegnata e' molto vicina alla soluzione ottima rispetto al tempo.

## 4. Il caso generale

Se il grafo di join non e' un albero, proponiamo un altro approccio euristico.

**Definizione**: Una componente join decreasing di un grafo di join JG e' un sottografo connesso join decreasing di JG. L'approccio proposto e' il seguente:

1. Trova le componenti join decreasing; esegui i join contenuti nelle componenti join decreasing;

sia G' il grafo ottenuto in tal modo.

2. Trova un minimum spanning tree di G'.
3. Applica l'algoritmo A allo spanning tree cosi' ottenuto ed esegui il piano calcolato dall'algoritmo A.

Si noti come il minimo albero ricoprente vada considerato minimo nel senso del prodotto dei fattori di selettivita' degli archi che ne fanno parte.

L'algoritmo e' intuitivamente basato sul fatto che e' meglio eseguire i join decreasing al piu' presto, in modo da eseguire i join increasing con relazioni piu' piccole possibile. Non diamo alcun bound formale sul risultato dell'algoritmo, ma e' possibile modificare la prova del teorema 1 in modo da provare il seguente:

**Teorema** Data una query Q ed una costante k, il problema di trovare un piano di esecuzione che ottenga una soluzione che non sia maggiore di k volte la soluzione ottima e' NP-completo.

## 5. Conclusioni

In questo lavoro e' stato considerato il problema della ottimizzazione della memoria per interrogazioni non ricorsive. Il problema e' stato caratterizzato caratterizzato dal punto di vista della complessita' ed e' stato mostrato essere risolubile in tempo polinomiale per classi particolari di query. L'approccio mostrato e' robusto, in quanto la minimizzazione della memoria ha un effetto positivo anche sul tempo di risposta.

## Riferimenti bibliografici

[D] D.De Witt, R.H.Katz, F.Olken, L.D.Shapiro, M.R.Stonebraker, D.Wood. Implementation Techniques for main memory database systems. Proc. SIGMOD 1984.

[HK] T.Hibaraki, T.Kameda. On the Optimal Nesting Order for Computing N-



Relational Joins. ACM Trans. on Database Syst. vol.9, September 1984.

[KBZ] R.Krishnamurthy, H.Boral, C.Zaniolo. Optimization of Nontecursive Queries. Proc. VLDB 1986.

[KZ] R.Krishnamurthy, C.Zaniolo. Safety and Optimization of Horne Clause Queries. (J.Minker ed.) Foundations of Deductive Databases and Logic Programming, Workshop. Washington D.C. 1986.

[S] P.G.Sellinger et al. Access Path Selection in a Relational Database Management System. Proc. SIGMOD 1979.

[W] K.Y.Whang. Query Optimization in Office-by-Example, IBM Research report, RC 11571, 1985.

[Z] C.Zaniolo. The Representation and Deductive Retrieval of Complex Objects, Proc. of 11th VLDB 1985.