Tecnologie delle Basi di Dati M

Antonio Davide Calì, Michele Gazzetti

23 maggio 2014

WWW.ANTONIOCALI.COM
Anno Accademico 2013/2014
Docenti: Marco Patella, Paolo Ciaccia

Indice

Parte I
Top-K Queries

L'obiettivo delle *Top-K Queries* è di trovare, da un insieme di risultati potenzialmente molto grande, solo le k (con $k \geq 1$) migliori risposte. Per migliori risposte si può intendere più importanti, interessanti, rilevanti, ... Il bisogno di avere query di questo tipo nasce dalla varietà dei moderni scenari come gli e-commerce, Database scientifici, ricerche web, sistemi multimediali, etc... La definizione di *top-k query* richiede un sistema capace di "classificare" gli oggetti (il primo dei migliori risultati, il secondo, il terzo, ...). Per "classificare" si intendere l'ordinamento degli oggetti del database basato sulla loro "rilevanza" per la query.

Facciamo un esempio: si voglia prenotare un volo su un sito di una compagnia aerea, indicando da dove si vuole partire e dove si vuole arrivare, dando anche le date di partenza e di ritorno. Il sistema potrebbe non trovare alcun risultato con i valori immessi, ma attraverso le top-k queries potrebbe consigliare all'utente le migliori alternative disponibili, come ad esempio con un giorno di andata o ritorno differente, o un aereoporto di partenza diverso.

Un primo approccio ingenuo per la costruzione di top-k queries potrebbe essere il seguente: assumiamo che data una *query* q esista una *funzione di successo* (o di *scoring*) S che assegna ad ogni tupla t un punteggio numerico in accordo a come le tuple sono classificate e successivamente ne fa ordinamento secondo S. Ad esempio se abbiamo la seguente relazione (Fig. 1) possiamo immaginare la seguente funzione di succeso $S(t) = t.Points + t.Rebounds$.

Figura 1:

Name	Points	Rebounds	...
Shaquille O'Neal	1669	760	...
Tracy McGrady	2003	484	...
Kobe Bryant	1819	392	...
Yao Ming	1465	669	...
Dwyane Wade	1854	397	...
Steve Nash	1165	249	...
...	...	...	...

L'algoritmo *naive* ha dunque questa struttura:

- **Input:** una query q, un set di record R
- **Output**: le k tuple con punteggio più alto rispetto alla funzione di successo S

```
begin.
        for each t in R
                compute S(t);
        sort tuples on their scores;
```

```
        return the first k highest-scored tuples;
end.
```

Riuscire a computare la *top-k query* utilizzando l'algoritmo appena descritto risulta essere molto costosa per database di grandi dimensioni, poichè richiede di ordinare un grand numero di informazioni. Il problema risulta essere anche peggiore se l'input consiste in attributi appartenenti a più di una relazione: si veda l'esempio in figura (Fig. 2) per maggiore chiarezza, in cui la funzione di successo S è rimasta invariata $S(t) = t.Points + t.Rebounds$.

Figura 2:

Name	Points	...
Shaquille O'Neal	1669	...
Tracy McGrady	2003	...
Kobe Bryant	1819	...
Yao Ming	1465	...
Dwyane Wade	1854	...
Steve Nash	1165	...
...	...	...

Name	Rebounds	...
Shaquille O'Neal	760	...
Tracy McGrady	484	...
Kobe Bryant	392	...
Yao Ming	669	...
Dwyane Wade	397	...
Steve Nash	249	...
...	...	...

Il problema si aggrava poichè ora dobbiamo anche effettuare il join fra tutte le tuple, il che richiede un ulteriore operazione di un certo costo. Si noti che nell'esempio precedente il join è 1:1, ma in generale può essere M:N (in cui cioè ogni tupla può fare il join con un numero arbitrario di tuple).

1 Top-k Query in SQL

Esprimere una *top-k query* in SQL richiede la capacità di **ordinare** le tuple in base al loro punteggio e **limitare** la cardinalità dell'output alle sole k tuple.

Consideriamo ora un primo caso di query scritta con lo schema seguente, il quale utilizza lo *standard SQL* in cui è presente solo la possibilità di ordinamento (non è consentito limitare il risultato)

```
SELECT   <some attributes>
FROM     R
WHERE    <Boolean conditions>
ORDER BY S(...)  [DESC]
```

e facciamo dunque due esempi.

Esempio A

```
SELECT   *
FROM     UsedCarsTable
WHERE    Vehicle='Audi/A4' AND Price <= 21000
ORDER BY 0.8*Price + 0.2*Mileage
```

Esempio B

```
SELECT   *
FROM     UsedCarsTable
WHERE    Vehicle='Audi/A4'
ORDER BY 0.8*Price + 0.2*Mileage
```

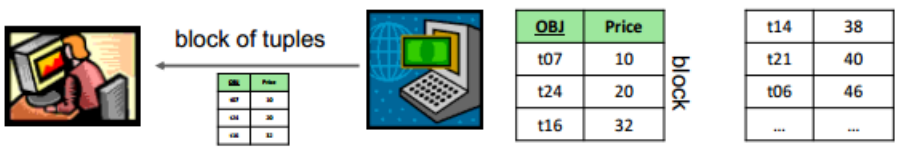
I valori 0.8 e 0.2 presenti nell'ORDER BY, anche chiamati *pesi*, sono un modo per normalizzare le nostre preferenze su Prezzo e Chilometraggio. La query A potrà perdere alcune risposte che potrebbero risultare rilevanti, e si parla in questo caso di *near-miss*, come ad esempio una macchina con un prezzo di 21.500$ ma con basso chilometraggio. La query B invece ritorna tutte le Audi/A4 presenti nel database, e in questo caso si parla di *information overload*, e la situazione risulta anche peggiore se non specifichiamo alcun tipo di veicolo (quindi eliminando la clausola WHERE il quale restituirà tutte le tuple della relazione).

Prima di considerare altre soluzioni, diamo un occhiata più da vicino a come il *server dbms* manda i risultati di una query ad un'applicazione client. Da lato client si lavora "una tupla per volta" usando, ad esempio, un metodo specifico come **rs.next()**. Anche se potrebbe sembrare, questo non vuol dire che il risultato totale viene trasmesso una tupla per volta dal server al client. La maggior parte dei *dbms* implementa una caratteristica chiamata *row blocking* che ha lo scopo di ridurre l'*overhead* di trasmissione.

Il *row blocking* funziona come descritto di seguito (si controlli la figura per una maggiore chiarezza):

1. Il *dbms* alloca alcuni buffer (un "blocco") da lato server
2. Riempe i buffer con le tuple risultanti dalla query
3. Invia l'intero "blocco" di tuple al client
4. Il client consuma (legge) le tuple nel blocco
5. Si ripete il passo 2 fino a che non vi sono più tuple (righe) disponibili nel risultato.

Figura 3:



Perchè il solo *row blocking* non è sufficiente? Cioè, perchè abbiamo bisogno del "k"? Razionalmente si potrebbe pensare al *fetch* delle sole tuple di cui si ha bisogno. Vediamo un esempio in DB2. La grandezza del "blocco" è stabilita quando l'applicazione si connette al database (grandezza di default: 32Kb). Se il buffer può mantenere, diciamo, 1000 tuple ma l'applicazione controlla solo,

diciamo, le prime dieci abbiamo uno spreco di risorse: infatti effettuiamo il *fetch* da disco e processiamo troppe (1000) tuple, successivamente trasmettiamo troppi dati (sempre le 1000 tuple) sulla rete. Se riducessimo la grandezza del blocco, potremmo incorrere in un *overhead* in trasmissione per le query che hanno risultati con un set elevato. Si potrebbe pensare anche di connettersi più volte al database per modificare ad ogni connessione la grandezza del blocco di trasmissione a seconda se voglia processare top-k query o meno: purtroppo anche questo approccio non è scalabile, poichè la richiesta di connessione risulta essere più pesante che il calcolo stesso. Dobbiamo sempre ricordarci che non abbiamo solo una query, la nostra applicazione potrebbe consistere in un insieme di query ognuna delle quali con delle specifiche differenti. Dobbiamo inoltre osservere che il *dbms* non sa nulla sulle intenzioni del client, cioè *ottimizza* e *calcola* la query come se dovesse trasmettere l'intero set del risultato.

Abbiamo visto finora le top-k query applicate a SQL standard. Ora le vedremo applicate a SQL extended, il quale permette di aggiungere predicati utili per la risoluzione delle query.

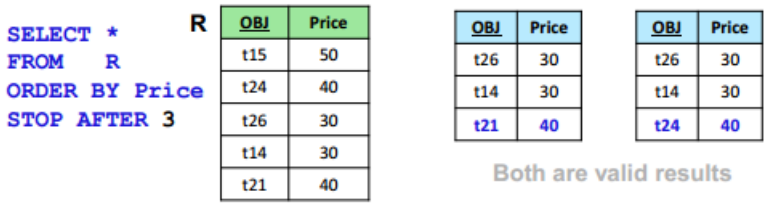
Il primo passo di sopporto alle *top-k* query è semplice: estendere SQL con una nuova clausola che esplicita la cardinalità limite del risultato. La sintassi proposta è la seguente

```
SELECT <some attributes>
FROM <some relation(s)>
WHERE <boolean conditions>
[GROUP BY <some grouping attributes>]
ORDER BY S(...) [DESC]
STOP AFTER k
```

dove k è un intero positivo. Questa è la sintassi proposta in [CK97], ma la maggior parte dei *dbms* hanno estensioni equivalenti proprietarie, ad esempio **FETCH FIRST k ROWS ONLY** in DB2, oppure **LIMT TO k ROWS** in ORACLE, o ancora **LIMIT k** in Postgres, ... [CK97] permette anche espressione numeriche, scorrelate con il resto della query, al posto del valore k.

Consideriamo una *top-k* query con la clausola **STOP AFTER k**. Concettualmente il resto della query è calcolata come al solito in accordo a una relazione T, successivamente *solo le prime k tuple di T* diventano parte del risultato. Se T contiene k o meno tuple, allora la clausola STOP AFTER k non ha alcun effetto; se invece contiene più di un set di tuple che soddisfano la clausola ORDER BY, allora ognuno di questo set risulta essere una risposta valida (e si parla di semantica non-deterministica). Si veda la seguente figura (Fig. 4) per una maggiore comprensione.

Figura 4:



Se infine non è presente la clausola ORDER BY, allora qualsiasi set di k tuple della tabella T è una soluzione valilda e corretta.

Mostriamo di seguito alcuni esempi di *top-k query*.

Esempio 1 Il miglior giocatore di NBA (considerando canestri e rimbalzi)

```
SELECT *
FROM NBA
ORDER BY Points + Rebounds DESC
STOP AFTER 1
```

Esempio 2 I due ristoranti cinesi più economici

```
SELECT *
FROM RESTAURANTS
WHERE Cuisine='chinese'
ORDER BY Price
STOP AFTER 2
```

Esempio 3 Il 5% di impiegati più pagati (si vede in questo caso una top-k query con un'espressione numerica invece che un valore numerico di k)

```
SELECT E.*
FROM EMP AS E
ORDER BY E.Salary DESC
STOP AFTER (SELECT COUNT(*)/20 FROM EMP)
```

Esempio 4 Le 5 migliori Audi-A4 basate su prezzo e chilometraggio

```
SELECT *
FROM USEDCARS
WHERE Vehicles='Audi/A4'
ORDER BY 0.8*Price + 0.2*Mileage
STOP AFTER 5
```

Esempio 5 I due hotel più vicini all'aeroporto di Bologna (si utilizza una *top-k* query con *distance* join in cui *Location* è un "punto" UDT, User-defined Data Type, e *distance* è un UDF, User-Defined Function)

```
SELECT H.*
FROM HOTELS AS H, AIRPORTS AS A
WHERE A.Code = 'BLQ'
ORDER BY distance(H.Location,A.Location)
STOP AFTER 2
```

1.1 Operatore Top

Per ciò che riguarda il calcolo, ci sono due aspetti base da considerare:

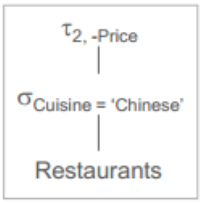
- Il **tipo** di query: a 1 relazione, su più relazioni, su risultati aggregati, ...
- Il **metodo** di accesso: senza indice, con indici su alcuni attributi di ranking, con indici su tutti gli attributi di ranking.

Il caso più semplice da analizzare è la *top-k selection query*, dove è coinvolta una sola relazione, la quale possiede la seguente semantica

```
SELECT <some attributes>
FROM R
WHERE <Boolean conditions>
ORDER BY S(...) [DESC]
STOP AFTER k
```

Per ragionare in maniera concisa sulle strategie di valutazione alternative, dobbiamo per prima cosa estendere l'algebra relazionale. A tal fine introduciamo l'*operatore logico TOP*, denotato con $\tau_{k,S}$, che ritorna le k tuple con miglior risultato in accordo alla funzione di successo S. Se non diversamente specificato, assumiamo S come funzione massimizzatrice. Si veda l'esempio in figura (Fig. 5) in cui nel metodo di accesso viene utilizzato anche l'operatore logico *top*.

Figura 5:



Più avanti introdurremo una rappresentazione più potente in cui il *ranking* non è un semplice "limiting" (cioè un limite alla cardinalità del risultato), ma un "cittadino di prima-classe".

Come possiamo implementare l'operatore logico top*?* Come può essere calcolato? Ricordiamo che abbiamo bisogno di operatori fisici che implementino operatori logici. Esistono 2 casi rilevanti:

1. **Top-Scan:** il flusso di tuple in input all'operatore Top è già ordinato secondo la funzione di successo S, in questo caso è sufficiente leggere (consumare) solo le prime k tuple dall'input. Il *Top-scan* può lavorare in *pipeline*: può restituire una tupla appena la legge.

2. **Top-Sort:** il flusso di tuple in input non è ordinato secondo la funzione di successo S. Risulta essere il caso tipico: piuttosto che ordinare l'intero input possiamo effettuare un *in-memory sort*. Il *Top-Sort* non può lavorare in *pipeline*: ha infatti bisogno di leggere l'intero input prima di poter restituire la prima tupla.

1.1.1 Top-Sort

L'idea dietro il metodo *Top-Sort* è di mantenere in un buffer B della memoria principale solo le miglior k tuple viste fino a quel punto. La *ratio* dietro questo procedimento è che se una tupla t non si trova nelle top-k tuple viste finora, allora t non può far parte del risultato.

Il problema cruciale è come organizzare B in modo tale che le operazioni di ricerca, inserzione e rimozione possano essere effettuate efficientemente. Siccome il buffer B dovrebbe comportarsi come una *coda prioritaria* (in cui la priorità è data dal punteggio ottenuto dalla tupla), può essere implementata usando un *heap*.

Per la realizzazione e la standardizzazione degli operatori fisici bisogna implementare i due metodi *open* e *next*.

OPEN *open*

- **input:** k valore intero, S funzione di successo

```
1. create a priority queue B of size k;
        //B can hold at most k tuples
        //B[i] is the current i-th best tuple,
        and B[i].score is its score
2. invoke open on the child node;
3. return;
```

Il metodo *open* crea un buffer B di grandezza k che potrà contenere le nostre k-tuple, si indica con *B[i]* la i-esima miglior tupla e con B[i].score il suo punteggio.

NEXT *next*

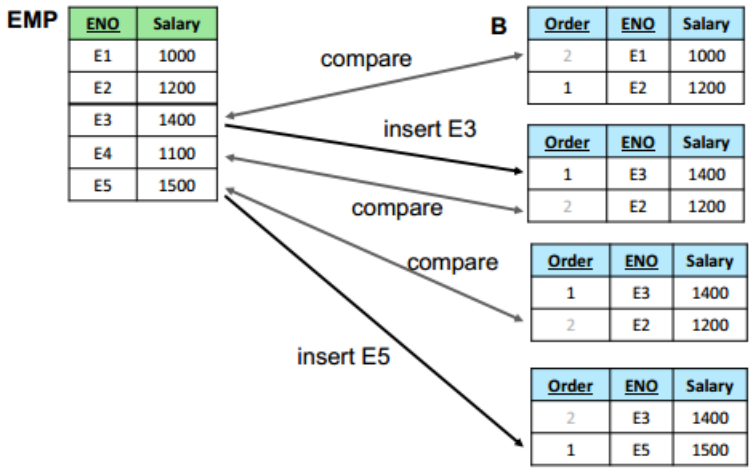
```
1. for i=1 to k //fills B with first k tuples
2.    t:=input_node.get_next();
      ENQUEUE(B,t) //insert t in B
3. while(input_node.has_next()) do
4.    t:=input_node.next();
      if S(t)>B[k].score then
            DELETE(B,B[k]);
         ENQUEUE(B,t);
5. return DEQUEUE(B) //returns the best tuple in B
```

Il metodo *next* inizialmente riempe B con le prime k tuple (per semplicità il pseudocodice non considera il caso di input con meno di k tuple). Successivamente, per ogni nuova tupla letta t, la confronta con B[k], cioè la peggior tupla presente al momento in B.

- Se S(t)>B[k].score allora B[k] viene cancellata da B e viene inserita t
- Se S(t)<B[k].score allora t non può essere una delle *top-k* tuple e di conseguenza viene scartata
- Se S(t)=B[k].score allora è sicuro ("safe") scartare la tupla t poichè Top ha una semantica non deterministica.

Un semplice esempio viene mostrato in figura (Fig. 6), in cui si ha k=2.

Figura 6:



2 Top-K Query Multidimensionali

Nel caso generale, la funzione di successo S coinvolge più di un attributo:

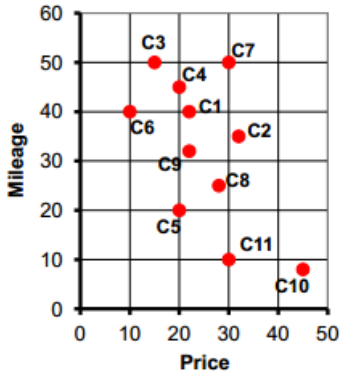
```
SELECT *
FROM USEDCARS
WHERE Vehicle = 'Audi/'A4
ORDER BY 0.8*Price + 0.2*Mileage
STOP AFTER 5;
```

Se nessun indice è disponibile non possiamo che applicare un operatore di *Top-Sort* leggendo sequenzialmente tutte le tuple. Se invece è disponibile un indice sull'attributo Vehicle allora la situazione migliora, però ancora si ha dipendenza da quante Audi/A4 siano presenti nel Database (dipende dalla selettività della clausole), e sinfine se se non è presente alcuna clausola di WHERE (è come se non ci fosse l'indice) si torna al caso precedente di *Top-Sort*.

Assumendo di avere l'*indice* sugli attributi di *ranking* (ad esempio su *Price* e *Milage*) come possiamo usarlo perr risolvere una *top-k query*? Che tipo di indice dovremmo utilizzare? Per prima cosa dobbiamo capire meglio la *geometria* alla base del problema.

Consideriamo lo spazio degli attributi sotto una visione geometrica considerando uno spazio bi-dimensionale (2-dim) sugli attributi (Price, Milage). Uno spazio siffatto è rappresentato dalla seguente figura (Fig. 7)

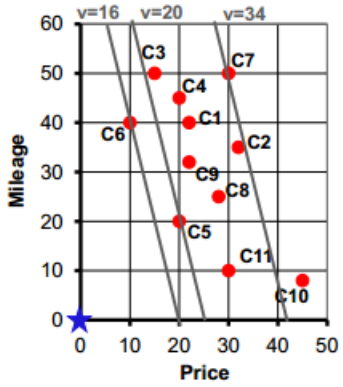
Figura 7:



Ogni tupla è rappresentata da un punto sulle due coordinate (p,m) in cui p è il valore di *price* e m il valore di *mileage*. Intuitivamente cercare di minimizzare la funzione $0.8 * Price + 0.2 * Mileage$ equivale a cercare un punto vicino $(0, 0)$: idealmente l'origine è il nostro "valore obiettivo" (una macchina gratuita con 0km).

Che ruolo hanno i "pesi" (ovvero i parametri) che attribuiamo nella funzione di successo S? I nostri parametri (ad esempio 0.8 e 0.2) sono essenziali per determinare il risultato. Utilizzando la figura seguente (Fig. 8) facciamo alcune osservazioni.

Figura 8:

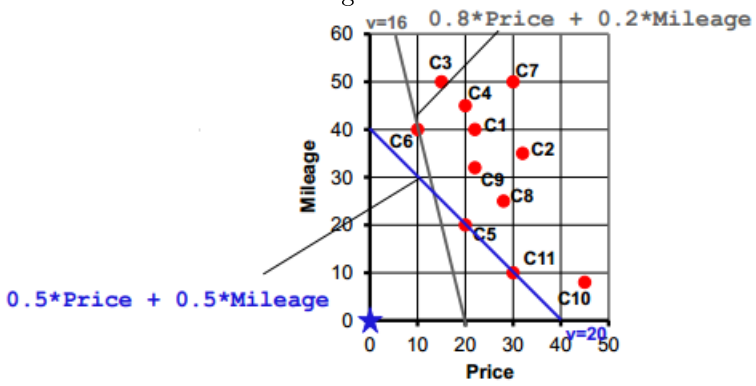


Consideriamo le varie rette l al variare del valore v, ovvero *l(v)*: essa è rappresentata dall'equazione $0.8 \cdot Price + 0.2 \cdot Mileage = v$ dove v risulta essere una costante. L'equazione può essere riscritta come $Mileage = -4 \cdot Price + 5 \cdot v$ tramite il quale è possibile vedere la pendenza di ogni retta ($pendenza = -4$). Per definizione tutti i punti che giaciono su una stessa retta *l(v)* sono egualmente "buoni". Con i nostri parametri (0.8,0.2) la miglior macchina è C6, la seconda miglior macchina è C5, etc. . . .

Dati i punti (p_1, m_1) e (p_2, m_2), i parametri sono un buon modo per determinare quale fra i due si avvicina di più al punto obiettivo (0,0).

Chiaramente, cambiando i valori dei pesi si potrà avere un cambiamento del risultato. Se sostituiamo i pesi $0.8 \cdot Price + 0.2 \cdot Mileage$ con $0.5 \cdot Price + 0.5 \cdot Mileage$ le migliori macchine risulteranno C5 e C11: si veda la figura (Fig. 9) per una maggiore comprensione.

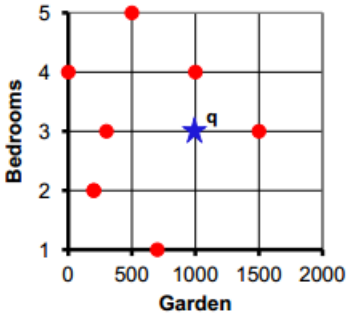
Figura 9:



D'altra parte, se i pesi non cambiano di molto, i risultati delle due *top-k query* avranno probabilmente un alto grado di sovrapposizione.

L'*obiettivo* di una query non è necessariamente il punto (0,0), può essere infatti un qualsiasi punto $q = (q_1, q_2)$ in cui q_i è il valore della query per l'i-esimo attributo. Per esempio: supponendo di voler cercare una casa con un giardino di 1000m^2 e 3 camere da letto, allora il punto obiettivo della query sarà (1000,3), in cui, come mostrato in figura (Fig. 10) vengono rappresentati su un'asse i m^2 del giardino e sull'altra il numero di camere da letto.

Figura 10:



In generale, per determinare la migliore tupla t fra un'insieme di tuple, calcoliamo la sua distanza dal punto obiettivo q: minore è la distanza da q migliore sarà t. Bisogna notare che la distanza dei valori può essere sempre convertita in *bontà del punteggio*, in modo che il punteggio maggiore significhi il miglior *match* (basta cambiare di segno e possibilmente aggiungere una costante), ovvero è sempre possibile passare dal calcolare il "punteggio" di una tupla, in cui punteggio maggiore implica tupla migliore, al calcolo della "distanza" della tupla rispetto all'obiettivo q, in cui distanza minore implica tupla migliore.

2.0.2 K-Nearest Neighbors (k-NN)

Puntando ad avere una gestione omogenea del problema quando usiamo un indice, è utile considerare le *distanze* invece che i punteggi.

Il modello che ne possiamo derivare è il seguente:

1. Uno spazio di attributi D-dimensionale (con $D \geq 1$) $A = (A_1, A_2, \ldots, A_D)$ fatto da *attributi di ranking*
2. Una relazione $R(A_1, A_2, \ldots, A_D, B_1, B_2, \ldots)$ dove $B_1, B_2, \ldots$ sono attributi 'superflui' della relazione
3. Un punto obiettivo della query $q = (q_1, q_2, \ldots, q_D)$ con ovviamente $q \epsilon A$.
4. Una fuzione $d : A \times A \rightarrow \mathbb{R}$ che misura la distanza tra punti di A: ad esempio $d(t, q)$ è la distanza tra il punto t e il punto q.

Sotto questo modello, una *top-k query* è trasformata in quella che viene chiamata **k-Nearest Neighbors (k-NN)**: dato un punto q, una relazione R, un intero

$k \geq 1$ e una funzione distanza d determinare le k tuple in R più vicine a q secondo d.

Le *distanze* più comuni che vengono utilizzate sono le funzioni $norme - L_p$

$$L_p(t,q) = \left(\sum_{i=1}^{D} |t_i - q_i|^p \right)^{\frac{1}{p}}$$

di cui i casi più rilevanti sono

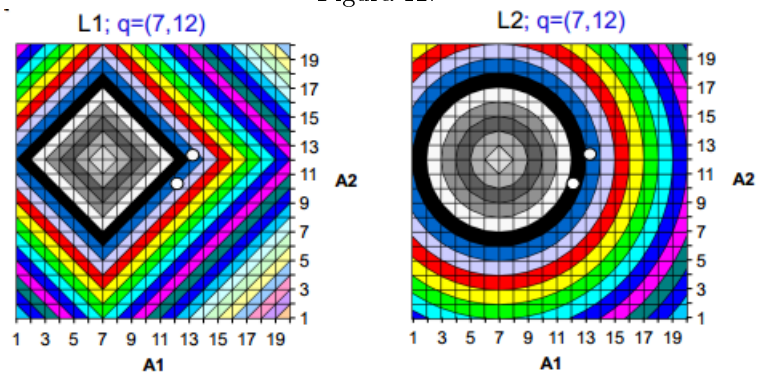
- *Distanza Euclidea:* $L_2(t,q) = \sqrt{\sum_{i=1}^{D} |t_i - q_i|^2}$
- *Distanza di Manhattan:* $L_1(t,q) = \sum_{i=1}^{D} |t_i - q_i|$
- *Distanza di Chebyshev:* $L_\infty(t,q) = max_i \{|t_i - q_i|\}$

e se ne veda la rappresentazione grafica su un piano (Fig. 11).

Figura 11:

Cambiare la funzione di distanza utilizzata conduce a un differente 'modello' degli spazi di attributi. Si vedano gli esempi in figura con funzioni L_1 e L_2 (Fig. 12) in cui ogni "striscia" colorata corrisponde al punto con distanza compresa tra v e $v+1$ con v intero.

Figura 12:



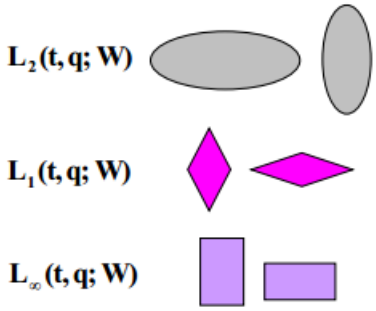
Si noti che per due tuple t_1 e t_2 è possibile avere $L_1(t_1, q) < L_1(t_2, q)$ e $L_2(t_2, q) < L_2(t_1, q)$ il che spiega che a seconda della distanza usata potrebbe risultare migliore in un caso t_1 e nell'altro t_2: di conseguenza la scelta della funzione di distanza influisce sul ranking.

Non abbiamo però ancora introdotto il "peso" degli attributi nelle funzioni distanza: infatti il loro uso comporta un modello "schiacciato" su alcune coordinate. Si faccia riferimento alle seguenti funzioni distanza con introdotto anche il "peso" w_i:

- $L_2(t, q; W) = \sqrt{\sum_{i=1}^{D} w_i \cdot |t_i - q_i|^2}$
- $L_1(t, q; W) = \sum_{i=1}^{D} w_i \cdot |t_i - q_i|$
- $L_\infty(t, q; W) = max_i \{w_i \cdot |t_i - q_i|\}$

le quali hanno vengono rappresentate attraverso figure "schiacciate" (Fig. 13).

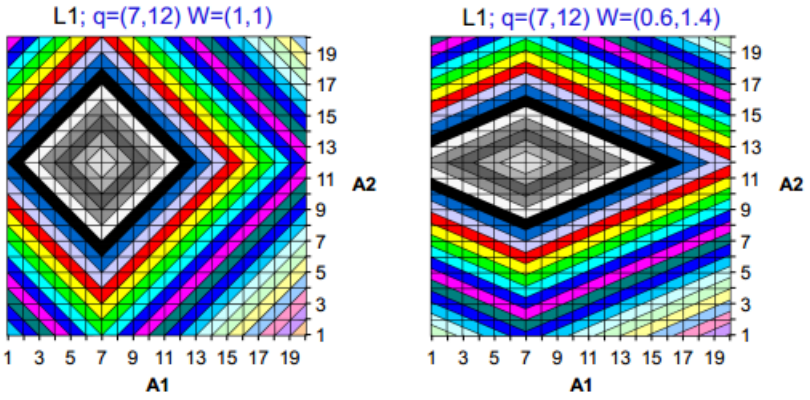
Figura 13:



Si noti che la nostra funzione $0.8 \cdot Price + 0.2 \cdot Mileage$ è un caso particolare di distanza L_1 "pesata".

È ora possibile modellare lo spazio di attributi con i pesi e la seguetne figura (Fig. 14) mostra l'effetto di utilizzare la distanza L_1 pesata.

Figura 14:



Notiamo che se $w_2 > w_1$ allora gli iper-rombi risultano più allungati lungo A1, cioè a parità di una eguale "differenza", essa inciderà molto di più su A2 che su A1, cioè se dati i punti ad esempio (3,0) e (0,5) che supponiamo essere su uno stesso iper-romboide, una differenza di 0,1 inciderà molto di più sull'attributo A1 che sull'attributo A2, cioè (2.9,0) risulterà essere sicuramente più vicino che (0,4.9).

2.1 Top-K Query con Indici

Utilizzare indici per la risolzione di Top-K query è possibile ma con alcuni accorgimenti. Utilizzare un indice B+tree multi-attributo che organizza le tuple seguendo l'ordine lessicografico degli attributi $A_1, A_2, \ldots, A_D$ (con ad esempio $A_1 = Price$ e $A_2 = Mileage$) non risulterà avere una buona performance: infatti risulta avere gli stessi problemi visti per la risoluzione di *window query*, cioè porta ad un pover clustering spaziale. Risulterà però possibile utilizzare D indici B+tree mono-attributo per risolvere delle *top-k join queries*, metodo che controlleremo più avanti. Risulta invece molto più interessante e conveniente utilizzare indici spaziali, come ad esempio R-tree.

Ricordiamo in breve cosa è un R-tree: è un albero *dinamico*, *perfettamente bilanciato* e *paginato* in cui ogni foglia contiene entry E nella forma $E = (chiave, RID)$ dove la *chiave* è una "chiave spaziale" (ovvero la posizione nello spazio) della tupla il cui indirizzo è dato da *RID*, ed ogni nodo *interno* invece contiene entry E nella forma $E = (MBR, PID)$ dove *MBR* (si veda anche *MBB*) è il "*Minimum Bounding Rectangle*" (si parla di rectangle e non di Box poichè stiamo introducendo anche i pesi nelle nostre funzioni che portano a "schiacciamento") di tutti i punti raggiungibili dal nodo il cui indirizzo è PID. Possiamo pensare di uniformare le entry usando il formato $E = (key, ptr)$ e diciamo che se N è il nodo puntato da E.ptr allora E.key è la "chiave spaziale" di N anche chiamata come Regione di N indicata con *Reg(N)*.

Prima di vedere come risolvere una *k-NN query* (che ricordiamo essere equivalente a una top-k query ma con ragionamenti applicati ad uno spazio D-dimensionale in cui si vuole minimizzare la distanza attraverso un'opportuna funzione), iniziamo a capire come avviene la *ricerca* in una *range query*.

Una *Range Query* è una query in cui: dati un punto q, una relazione R, un raggio di ricerca $r \geq 0$ e una funzione di distanza d si vogliono determinatare tutti gli oggetti t della relazione R tali per cui $d(t,q) \leq r$. La regione dello spazio $\mathbb{R}^{\mathbb{D}}$ definita come $Reg(q) = \{p : p \epsilon \mathbb{R}^{\mathbb{D}}, d(p,q) \leq r\}$ è anche chiamata *regione* della query ("region query"), e quindi il risultato è sempre contentuto nella regione della query). Esistono diverse varianti alla *range query* come la *point query* che si ottiene per $r = 0$ oppure la stessa *window query* che si ottiene utilizzando come funzione di distanza la L_∞ "pesata".

L'algoritmo per la valutazione di una *range query* è estremamente semplice:

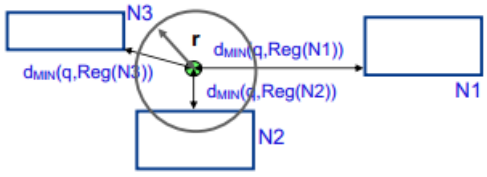
1. Si parte dalla radice, per ogni entry E e il corrispondente nodo N, si controlla se *Reg(N)* interseca *Reg(q)*.
2. Se il nodo è una foglia, controlla per ogni entry E tale per cui $E.key \epsilon Reg(q)$ se $d(E.key, q) \leq r$.

2.1.1 k-NN Query: Ricerca

Introduciamo ora il concetto di *distanza minima* che ci permetterà di calcolare la nostra k-NN "sbarazzandoci" del concetto di raggio di ricerca.

Dato un punto N sia $d_{MIN}(q, Reg(N)) = inf_p\{d(q,p) \,|\, p \epsilon Reg(N)\}$ la più piccola distanza possibile fra il punto q e un punto della regione Reg(N). Notiamo che la distanza minima $d_{MIN}(q, Reg(N))$ è un *lower bound* sulla distanza tra q e un qualsiasi punto raggiungibile di N: intuitivamente la distanza minima di un fra un punto q e una regione N è sicuramente minore o uguale alla distanza minima tra il punto q e un qualsiasi punto appartenente alla regione stessa.

Figura 15: Esempio distanza minima



Possiamo fare ora la seguente osservazione:

$$Reg(q) \cap Reg(N) \neq \emptyset \iff d_{MIN}(q, Reg(N)) \leq r$$

ovvero che la region query interseca la regione del punto N se e solo se la distanza minima fra il punto q e la regione N è minore o uguale al raggio di ricerca r. Nell'esempio mostrato in figura 15 solo N2 ha intersezione non vuota con la region query.

Il calcolo della distanza minima fra il punto q di cordinate $(q_1, q_2, \ldots, q_D)$ e la regione N in L_p è molto semplice. Indichiamo con

$$\triangle_i = \begin{cases} l_i - q_i & if\, l_i < q_i \\ q_i - h_i & if\, q_i > h_i \\ 0 & otherwise \end{cases}$$

in cui l_i e h_i indicano il valore minore e il valore maggiore della regione N nella i-esima coordinata. Allora la distanza minima tra q e la Reg(N) è data da

$$d_{MIN}(q, Reg(N)) = \sqrt[p]{\left(\sum_{i=1}^{D} (\triangle_i)^p\right)^{\frac{1}{p}}}$$

2.1.2 Algoritmo k-NNOptimal

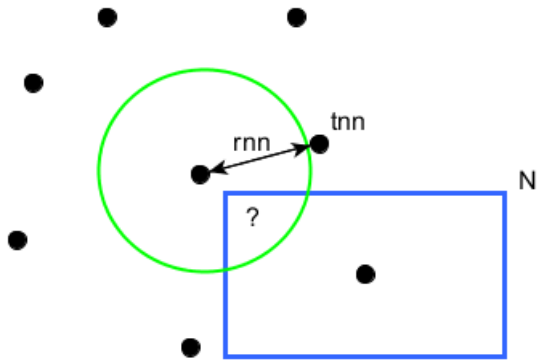
Presentiamo ora un algoritmo chiamato *k-NNOptimal* [BBK+97] per la risoluzione di *k-NN queries* che utilizzi un indice R-tree che risulta essere *ottimo* dal punto di vista di operazioni I/O: ovvero si legge il minor numero possibile di nodi dell'albero per la risoluzione della query. L'algoritmo può essere utilizzato anche per altre strutture indice come ad esempio l'M-tree.

Iniziamo con il caso base in cui $k = 1$ per poi, più avanti estendere l'algoritmo.

Dato il punto q target della nostra query, indichiamo con $t_{NN}(q)$ il punto più vicino a q (ovvero il $1 - NN$) in R, e indichiamo con $r_{NN} = d(q, t_{NN}(q))$ la sua distanza da q che risulterà dunque essere la distanza minima tra tutti i punti del dominio. Chiaramente r_{NN} si conoscerà solo al termine dell'algoritmo (è ciò che stiamo cercando, insieme a t_{NN}).

Esiste il seguente *teorema*: ogni algoritmo *corretto* per le $1 - NN$ *queries* deve visitare almeno tutti i nodi N la cui distanza minima è strettamente minore di r_{NN} ovvero tutti i nodi N per cui $d_{MIN}(q, Reg(N)) < r_{NN}$. La dimostrazione intuitiva è la seguente: supponiamo di avere un algoritmo che trovi t_{NN} non considerando tutti i nodi N per cui la distanza rispetto a q risulti strettamente minore di r_{NN}, allora l'algoritmo non può essere corretto, perchè non visitando tutti i nodi non può certificare che dentro ad uno di questi nodi non esista la vera t_{NN}. Si veda la figura (Fig. 16) per una maggiore comprensione.

Figura 16:



L'algoritmo *k-NNOptimal* usa una *coda prioritaria PQ* i cui elementi sono coppie del tipo $[ptr(N), d_{MIN}(q, Reg(N))]$. La coda PQ è ordinata secondo valori *crescenti* di $d_{MIN}(q, Reg(N))$ e le operazioni di inserimento funzionano come di seguito:

- *DEQUEUE(PQ)* estra da PQ la coppia con la minima "distanza minima"
- *ENQUEUE(PQ, [ptr(N), d_{MIN}(q,Reg(N))])* inserisce la coppia nella coda PQ in maniera ordinata.

Il *pruning* dei nodi è basato sulla seguente osservazione: se ad un certo punto dell'esecuzione dell'algoritmo abbiamo trovato un un punto t tale che $d(q,t) = r$, allora tutti i nodi N con distanza minima $d_{MIN}(q, Reg(N)) \geq r$ possono essere esclusi dalla ricerca poichè non possono portare a un miglioramento del risultato. Nella descrizione dell'algoritmo, il *pruning* delle coppie della coda PQ basata sul ragionamento appena descritto viene effettuata nell'*UPDATE(PQ)*. Con abuso di terminologia dire che "il nodo N è in PQ" significa che la corrispondente coppia [ptr(N), d_{MIN}(q,Reg(N))] sta in PQ.

Intuitivamente *k-NNOptimal* compie una *ricerca per range* con un *raggio di ricerca* variabile che man mano si restringe fino a che non è possibile più soluzioni.

Di seguito illustriamo il pseudo-codice dell'algoritmo:

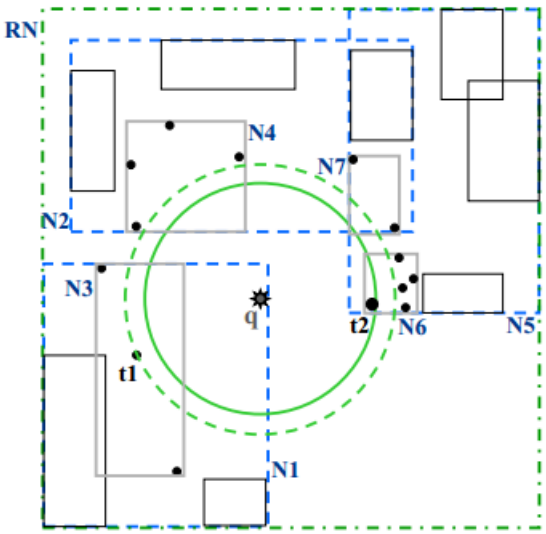
- **Input**: un *query point q*, un indice ad albero con nodo *radice RN*
- **Output**: $t_{NN}(q)$, il punto più vicino a q, e $r_{NN} = d(q, t_{NN}(q))$

Figura 17: Pseudo-codice k-NNOptimal

```
1.  Initialize PQ with [ptr(RN),0];        // starts from the root node
2.  r_NN := ∞;                             // this is the initial "search radius"
3.  while PQ ≠ ∅:                                  // until the queue is not empty...
4.      [ptr(N), d_MIN(q,Reg(N))] := DEQUEUE(PQ);  // ... get the closest pair...
5.      Read(N);                                   // ... and reads the node
6.      if N is a leaf then: for each point t in N:
7.                              if d(q,t) < r_NN then: {t_NN(q) := t; r_NN := d(q,t); UPDATE(PQ)}
                                // reduces the search radius and prunes nodes
8.                      else: for each child node Nc of N:
9.                              if d_MIN(q,Reg(Nc)) < r_NN then:
10.                                 ENQUEUE(PQ,[ptr(Nc), d_MIN(q,Reg(Nc))]);
11. return t_NN(q) and r_NN;
12. end.
```

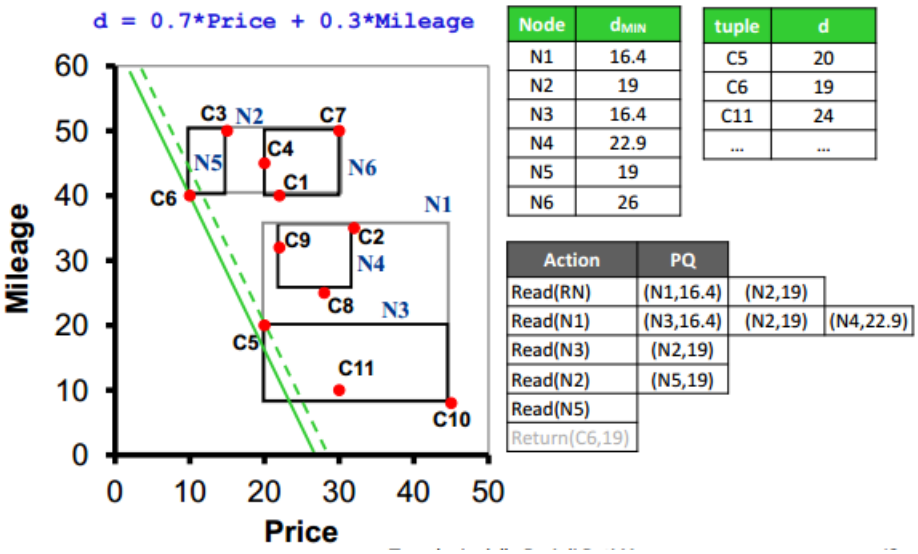
E vediamo il funzionamento di *k-NNOptimal* in azione con la seguente figura (Fig. 18), in cui i nodi sono numerati seguendo l'ordine con il quale vi si accede, gli oggetti sono numerati come vengono scoperti per migliorare (e dunque ridurre) il raggio di ricerca e in cui le foglie a cui si accede sono colarate in grigio chiaro.

Figura 18:



Vediamo un ulteriore esempio (Fig. 19) applicato ancora una volta alla miglior macchina usata disponibile sul nostro database.

Figura 19:



Node	d_{MIN}
N1	16.4
N2	19
N3	16.4
N4	22.9
N5	19
N6	26

tuple	d
C5	20
C6	19
C11	24
...	...

Action	PQ		
Read(RN)	(N1,16.4)	(N2,19)	
Read(N1)	(N3,16.4)	(N2,19)	(N4,22.9)
Read(N3)	(N2,19)		
Read(N2)	(N5,19)		
Read(N5)			
Return(C6,19)			

Dimostriamo ora che l'algoritmo è sia *corretto* che *ottimale*.

Per la *correttezza* dell'algoritmo la dimostrazione è ovvia: l'algoritmo non "scarta" nessun nodo a meno che non abbia una distanza minima maggiore rispetto al valore di raggio di ricerca corrente, il quale (il raggio di ricerca) è sicuramente maggiore o uguale a r_{NN}.

Per quanto riguarda l'*ottimalità* dobbiamo dimostrare il seguente teorema: l'algoritmo *k-NNOptimal* per *1-NN queries* non legge mai un nodo N la cui distanza minima è strettamente maggiore di r_{NN}, cioè non legge mai un nodo tale per cui $d_{MIN}(q, Reg(N)) > r_{NN}$.

La dimostrazione è la seguente:

1. Il nodo N (la cui distanza minima è strettamente maggiore di r_{NN}) viene letto solo se, ad un qualche passo dell'esecuzione, esso diventa il primo elemento della coda PQ

2. Sia N1 il nodo contente $t_{NN}(q)$ (ovvero il nodo che contiene la soluzione), e N2 il suo padre, e N3 il padre di N2 e così via fino a Nh=RN (con h =altezza dell'albero) in cui RN è la radice.

3. Osserviamo che, per definizione di distanza minima, si ha

   $$r_{NN} \geq d_{MIN}(q, Reg(N1)) \geq d_{MIN}(q, Reg(N2) \geq \ldots \geq d_{MIN}(q, Reg(Nh))$$

4. Ad ogni passo precedente al ritrovamente di $t_{NN}(q)$, uno e solo uno dei nodi $N1, N2, \ldots, Nh$ è presente nella coda prioritaria

5. Segue che N non potrà mai diventare il primo elemento di PQ

Intuitivamente, un nodo che non fa parte della catena che lega il nodo N1 a Nh non potrà mai essere visitato prima di qualsiasi nodo che ne fa parte.

Cosa succede se troviamo un nodo N la cui distanza sia esattamente r_{NN}? Il teorema di ottimalità non dice nulla a riguardo perchè non può dire nulla. Notiamo infatti regioni di questo genere sono casi di *parità* di cui non si può sapere l'ordine relativo con cui verranno inserite in PQ. I possibili casi sono due:

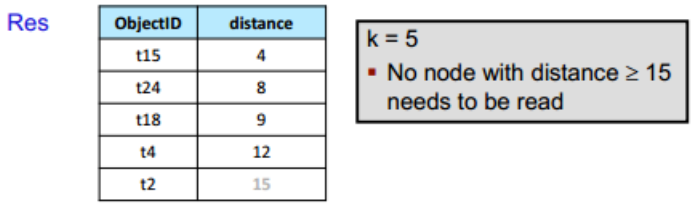
1. Il *Nearest Neighbor* è in un nodo la cui regione ha distanza minima strettamente minore di r_{NN}. In questo cosa *nessun nodo* con distanza $d_{MIN}(q, Reg(N)) = r_{NN}$ verrà letto
2. Il *Nearest Neighbor* si trova in un nodo la cui regione ha distanza minima esatamente uguale a r_{NN}. In questo caso tutto dipende da come PQ gestisce i casi di parità: nel caso peggiore *tutti i nodi* con $d_{MIN}(q, Reg(N)) = r_{NN}$ saranno letti.

Possiamo ora facilmente estendere l'algoritmo al caso in cui $k \geq 1$ utilizzando:

- Una struttra dati, che chiamiamo *Res*, nel quale manteniamo i k oggetti più vicini finora trovato con la loro distanza da q.
- Come "attuale raggio di ricerca" viene usata la distanza r_{k-NN} dell'attuale k-esimo *Nearest Neighbor* di q, ovvero il k-esimo elemento di *Res*.

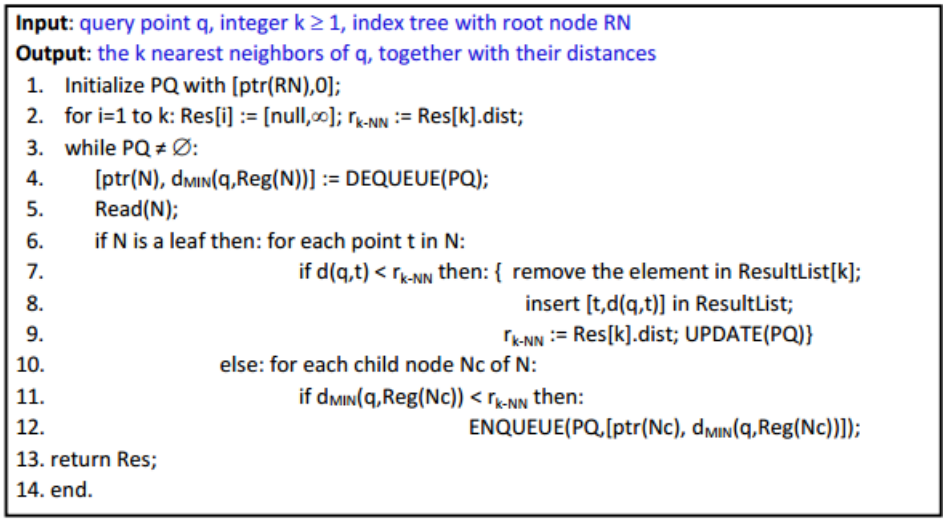
Si veda un semplice esempio di *Res* (Fig. 20) con $k = 5$ in cui nessun nodo con distanza minima ≥ 15 verrà letto.

Figura 20:



Tutto il resto dell'algoritmo rimane invariato. Di seguito vediamo dunque la sua implementazione in pseudo-codice.

Figura 21: k-NNOptima con $k \geq 1$



2.2 Distance Browsing

Ora sappiamo come risolvere le *top-k selection queries* usando un indice multidimensionale; ma che succede se la nostra query fosse

```
SELECT *
FROM USEDCARS
WHERE Vehicle = 'Audi/'A4
ORDER BY 0.8*Price + 0.2*Mileage
STOP AFTER 5;
```

e avessimo un R-tree su (Price,Mileage) costruito però su *tutte* le macchine? I migliori $k = 5$ risultati ritornati dall'indice non saranno necessariamente Audi/A4.

Una prima idea potrebbe essere di avere degli indici chiamati "colorati", ovvero con informazioni aggiuntive, ad esempio un R-tree su (Price, Mileage) che contenga anche informazioni sul tipo di veicolo potrebbe risolvere completamente il problema poichè la query verrebbe risolta via INDEX ONLY. Il problema nasce comunque nella realtà, poichè è difficile pensare che un indice aggiunga "tutte" le informazioni aggiuntive non necessarie all'indice stesso.

Ecco che in casi come questo possiamo usare una variante del *k-NNOptimal* che supporta la così detta *distance browsing* [HS99] o *incremental NN queries.*

Per il caso $k = 1$ la logica globale per l'uso dell'indice è la seguente:

- ottenere dall'indice il primo *Nearest Neighbor*

- se soddisfa le condizioni della query (ad esempio Audi/A4) allora fermati, altrimenti ottieni il secondo *Neareast Neighbor* e ripeti il passaggio fintato che non viene trovato il primo oggetto che soddisfa le clausole della query

Dal punto di vista dell'indice, a questo punto k non è più un parametro, ovvero l'indice non sa più quando effettivamente fermarsi nella restituzione del prossimo NN: ovvero, utilizzando sempre il caso $k = 1$, per l'indice basterebbe restituire solo il primo NN, ma se che questo soddisfi o meno la clausola WHERE non è più compito suo saperlo. Si può dunque immaginare che esista un "client" (per client si può intendere anche un operatore a livello più alto) che utilizza un algoritmo, quale next_NN che vedremo a breve, fintanto che non raggiunge il numero di tuple che gli servono e a quel punto chiude la "connessione" rilasciando le risorse sull'indice attraverso il metodo *close()*.

2.2.1 L'algoritmo next_NN

In una coda prioritaria PQ ora manteniamo sia le *tuple* che i *nodi*: se una Entry di PQ è una tupla t allora la sua distanza $d(q, t)$ è scritta come $d_{MIN}(q, Reg(t))$ (giusto per uniformità, in qui la distanza da q a t viene scritta sempre come distanza minima tra q e la regione t, regione che risulta essere puntuale). Notiamo che in questo caso non è possibile alcun "potamento" (*pruning)* poichè non sappiamo quanti oggetti bisogna restituire prima di fermarsi, per il motivo prima discusso in cui il paramentro k non è più un parametro dell'indice. Prima di effettuare la prima chiamata all'algoritmo inizializziamo la coda PQ con [ptr(RN),0]. Quando una tupla t diventa il primo elemento della coda l'algoritmo la restituisce.

Vediamo lo pseudo-codice del metodo next_NN:

- **Input:** la *query point q*, l'indice ad albero con nodo radince *RN*
- **Output:** il prossimo *Nearest Neighbor* di q insieme alla sua distanza

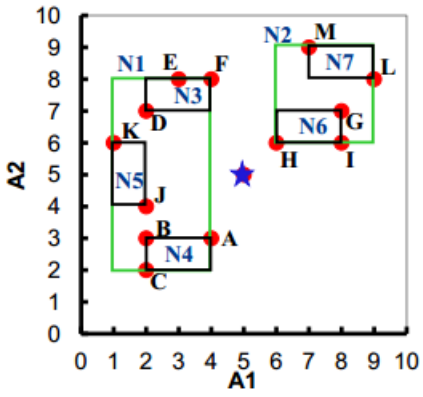
Figura 22: Pseudo-Codice next_NN

```
1. while PQ ≠ ∅:
2.     [ptr(Elem), d_MIN(q,Reg(Elem))] := DEQUEUE(PQ);
3.     if Elem is a tuple t then: return t and its distance // no tuple can be better than t
4.                 else: if N is a leaf then: for each point t in N: ENQUEUE(PQ,[t,d(q,t)])
5.                                 else: for each child node Nc of N:
6.                                             ENQUEUE(PQ,[ptr(Nc), d_MIN(q,Reg(Nc))]);
7. end.
```

Si vede che l'algoritmo, in caso abbia in testa alla lista un Nodo e successivamente una Tupla, deve esaminare per forza prima il Nodo poichè potrebbero esserci all'interno del Nodo delle tuple più vicine rispetto alla Tupla già presente. Se invece vi è una Tupla in testa, essa può essere restituita in quanto sicuramente è la tupla più vicina (in quel momento).

Vediamo l'algoritmo applicato ad un esempio (Fig. 23).

Figura 23:



Elem	d_{MIN}
A	√5
B	√13
C	√18
D	√13
E	√13
F	√10
G	√13
H	√2
I	√10
J	√10
K	√17
L	√25
M	√20
N1	√1
N2	√2
N3	√5
N4	√5
N5	√9
N6	√2
N7	√13

Action	PQ								
Read(RN)	(N1,√1)	(N2,√2)							
Read(N1)	(N2,√2)	(N3,√5)	(N4,√5)	(N5,√9)					
Read(N2)	(N6,√2)	(N3,√5)	(N4,√5)	(N5,√9)	(N7,√13)				
Read(N6)	(H,√2)	(N3,√5)	(N4,√5)	(N5,√9)	(I,√10)	(G,√13)	(N7,√13)		
Return(H,√2)	(N3,√5)	(N4,√5)	(N5,√9)	(I,√10)	(G,√13)	(N7,√13)			
Read(N3)	(N4,√5)	(N5,√9)	(I,√10)	(F,√10)	(G,√13)	(D,√13)	(E,√13)	(N7,√13)	
Read(N4)	(A,√5)	(N5,√9)	(I,√10)	(F,√10)	(G,√13)	(D,√13)	(E,√13)	(B,√13)	(N7,√13)	(C,√18)
Return(A,√5)	(N5,√9)	(I,√10)	(F,√10)	(G,√13)	(D,√13)	(E,√13)	(B,√13)	(C,√18)	
Read(N5)	...	...	...						
...									

L'algoritmo next_NN è solo un implementazione più generale del metodo *next* per gli indici che supportano le *k-NN queries incrementali*. In pratica, lo specifico tipo di query (che sia *range*, *k-NN*, *k-NN incrementale,* etc...) è un parametro passato all'indice nel metodo *open* così che si possa rendere sufficiente il semplice metodo *next*() che implementi il "next" voluto.

2.3 Top-k join 1:1 queries

In una *top-k join query* abbiamo $n > 1$ relazione di input e una funzione di *scoring* S definita sul risultato di join, ad esempio:

SELECT <some attributes>

```
FROM R1,R2,...,Rn
WHERE <join and local conditions>
ORDER BY S(p1,p2,...,pm) [DESC]
STOP AFTER k
```

dove p1,p2,...,pm sono i criteri di *scoring* (le "preferenze"). In questo caso il nostro criterio di bontà rappresentato dalla scoring function si compone da tanti "pezzi" presenti perl su relazioni diverse.

Vediamo subito alcuni esempi:

- Gli impiegati più pagati in rapporto al budget del dipartimento in cui lavorano

  ```
  SELECT E.*
  FROM EMP E, DEPT D
  WHERE E.DNO = D.DNO
  ORDER BY E.Salary / D.Budget DESC
  STOP AFTER 1
  ```

- Le due combinazioni di ristoranti-hotel più economiche nella stessa città italiana

  ```
  SELECT *
  FROM RESTAURANTS R, HOTELS H
  WHERE R.City = H.City
  AND R.Nation = ''Italy AND H.Nation = ''Italy
  ORDER BY R.Price + H.Price
  STOP AFTER 2
  ```

Una *top-k selection query* multidimensionale basate sulla funzione di scoring S(p1,p2,...,pm) può anche essere vista come un *caso particolare di join query* nel quale la relazione di input R è virtualmente "partizionata" in m parti, dove la j-esima parte R_i consiste nell'oggetto id e nell'attributo necessario per computare p_j. Tentiamo di capire cosa si vuol dire con ciò: una top-k query effettuata su una singola relazione R ma che si basi su una scoring function S definita su più "preferenze" può essere vista come una top-k query con join 1:1 in cui ogni preferenza della scoring function appartiene a una relazione diversa.

Ad esempio se S è definita come $S(t) = (t.Price + t.Milage)/(t.Year - 1970)$ possiamo "partizionare" la relazione USEDCARS come $UC1(CarID, Price)$, $UC2(CarID, Mileage)$, $UC3(CarID, Year)$. Una *top-k selection query* con la scoring function prima definita, potrebbe essere visualizzata dunque come una *top-k join* 1:1 *query* così scritta:

```
SELECT *
FROM USEDCARS UC1, USEDCARS UC2, USEDCARS UC3
WHERE UC1.CarID = UC2.CarID AND UC2.CarID = UC3.CarID
ORDER BY (UC1.Price + UC2.Mileage)/(UC3.Year-1970)
STOP AFTER 2
```

Nel seguente caso il join è sempre 1:1 (PK-PK join). Altri casi di partizionamento possono presentarsi come ad esempio $UC1(CarID, Price, Mileage)$, $UC2(CarID, Year)$.

Il caso in cui tutti i join sono su una stesso attributo di chiave è stato il primo caso che è stato largamente studiato. La sua rilevanza è dovuta al fatto che è il caso più semplice da trattare, la sua soluzione fornisce le basi per un caso più generale ed infine è il caso che più volte accade.

I due scenari in cui sono presenti join 1:1 che ha senso trattare sono i seguenti:

1. Per ogni "preferenza" p_j c'è un indice capace di trovare le tuple secondo l'attributo. Ovvero, se precedentemente avevamo detto che per riuscire a calcolare efficacemente una query top-k (o addirittura una window query, o ancora una range-query) attraverso indici su singoli attributi portava a grosse perdite di performance, tentiamo ora di riuscire ugualmente a gestire la situazione. Ovvero se effettivamente di una relazione R abbiamo due indici mono attributo (ad esempio un B+tree su *price* e uno su *mileage*), come risulta possibile applicarli per riuscire a risolvere effettivamente la top-k query (si ricade al caso in cui una top-k selection query può essere partizionata, in cui in questo caso ogni partizione viene rappresentata effettivamente dalle informazioni che un singolo indice B+tree può fornire).

2. Le "partizioni" di R sono partizioni reali, in cui ogni partizione è situata in diversi siti, ognuno dei quali fornisce informazioni solo su una parte degli oggetti

Storicamente il secondo scenario, molto spesso chiamato "middleware scenario", è quello che ha motivato lo studio di *top-k* 1:1 *join query*, il quale ha portato all'introduzione del primo algoritmo conosciuto, per altro per nulla banale.

2.3.1 Middleware Scenario

Il *middleware scenario* può essere descritto in maniera approssimata come segue:

1. Abbiamo un numero di "*sorgenti di dati*" (data source)

2. Le nostre richieste (query) possono coinvolgere diverse sorgenti allo stesso tempo

3. Il risultato della nostra query è ottenuto combinando in qualche modo il risultato tornato dalle diverse sorgenti

Queste query sono chiamate "*middleware query*" dato che richiedono la presenza di *middleware* (ovvero degli intermediari), il cui ruolo è di fare da intermediario tra l'utente/client e i vari data source/server.

Vediamo un esempio semplificato. Assumiamo si voglia organizzare un sito web che integra l'informazioni di due sorgenti: la prima sorgente "esporta" il

seguente schema *CarPrices*(*CarModel, Price*), mentre il secondo esporta *CarSpec*(*Make, Model, FuelConsumpion*). Dopo una prima fase di "riconciliazione" in cui si trasforma il nostro CarModel="Audi/A4" come segue

$$CarModel =' Audi/A4' \Leftrightarrow (Make, Model) = ('Audi', 'A4')$$

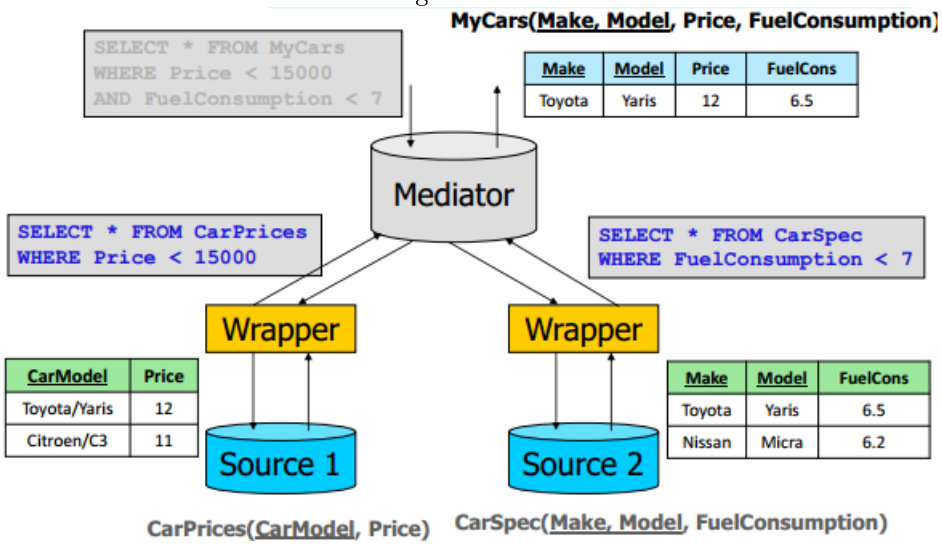
possiamo supportare ora query su entrambi gli attributi rimanenti *Price* e *FuelConsumption* come ad esempio trovare tutte le macchine il cui consumo è minore di 7 litri ogni 100km e con un costo minimo di 15000$.

Come ci riusciamo?

1. Inviamo la (sub-)query su *Price* alla fonte *CarPrices*
2. Inviamo la query sul consumo di carburante alla fonte *CarSpec*
3. Eseguiamo il *join* dei risultati

Si guardi la figura (Fig. 24) per una maggiore comprensione

Figura 24:



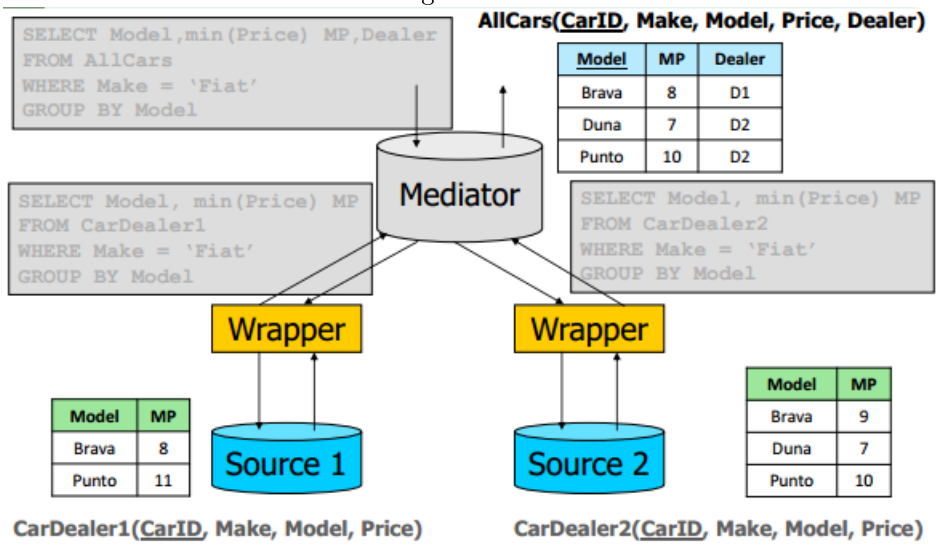
Vediamo un altro esempio in cui si voglia costruire un sito che integra informazioni di siti di m rivenditori di auto. Ogni rivenditore di auto CD_j può darci la seguente informazione $CarDealer_j$(*CarID, Make, Model, Price*) e il nostro obiettivo è di fornire ai nostri utenti le macchine più economiche disponibili, ossia di riuscire a supportare query del tipo: "Per ogni modello FIAT, quale ha un'offerta migliore?". Di nuovo come ci riusciamo?

1. Mandiamo la stessa (sub-)query a tutti i nostri data source
2. Facciamo l'unione dei risultati

3. Per ogni modello, prendiamo la miglior offerta e il corrispondente rivenditore

Per query di questo tipo, il mediatore è spesso chiamato "meta-broker" o "meta-search engine". Si veda di seguito lo schema di funzionamento (Fig. 25)

Figura 25:



Le *top-k middleware query* è solo un altro nome per le *top-k* 1:1 *join query*, che risulta appropriato quando i dati che necessitano di join 1:1 sono distribuiti. Sebbene lo scenario locale e quello distribuito abbiano proprietà differenti (ad esempio costi di comunicazione, disponibilità di risorse, etc...), per entrambe si possono applicare gli stessi principi (e algoritmi) per computare il risultato di una *query top-k*. In particolare in entrambi gli scenari ragioniamo in termini di "input": *data source* nel caso di sistema distribuito, *relazioni* in caso di scenario locale.

Per ragioni che presto chiariremo, il j-esimo input sarà indicato con L_j. Inoltre per semplificare la notazione ragioneremo in termini di "oggetti" invece che tuple per farne lo score globale: questo perchè è ragionevole dire che un oggetto o appartiene a due differenti input, mentre sarebbe sbagliato dire lo stesso per una tupla.

Se si vuole evitare di leggere tutti gli input, le seguenti assunzioni risultano essere abbastanza necessarie:

1. Ogni input L_j supporta un'interfaccia ad *accesso ordinato* (s.a):

$$getNext_{L_j}() \rightarrow (OID, Attributes, p_j)$$

Un accesso ordinato ottiene l'*id* del prossimo miglior oggetto (rappresentato dall'*OID*) secondo il suo score parziale p_j (ovvero lo score che ottiene

su quel determinato input seguendo una determinata scoring function parziale) assieme, possibilmente, ai restanti attributi richiesti dalla query. Per questo motivo L_j risulta essere una *lista con priorità* (ranked list) il che giustifica il suo nome ("L" sta per lista).

2. Ogni input L_j supporta anche un'interfaccia ad *accesso casuale* (r.a):

$$getScore_{L_j}(OID) \rightarrow p_j$$

ovvero, dato l'id di un oggetto, si riesca ad avere il suo risultato parziale p_j.

3. L'identificatore di un oggetto OID è un identificatore *globale*: dato un oggetto esso ha lo stesso identificatore su tutti gli input

4. Ogni input comprende lo stesso set di oggetti, ovvero se un oggetto X è presente su un input j, deve essere presente anche su tutti i restanti (questo per evitare di dover trattare i valori nulli, in cui la situazione si complica).

L'assunzione 3 e 4 sono banalmente verificate se la top-k 1:1 join query è eseguita localmente poichè le liste di join sono semplici *ranking* diversi di una stessa relazione.

In un ambiente distribuito l'assunzione 3 è difficilmente verificata: la sfida dunque di riuscire ad "unificare" le descrizioni fornite dalle diverse data source. Se l'assunzione 4 non è verificata potrebbe verificarsi la perdita di alcuni risultati parziali (poichè non presente un determinato oggetto su un determinato input): la strategia da attuare dipende dalla specifica *scoring function* (ad esempio se Budget non è definito allora neanche Salary/Budget lo è). Per riuscire a supportare l'accesso ordinato è possibile utilizzare l'algoritmo *next_NN*, invece per supportare l'accesso casuale è richiesto un indice su chiave primaria.

Vediamo un esempio in cui si vogliano aggregare le recensioni di alcuni ristoranti (Fig. 26). Si nota che il vincitore globale non necessariamente è il migliore a livello locale.

Figura 26:

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

```
SELECT *
FROM MangiarBene MB, PaneeVino PV
WHERE MB.Name = PV.Name
ORDER BY MB.Score + PV.Score DESC
STOP AFTER 1
```

Note: the winner is never the best locally!

Name	Global Score
Il desco	16.8
Al vecchio mulino	16.7
Da Gino	16.5
La tavernetta	16.0
Le delizie del palato	13.0
Tutti a tavola!	12.4
Acqua in bocca	11.5

Per riuscire ad ottenere un approccio univoco al problema considereremo:

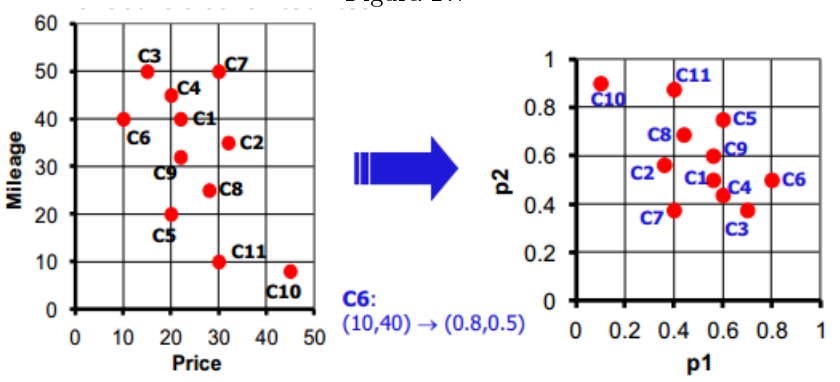
- Una *top-k* 1:1 *join query* $Q = (Q_1, Q_2, \ldots, Q_m)$ in cui ogni Q_j è la sub-query richiesta alla j-esima relazione/data source.
- Ogni oggetto o ritornato dall'input L_j ha associato un punteggio locale/-parziale $p_j(o)\,\epsilon\,[0,1]$ dipendente da Q_j (ovvero dipendente dalla sub-query che viene richiesta al'input j-esimo)
  - Per convenzione i punteggi sono normalizzati con il punteggio più alto che si ottiene
  - Questo vincolo può essere facilmente rilassato: ciò che conta infatti è sapere il miglior e il peggior valore possibile di p_j
  - L'iper-cubo $[0,1]^m$ è chiamato "spazio dei punteggi" (*score space*)
- Il punto $p(o) = (p_1(o), p_2(o), \ldots, p_m(o))\,\epsilon\,[0,1]^m$ rappresenta l'oggetto o nello spazio dei punteggi, in cui dunque un oggetto può essere rappresentanto il piano calcolando per ogni coordinata il suo punteggio parziale che ottiene su quella determinata coordinata, in cui la coordinata in questo caso rappresenta il valore di bontà di una determinata "preferenza"
- Il punteggio totale/globale $S(o)$ dell'oggetto o è calcolato mediante una funzione di successo S (una *scoring function*) che combina in qualche modo tutti i possibili punteggi locali di o:

$$S : [0,1]^m \to \mathbb{R}$$

$$S(o) \equiv S(p(o)) = S(p_1(o), p_2(o), \ldots, p_m(o))$$

Riprendiamo in esempio lo spazio degli attributi, considerando lo spazio a due dimensioni $A = (Price, Mileage)$. Sia Q1 la sub-query su *Price* e Q2 la sub-query su *Mileage*. Possiamo impostare $p_1(o) = 1 - \frac{o.Price}{MaxP}$ e $p_2(o) = 1 - \frac{o.Mileage}{MaxM}$ in cui MaxP e MaxM sono rispettivamente il massimo prezzo (per esempio $MaxP = 50000$) e il massimo chilometraggio (ad esempio $MaxM = 80000$). Gli oggetti in A (Fig. 27 a sinitra) vengono mappati nello spazio dei punteggi (Fig. 27 a destra). Gli ordini relativi su ogni coordinata (*local ranking*) rimane invariato

Figura 27:



Alcune delle *scoring function* più comuni sono le seguenti:

- SUM (AVG): si sommano i valori delle "preferenze" (sarebbe quasi identico se ne calcolassimo la media).

$$SUM(o) \equiv SUM(p(o)) = p_1(o) + p_2(o) + \ldots + p_m(o)$$

- WSUM (Weighted sum): attribuendo pesi differenti sugli attributi di ranking

$$WSUM(o) \equiv WSUM(p(o)) = w_1 \cdot p_1(o) + \ldots + w_m \cdot p_m(o)$$

- MIN (Minimum): considera semplicemente il peggior punteggio parziale

$$MIN(o) \equiv MIN(p(o)) = min\{p_1(o), p_2(o), \ldots, p_m(o)\}$$

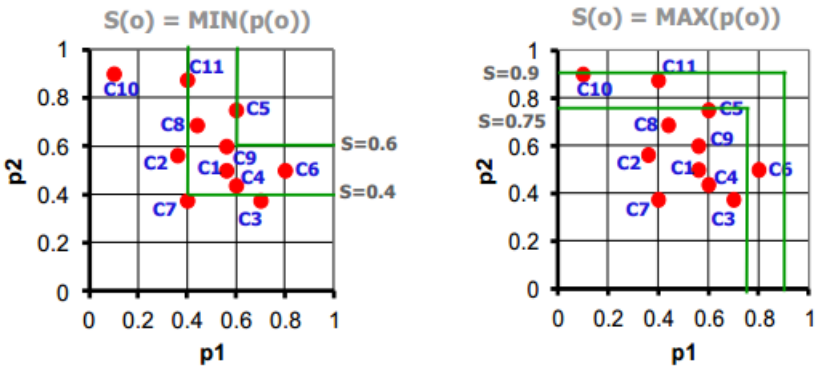
- MAX (Maximum): considera semplicemente il miglior punteggio parziale

$$MAX(o) \equiv MAX(p(o)) = max\{p_1(o), p_2(o), \ldots, p_m(o)\}$$

Si ricordi, anche se utilizziamo la funzione MIN, che vogliamo sempre ottenere i k oggetti con il più alto punteggio globale.

Come abbiamo fatto nello spazio degli attributi in cui abbiamo definito curve equi-distanti, possiamo ugualmente definire nello spazio dei punteggi curve con lo stesso punteggio. Si guardi la figura (Fig. 28) in cui vengono mostrate le curve delle funzioni MIN e MAX.

Figura 28:



Possiamo ora domandarci come riuscire a calcolare efficientemente il risultato di una *top-k* 1:1 *join query* usando una *scoring function* S.

Nel caso particolare in cui $S \equiv MAX$ la soluzione è molto semplice [Fag96]: possiamo utilizzare l'algoritmo B_0 che riesce ad ottenere la soluzione semplicemente calcolando i migliori k oggetti da ogni risorsa. **NB** B_0 funziona solo in cui la *scoring function* è MAX, per tutte le altre funzioni di successo si richiede l'utilizzo di algoritmi più intelligenti e costosi.

Algoritmo B_0

- **Input**: *Ranked List* L_j $(j = 1, \ldots, m)$, un inter $k \geq 1$
- **Output**: i *top-k* oggetti in accordo alla funzione di successo MAX

Figura 29: Pseudo-codice Algoritmo B_0

```
Input: ranked lists Lj (j=1,...,m), integer k ≥ 1
Output: the top-k objects according to the MAX scoring function
 1. B := ∅;                                  // B is a main-memory buffer
 2. for j = 1 to m:
 3.     Obj(j) := ∅;                         // the set of objects "seen" on Lj
 4.     for i = 1 to k:                      // get the best k objects from each list
 5.         t := getNext_Lj();
 6.         Obj(j) := Obj(j) ∪ {t.OID};
 7.         if t.OID was not retrieved from other lists then: INSERT(B,t)  // adds t to the buffer
 8.                         else: join t with the entry in B having the same OID;
 9. for each object o ∈ Obj := ∪_j Obj(j):   // for each object with at least one partial score...
10.     MAX(o) := max_j{pj(o): pj(o) is defined};  // ...compute MAX using the available scores
11. return the k objects with maximum score;
12. end.
```

Predisponiamo in memoria un buffer B, e inizializziamo gli insiemi Obj(j), uno per ogni risorsa, vuoti. Per ogni input j, tento dunque di riempire l'insieme Obj(j) semplicemente usando il getNext (ovvero un next_NN) che restituisce il prossimo NN fino ad averne k: inoltre ogni volta che trovo un oggetto controllo se l'avevo già incontrato (e in caso aggiungo all'oggetto anche il risultato parziale appena trovato) oppure lo inserisco nel buffer B. Infine per ogni oggetto che ho visto almeno una volta, ovvero che ha almeno un risultato parziale, calcolo la *scoring function* MAX su di esso e infine restituisco i primi k oggetti.

L'algoritmo B_0 semplicemente adopera k accessi ordinati su ogni lista (k s.a. "round") e successivamente calcola il risultato senza il bisogno di ottenere i risultati parziali mancanti (cioè non è necessario alcun accesso casuale).

E vediamo l'algoritmo in azione su due esempi (Fig. 30).

Figura 30:

k = 2

OID	p1
o7	0.7
o3	0.65
o4	0.6
o2	0.5

OID	p2
o2	0.9
o3	0.6
o7	0.4
o4	0.2

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7

OID	S
o7	1.0
o2	0.9
o3	0.65

k = 3

MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
Da Gino	7.5
Tutti a tavola!	6.4
Le delizie del palato	5.5
Acqua in bocca	5.0

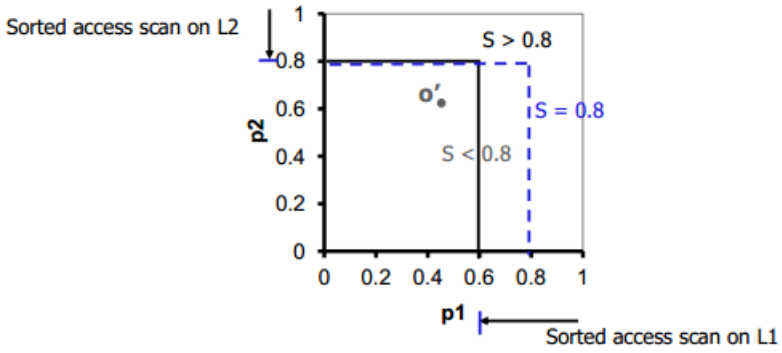
PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
Le delizie del palato	7.5
La tavernetta	7.0
Acqua in bocca	6.5
Tutti a tavola!	6.0

Name	S
Al vecchio mulino	9.2
Da Gino	9.0
La tavernetta	9.0
Il desco	8.5

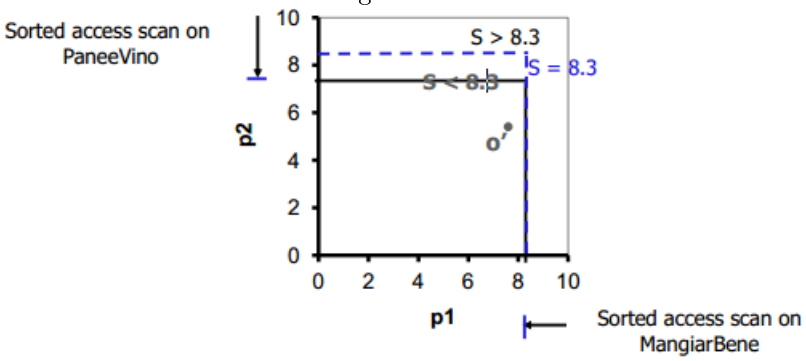
Vediamo effettivamente perchè l'algoritmo B_0 riesca a lavorare grazie ad un intuizione grafica. Prendiamo come esempio la seguente figura (Fig. 31). L'algoritmo B_0 adopera inizialmente un accesso ordinato secondo le varie preferenze, il che può essere visto attraverso le frecce presenti su L1 e L2, in cui viene restituito prima il valore più alto e man mano valori più bassi fino a trovare i primi k. Seguendo prima L1, si vede che abbiamo trovato k oggetti con score parziale su S1$\geq$ 0.6. Successivamente seguendo L2 si vede che si trovano k oggetti con score parziale su S2$\geq$ 0.8. Poichè la stiamo trattando la scoring function MAX, vuol dire che lo score globale di un oggetto nel risultato sarà sicuramente ≥ 0.8 poichè appunto ho trovato k oggetti su S2 che rispettano questo vincolo. Dunque, ci saranno almeno k oggetti o tali per cui $S(o) \geq 0.8$ e ciò accade perchè almeno una scansione ad accesso ordinato (nel nostro caso L2) si ferma dopo il ritrovamento, al k-esimo "round" e quindi alla k-esima tupla che ci serve, di un oggetto con punteggio locale uguale a 0.8. Un oggetto o' che non è stato trovato lungo alcuna scansione ad accesso ordinato (e dunque $o' \notin Obj$, ovvero l'insieme degli oggetti che hanno almeno uno score parziale) non può avere un punteggio globale migliore di 0.8 e quindi sicuramente non farà parte del risultato.

Figura 31:



Riprendendo l'esempio dei ristoranti, dopo 3 accessi ordinati vi è la garanzia che ci siano almeno 3 ristornati o con $S(o) \geq 8.3$ (Fig. 32). Un ristorante come o' non è stato trovato da alcuna scansione ad accesso ordinato (e dunque $o' \notin Obj$) non può avere un punteggio glovale maggiore di 8.3

Figura 32:



Diamo ora la prova che l'algoritmo B_0 sia *corretto*. Sia Res il risultato di B_0 ($Res \subseteq Obj$). Il bisogno di avere una dimostrazione formale della correttezza è motivata dalla seguente affermazione: se $o \in Obj - Res$, allora non vi è garanzia sulla correttezza di $S(o)$. Dobbiamo dunque dimostrare che ciò non altera il risultato. D'altra parte se $o \notin Obj$ allora abbiamo appena dimostrato che o non può essere migliore di qualsiasi oggett in Res.

Teorema: L'algoritmo B_0 determina correttamente i *top-k* oggetti e il loro punteggio globale.

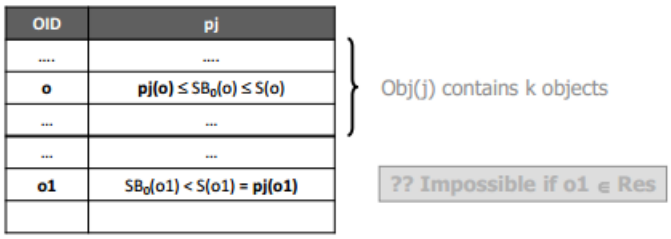
Dividiamo la dimostrazione in due parti: prima dimostriamo che se $o \in Res$ allora $S(o)$ è *corretto*, poi dimostreremo che se $o \in Obj - Res$ allora anche se il suo risultato globale non è corretto, l'algoritmo determina comunque correttamente i *top-k* oggetti.

Passo 1 Sia $SB_0(o)$ il punteggio globale così come lo calcola B_0 per un determinato oggetto $o \in Obj$. Per definizione di MAX allora $SB_0(o) \leq S(o)$. Sia $o1 \in Res$ e assumiamo per contraddizione che $SB_0(o1) < S(o1)$. Questo implica dire che esiste una L_j tale che $o1 \notin Obj(j)$ e che $S(o1) = p_j(o1)$. A sua volta questo implica che ci sono k oggetti $o \in Obj(j)$ tali che

$$SB_0(o1) < S(o1) = p_j(o1) \leq p_j(o) \leq SB_0(o) \leq S(o) \ \forall o \in Obj(j)$$

E dunque *o1* non può appartenere a *Res*. Si ha una contraddizione.

Figura 33:

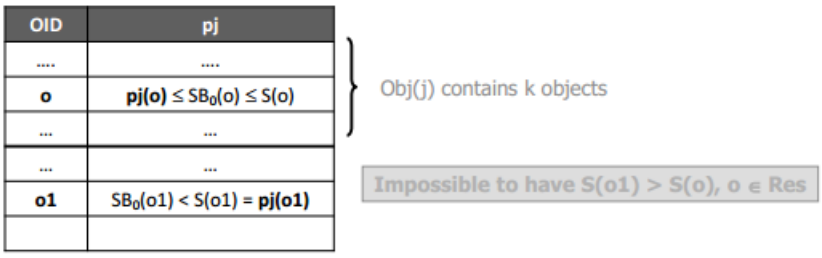


Passo 2 Consideriamo un oggetto $o1 \in Obj - Res$. Se $SB_0(o1) = S(o1)$ non vi è nulla da dimostrare. Altrimenti assumiamo che almeno un punteggio parziale di o1, $p_j(o1)$, non sia disponibile e che $SB_0(o1) < S(o1) = p_j(o1)$. Allora

$$SB_0(o1) < S(o1) = p_j(o1) \leq p_j(o) \leq SB_0(o) \leq S(o) \ \forall o \in Obj(j)$$

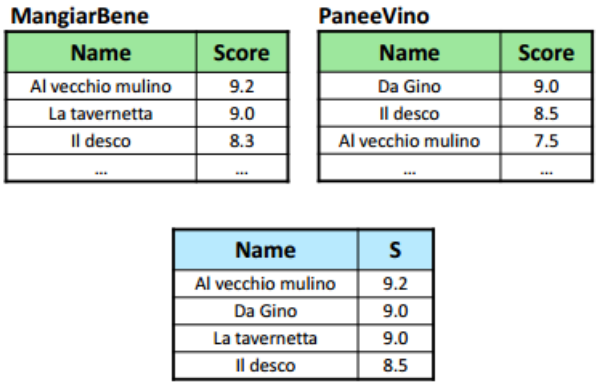
Poichè ogni oggetto in *Res* ha un punteggio globale almeno pari al peggior punteggio visto su L_j, segue che è impossibile avere $S(o1) > S(o)$ se $o \in Res$.

Figura 34:



Una domanda interessante che possiamo porci è se è possibile calcolare il risultato corretto tilizzando meno di k round di accessi ordinati. Si veda il seguente esempio (Fig. 35)

Figura 35:



MangiarBene

Name	Score
Al vecchio mulino	9.2
La tavernetta	9.0
Il desco	8.3
...	...

PaneeVino

Name	Score
Da Gino	9.0
Il desco	8.5
Al vecchio mulino	7.5
...	...

Name	S
Al vecchio mulino	9.2
Da Gino	9.0
La tavernetta	9.0
Il desco	8.5

Se:

- $k = 1$ serve necessariamente almeno un round: potremmo al più risparmiare un accesso ordinato se avessimo il *fetch* dell'oggetto con il maggior punteggio.
- $k = 2$ servono necessariamente due round: potremmo salvare un accesso ordinato se prima accedessimo alla relazione MangiarBene
- $k = 3$ potremmo fermarci dopo soli due round
- $k = 4$ potremmo fermarci al terzo round.

Esiste dunque una regola generale per riuscire a capire se possiamo usare meno di k round?

Per ogni lista L_j, assumiamo che $\underline{p_j}$ denoti il *peggior* punteggio visto finora sulla deermina lista. Assumiamo che *Res* denoti l'insieme dei top-k oggetti, ordinati secondo il loro attuale valore MAX; allora Res[k].score è il peggiore di essi. La seguente condizione di stop è sempre verificata dopo k *round* di accesso ordinato, ma potrebbe verificarsi anche prima.

Teorema: Un algoritmo per *top-k* 1:1 *join queries* che usi MAX come *scoring function* può fermarsi se e solo se $Res[k].score \geq max_j\{\underline{p_j}\}$.

Dimostrazione della *sufficienza*: Siccome ogni L_j è ordinata secondo valori non crescenti di p_j, nessun oggetto che non è stato incontrato su L_j può avere un punteggio parziale più alto di $\underline{p_j}$. Perciò nessun oggetto non incontrato può avere un valore più alto fra $max_j\{\underline{p_j}\}$. Segue che *Res* è corretta e che i punteggi degli oggetti in *Res* sono anch'essi corretti.

Dimostrazione della necessità: Assumiamo che l'algoritmo si fermi quando $Res[k].score < max_j\{\underline{p_j}\}$. Allora una lista L_j con un $\underline{p_j} > Res[k].score$ potrebbe contenere un oggetto o tale che $p_j(o) > Res[k].score$.

Si può pensare anche di creare un algoritmo che invece che minimizzare il numero di *round* minimizzi il numero degli accessi ordinati effettivi. Per far ciò

l'esecuzione degli accessi ordinati sulle liste non deve eseguire uno schema *round robin* e quindi la profondità su ogni lista potrebbe essere diverso e potremmo trovarci in una situazione come la seguente (Fig. 36)

Figura 36:

OID	p1
o9	0.7
...	...

OID	p2
o2	0.9
o3	0.6
o5	0.4
...	...

OID	p3
o1	0.8
o2	0.8
...	...

L'algoritmo che utilizza questo approccio, denominato **MaxOptimal**, utilizza gli stessi princi di k-NNOptimal, ovvero si prosegue l'accesso casuale sulla lista più "ottimistica". Ad ogni passo effettua un accesso ordinato sulla lista più promettente L_j*, per il quale cioè $\underline{p_j}*$ risulti massimo: $j* = argmax_j\{\underline{p_j}\}$. L'algoritmo mantiene in memoria solo i k migliori oggetti visti finora.

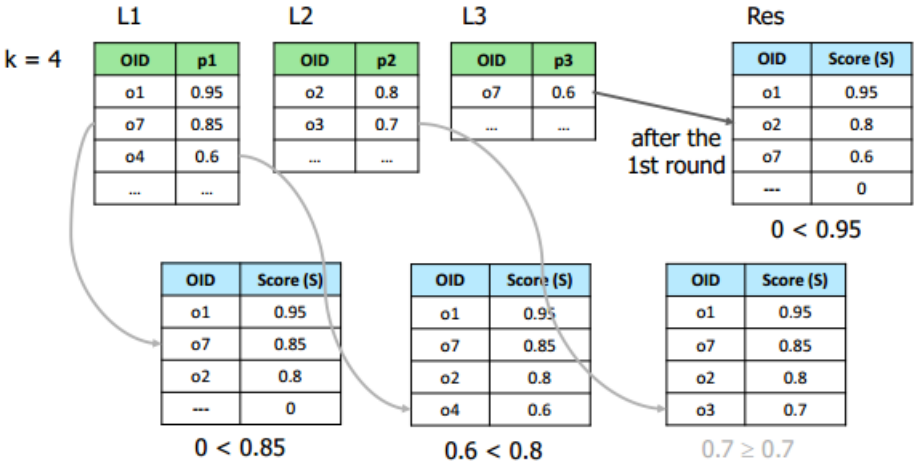
Figura 37: Pseudo-Codice MaxOptimal

```
Input: ranked lists Lj (j=1,...,m), integer k ≥ 1
Output: the top-k objects according to the MAX scoring function
 1. for i=1 to k: Res[i] := [null,0];          // entry of type: [OID,score] (+ other attr.'s as needed)
 2. for j = 1 to m: pj = 1;                    // the best possible score on Lj
 3. while Res[k].score < max_j{pj}:
 4.       j* = argmax_j{pj}; t := getNext_Lj*();          // get the next best object from list Lj*
 5.       if t.pj* > Res[k].score then:
 6.             if t.OID ∈ Res.OID then: update the entry in Res having the same OID
 7.                   else: {remove the object in Res[k]; insert [t.OID,t.pj*] in Res};
 8.  return Res;
 9.  end.
```

La condizione di stop è effettivamente quella vista prima, e l'algoritmo se trova un oggetto migliore fra quelle presenti in Res (controllando solo Res[k]), allora aggiorna Res con questa nuovo oggetto (o modificando un valore già esistente se l'oggetto faceva già parte di Res, o eliminando Res[k] e aggiungendo nella giusta posizione il nuovo oggetto).

Si veda il seguente esempio (Fig. 38).

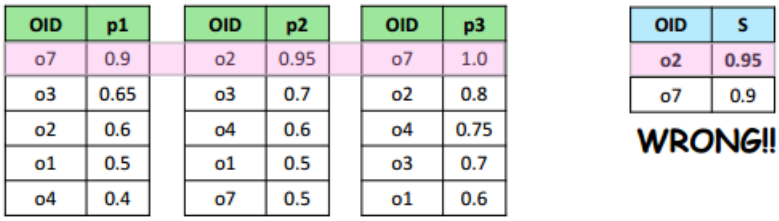
Figura 38:



La dimostrazione dell'ottimalità di MaxOptimal si basa sulla stessa di k-NNOptimal, ovvero se non controllassimo la lista con valore peggiore "più alto" per prima, saremmo comunque costretti a controllarla successivamente poichè nulla potrebbe escluderci che nella lista più promettente ci sia effettivamente un risultato migliore.

Ci si potrebbe chiedere se l'algoritmo B_0 funzioni anche utilizzando *scoring function* diverse rispetto al MAX. Vediamo subito un esempio (Fig. 39) in cui utilizziamo la *scoring function* MIN e con $k = 1$

Figura 39:



Come si vede il risultato è sbagliato, infatti controllando solo il primo NN su ogni input, si ottengono solo i valori per o2 e o7 (di cui di o7 prendiamo il valore minimo), e dell'insieme Obj restituiamo solo la prima tupla, ovvero o2. Il risultato è sbagliato in quanto controllando i restanti risultati parziali di o2 (ovvero anche 0.6 e 0.8) il valore della scoring function MIN applicato a o2 sarebbe $S(o2) = 0.6$ e non 0.95. Potrebbe venire il dubbio che il problema sia proprio questo, ovvero il non vedere tutti i risultati parziali degli oggetti.

Calcolando dunque tutti i risultati parziali degli oggetti in Obj ($Obj = \{o2, o7\}$) attraverso degli *accessi casuali* come,

$$getScore_{L_1}(o2),\ getScore_{L_3}(o2),\ getScore_{L_2}(o7)$$

si otterebbe dunque il seguente risultato (Fig. 40)

Figura 40:

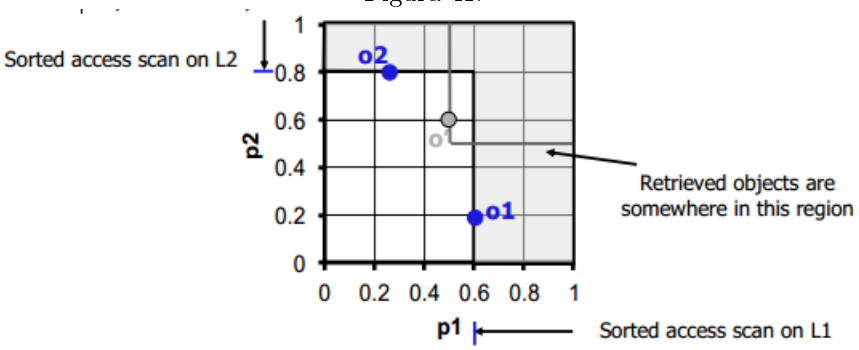
OID	S
o2	0.6
o7	0.5

STILL WRONG!!? ☹

il quale risulterebbe ancora sbagliato poichè in realtà, il primo NN della nostra query sarebbe in realtà o3 (si veda come i punteggi parziali di o3 sono $\{0.65, 0.7, 0.7\}$ di cui il risultato globale, ovvero il minimo, è 0.65 che risulta essere migliore del risultato globale di o2 ovvero 0.6).

Perchè dunque B_0 non funziona con *scoring function* diverse da MAX. Ancora una volta un intuizione grafica potrebbe aiutarci. Utilizziamo di nuovo come *scoring funcion* MIN e $k = 1$. Quando la scansione ad accesso ordinata termina, non abbiamo in realtà alcun *lower bound* sul risultato globale degli oggetti finora visti (il quale potrebbe essere anche uguale a 0). Un oggetto *o'* che non è stato incontrato su alcun accesso ordinato potrebbe essere in realtà il vero vincitore ovvero potrebbe dominare sugli oggetti trovati in Obj. Lo stesso problema si potrebbe avere anche in caso di funzione SUM. Si veda la figura (Fig. 41) per una maggiore comprensione.

Figura 41:



E dunque? Esiste soluzione?

2.3.2 Algoritmo FA

L'algoritmo FA (o A_0) [Fag96] può essere utilizzato per risolovere *top-k* 1:1 *join queries* che utilizzino una qualsiasi *scoring function* S monotona. Definizione di

scoring function monotona: data una *scoring function* S n-aria, essa è monotona se

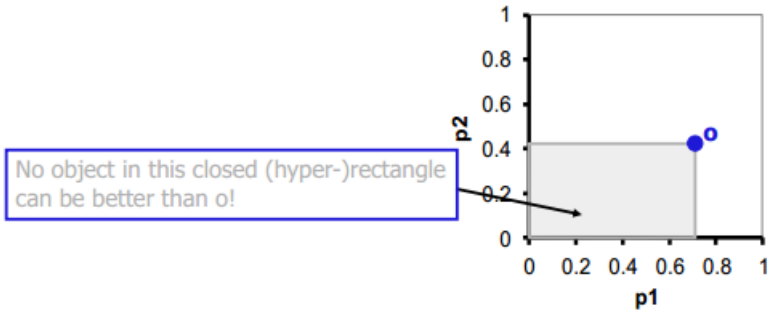
$$x_1 \leq y_1,\, x_2 \leq y_2,\, \ldots,\, x_n \leq y_n \Rightarrow S(x_1, x_2, \ldots, x_n) \leq S(y_1, y_2, \ldots, y_n)$$

ovvero una funzione di scoring è monotona se dati due punti x e y, se ogni cordinata di x risulta minore o uguale a ogni coordinata di y allora la scoring function calcolata su x è minore o uguale alla scoring function calcolata su y.

Perchè introdurre funzioni monotone? Perchè nella realtà è ciò che ci potrebbe aspettare, ovvero migliorando la bontà di una determinata caratteristica, ci si aspetta che la bontà globale non risulti peggiore, ovvero sarebbe strano che migliorando un criterio la valutazione globale cali.

L'algoritmo *FA* sfrutta la proprietà di monotonicità per sapere quando la scansione ad accesso ordinato può fermarsi. Si veda infatti la seguente figura (Fig. 42)

Figura 42:



Nessun oggetto racchiuso all'interno dell'iper-rettangolo definito dall'oggetto o, indipendentemente dalla scoring function utilizzata (importante è che sia monotona), potrà avere uno punteggio globale maggiore o uguale dell'oggetto o stesso.

Algoritmo FA

- **Input**: Ranked List L_j $(j = 1, \ldots, m)$, un intero $k \geq 1$, una *scoring function* S *monotona*
- **Output**: i *top-k* oggetti in accordo a S

Figura 43: Pseudo-Codice Algoritmo FA

```
Input: ranked lists Lj (j=1,...,m), integer k ≥ 1, monotone scoring function S
Output: the top-k objects according to S
// 1st phase: sorted accesses
 1. for j = 1 to m: Obj(j) := ∅; B := ∅; M := ∅;
 2. while |M| < k:
 4.     for j = 1 to m:
 5.         t := getNext_Lj(); Obj(j) := Obj(j) ∪ {t.OID};  // get the next best object from list Lj
 6.         if t.OID was not retrieved from other lists then: INSERT(B,t)
 7.                         else: join t with the entry in B having the same OID;
 8.     M := ∩_j Obj(j);      // the set of objects seen on all the m lists
// 2nd phase: random accesses
 9. for each object o ∈ Obj := Obj(j):  // for each object with at least one partial score...
        perform random accesses to retrieve the missing partial scores for o;
// 3rd phase: score computation
10. for each object o ∈ Obj: compute S(o);
11. return the k objects with maximum score;
12. end.
```

L'algoritmo accede (prima fase) alle varie liste in modalità *round robin* e colleziona in un insieme M gli oggetti che ho trovato in tutte le liste (ovvero gli oggetti per cui ho effettivamente tutti gli score parziali) e proseguo fintanto che l'insieme M non raggiunge cardinalità k. Per questi oggetti k conosco effettivamente tutti i loro valori, ma non è detto che siano loro il risultato della mia top-k join query: infatti, per ogni oggetto (seconda fase) che ho incontrato almeno una volta, non presente in M, cioè per cui ho almeno un risultato parziale (ma non tutti i risultati parziali), trovo i rimanenti risultati parziali attraverso degli accessi casuali. Infine (terza fase) per ogni oggetto o di cui ho appena trovato tutti i risultati parziali, ne calcolo la *scoring function* S e infine restituisco i k oggetti con il risultato migliore.

Notiamo che la condizione di stop sugli accessi ordinati, non dipende in alcun modo dalla *scoring function* S applicata, la quale entra in gioco solo al terzo passo: ovvero il numero degli accessi ordinati risulta sempre lo stesso al variare anche di S.

Vediamo come FA funziona applicandolo ad un esempio in cui $k = 1$ (Fig. 44).

Figura 44:

OID	p1
o7	0.9
o3	0.65
o2	0.6
o1	0.5
o4	0.4

OID	p2
o2	0.95
o3	0.7
o4	0.6
o1	0.5
o7	0.5

OID	p3
o7	1.0
o2	0.8
o4	0.75
o3	0.7
o1	0.6

Applicando FA ai dati mostrati, dopo aver effettuato accessi ordinati otteniamo $M = \{o2\}$ e $Obj = \{o2, o3, o4, o7\}$. Dopo aver effettuato gli accessi casuali necessari per avere i restanti risultati parziali otteniamo i seguenti risultati (Fig. 45), entrambi corretti.

Figura 45:

S ≡ MIN

OID	S
o3	**0.65**
o2	0.6
o7	0.5
o4	0.4

S ≡ SUM

OID	S
o7	**2.4**
o2	2.35
o3	2.05
o4	1.75

Perchè l'algoritmo FA è corretto?

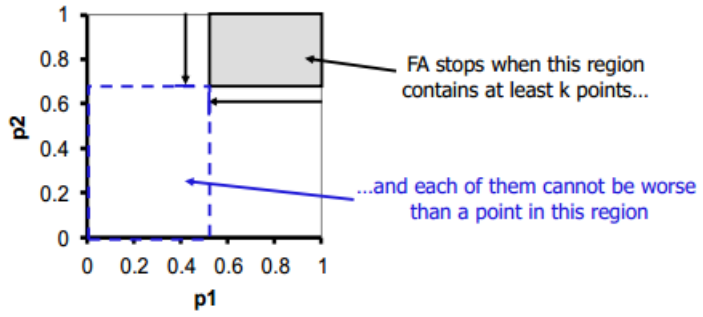
Vale il seguente teorema: L'algoritmo FA è corretto per qualsiasi *scoring function* S *monotona.*

Dimostrazione: Sia *Res* l'insieme degli oggetti ritornati dall'algoritmo FA. È sufficiente mostrare che se $o' \notin Obj$ (ovvero all'insieme degli oggetti di cui conosco almeno un risultato parziale, ovvero *o'* è un oggetto che non ho mai incontrato), allora *o'* non può essere migliore di qualsiasi oggetto $o \in Res$. Indichiamo con o un qualsiasi oggetto in *Res*. Allora, esiste almeno un oggetto $o'' \in M$ (potrebbe coincidere con o stesso) tale che $S(o) \geq S(o'')$, altrimenti o non sarebbe in *Res*. Siccome $o' \notin Obj$, per ogni L_j accade che $p_j(o') \leq p(o'')$ (poichè o' non è stato mai incontrato) e dall'assunzione di monotonicità della *scoring function* S si ha che $S(o') \leq S(o'')$. Ne segue che $S(o') \leq S(o)$.

L'algoritmo, dunque, individua un sottoinsieme di oggetti che potrebbero risultare fra il risultato della query e si ferma quando questo sotto insieme riesce ad escludere tutti gli oggetti, non ancora visti, per il quale il risultato non può essere effettivamente migliorato. Utilizzando la seguente figura (Fig. 46), possiamo intuire graficamente che FA si ferma quando la regione colorata in grigio contiene almeno k punti, e data la monotonicità di S accade che ognuno di questi punti non può essere peggiore di punto all'interno della regione tratteggiata. L'algoritmo si ferma quando la cardinalità di M è maggiore o uguale a K, ed essendo la funzione monotona, ogni oggetti presente nell'area grigia ha un

punteggio maggiore o uguale al vertice di intersezione con l'area tratteggiata, il quale avrà sicuramente un punteggio maggiore o uguale a tutti gli oggetti presenti nel rettangolo inesplorato (quello tratteggiato).

Figura 46:



Vediamone ora le *performance*. Quando le *sub-query* sono indipendenti (ovvero ogni *rank* locale è indipendente dall'altro) si può provare che il costo di FA (ovvero il numero di accessi ordinati e casuali) per un Database di N oggetti è, con grande probabilità, assumendo m costante

$$O\left(N^{\frac{m-1}{m}} \cdot k^{\frac{1}{m}}\right)$$

Siccome FA esegue gli accessi ordinati utilizzando una strategia *round robin*, su ogni lista si arriva a profondità diciamo X. Non facendo alcuna ipotesi sull'ordinamento degli oggetti in ogni lista e supponendo che gli ordinamenti siano indipendenti da una lista all'altra, ci interessa sapere la cardinalità di M, ovvero "la cardinalità dell'intersezione di m sotto-insiemi casuali di cardinalità X". La probabilità che un oggetto o_j appartenga a una lista L di N oggetti tra cui ne dobbiamo scegliere X è $\frac{X}{N}$. Essendo le probabilità indipendenti, la probabilità che un oggetto o_j appartanga a tutte le m liste è $\left(\frac{X}{N}\right)^m$. Se moltiplichiamo questo risultato per il numero di oggetti N otteniamo il numero medio di oggetti presenti nell'insieme M. Dovendo eguagliare la dimensione di M a k (poichè è la condizione di stop) ottengo

$$k = N \cdot \left(\frac{X}{N}\right)^m \rightarrow \frac{k}{N} = \left(\frac{X}{N}\right)^m \rightarrow \left(\frac{k}{N}\right)^{\frac{1}{m}} = \frac{X}{N}$$

ovvero

$$X = N \cdot \left(\frac{k}{N}\right)^{\frac{1}{m}} = \left(\frac{N^m \cdot k}{N}\right)^{\frac{1}{m}} = N^{\frac{m-1}{m}} \cdot k^{\frac{1}{m}}$$

Il risultato segue osservando che il numero di accessi ordinati è dato da $m \cdot X$ e il numero di accessi casuali è al più $(m \cdot X - m \cdot k) \cdot (m-1)$.

Il maggior inconveniente dell'algoritmo è che non sfrutta al massimo la specifica *scoring function* S. In particolare poichè S è usata solo al terzo passo

(quando vengono calcolati i punteggi globali), per un dato database il costo di accessi ordinati e casuali di FA è indipendente da S. In più, la memoria richiesta da FA potrebbe diventare proibitiva poichè FA richiede di bufferizzare tutti gli oggetti man mano che li incontra fra gli accessi ordinati. Alcuni miglioramenti sono possibili come ad esempio l'interleaving tra gli accessi casuali e il calcolo del punteggio che può far evitare alcuni accessi casuali stessi, ma il maggior miglioramento è possibile solo se si cambia la condizione di stop utilizzata: infatti FA utilizza una condizione di stop basata solo sui *ranking* locali degli oggetti.

Notiamo infine che se FA mantiene in buffer gli insiemi M e Obj, si potrebbe cambiare al volo la funzione S e si riuscirebbe ad ottenere immediatamente il risultato, proprio perchè la funzione S viene applicata solo all'ultimo passo.

2.3.3 Algoritmo TA

L'algoritmo TA (Threshold Algorithm) [FLN01, FLN03] differisce da FA in quanto alterna accessi ordinati e casuali e si basa su una regola di stop numerica. In particolare TA utilizza una soglia T la quale fa da *upper bound* a tutti i punteggi degli oggetti non ancora visitati. Ricordiamo un concetto fondamentale: un valore p_j per una lista L_j indica il peggior valore sulla lista finora incontrato. Nell'algoritmo viene introdotto il concetto di soglia T siffatta $T = S(p_1, p_2, \ldots, p_m)$ ovvero il valore della scoring function applicato al punto "virtuale" (non necessariamente appartanente al database) di coordinate $(p_1, p_2, \ldots, p_m)$ che ricordiamo essere gli ultimi valori incontrati su ogni lista L_j durante gli accessi ordinati.

Algoritmo TA

- **Input**: Ranked List L_j $(j = 1, \ldots, m)$, un intero $k \geq 1$, una *scoring function S monotona*
- **Output**: i *top-k* oggetti in accordo a S

Figura 47: Pseudo-Codice Algoritmo TA

```
Input: ranked lists Lj (j=1,...,m), integer k ≥ 1, monotone scoring function S
Output: the top-k objects according to S
 1. for i = 1 to k: Res[i] := [null,0];
 2. for j = 1 to m: pj := 1;
 3. while Res[k].score < T := S(p1,p2,...,pm):        // T is the "threshold"
 4.   for j = 1 to m:
 5.       t := getNext_Lj(); o := t.OID;
 6.       perform random accesses to retrieve the missing partial scores for o;
 7.       if S(o) := S(p1(o),...,pm(o)) > Res[k].score then:
 8.            {remove the object in Res[k]; insert [o,S(o)] in Res};
 9.   return Res;
10.   end.
```

L'algoritmo inizializza tutti i $\underline{p_j}$ a 1 (se normalizzato) e il risultato *Res* con valori di score 0: man mano che incontra un nuovo oggetto in una delle liste L_j calcola gli score parziali mancanti attraverso accessi casuali e controlla se il nuovo oggetto può modificare il risultato ovvero se il suo score è migliore dello score del k-esimo oggetto in *Res*. Quando lo score del k-esimo oggetto in *Res* risulta maggiore del valore di soglia T l'algoritmo può fermarsi, poichè tutti gli oggetti successivi saranno, non ancora visti, data la monotonicità della scoring function S saranno non migliori della soglia T e quindi non migliori degli oggetti in *Res*.

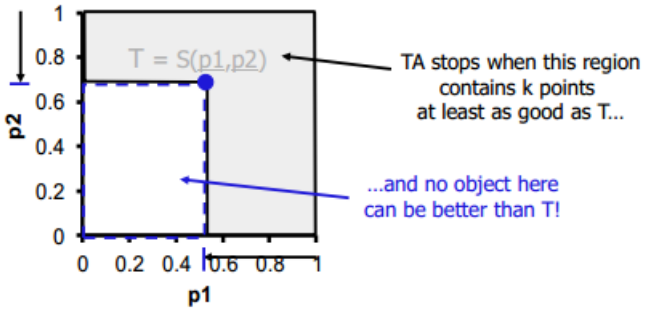
Perchè l'algoritmo TA è corretto?

Vale il seguente *teorema*: L'algoritmo TA è corretto per qualsiasi *scoring function* S *monotona*.

Dimostrazione: Consideriamo un oggetto *o'* che non è stato visto da nessuna scansione ad accesso ordinato. Dunque per ogni j si ha $p_j(o') \leq \underline{p_j}$. Data la monotonicità di S, questo implica che $S(o') \leq T$. Per definizione di *Res* per ogni oggetto $o \in Res$ si ha $S(o) \geq T$, cioè $S(o') \leq S(o)$.

Utilizzando la seguente figura (Fig. 48), possiamo intuire graficamente che TA si ferma quando la regione colorata in grigio contiene almeno k punti buoni almeno quanto T, e data la monotonicità di S accade che nessun punto nella regione tratteggiata può essere migliore di T.

Figura 48:



Vediamo di seguito come TA lavora attraverso due esempi (Fig. 49)

Figura 49:

- Let's take S ≡ MIN and k = 1

OID	p1	OID	p2	OID	p3
o7	0.9	o2	0.95	o7	1.0
o3	0.65	o3	0.7	o2	0.8
o2	0.6	o4	0.6	o4	0.75
o1	0.5	o1	0.5	o3	0.7
o4	0.4	o7	0.5	o1	0.6

S(o2) = 0.6. T = 0.9
S(o3) = 0.65. T = 0.65

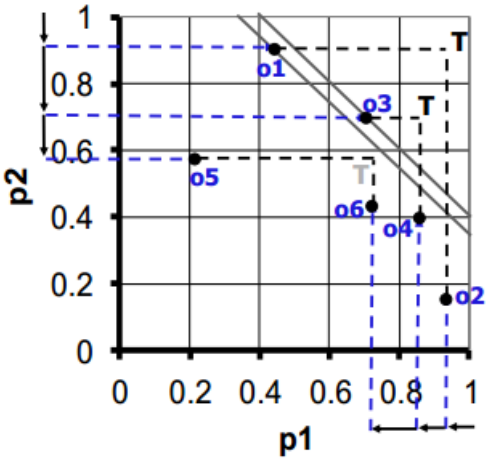
- Let's take S ≡ SUM and k = 2

OID	p1	OID	p2	OID	p3
o7	0.9	o2	0.95	o7	1.0
o3	0.65	o3	0.7	o2	0.8
o2	0.6	o4	0.6	o4	0.75
o1	0.5	o1	0.5	o3	0.7
o4	0.4	o7	0.5	o1	0.6

S(o7) = 2.4; S(o2) = 2.35. T = 2.85
S(o7) = 2.4; S(o2) = 2.35. T = 2.15

E vedamo di seguito anche un approccio geometrico all'algoritmo in cui si ha $S \equiv SUM$ e $k = 2$ (Fig.). Come si vede dalla figura, quando si arriva a trovare l'oggetto o1 ci si ferma. Si noti che $S(o3) > S(o1) \geq T$

Figura 50:



In generale TA funziona molto meglio di FA poichè riesce ad "adattarsi" alla specifica *scoring function* S. Sicuramente è più efficiente dell'algoritmo FA poichè la condizione di stop di FA implica quella di TA, ovvero se la condizione di stop di FA risulta vera allora ancora la condizione di stop di TA è vera.

Per riuscire a caratterizzare le performace di TA dobbiamo considerare il cosiddetto costo "middleware": $cost = SA \cdot c_{SA} + RA \cdot c_{RA}$ dove:

- SA (RA) è il numero totale di accessi ordinati (casuali)
- c_{SA} (c_{RA}) è il costo unitario (base) di ogni accesso ordinato (casuale).

e dunque il costo è dato dalla somma degli accessi ordinati e casuali moltiplicato per il costo unitario rispettivo.

Nelle impostazioni base si ha che $c_{SA} = c_{RA} (= 1$ per semplicità). In altri casi il costo base potrebbe differire, ad esempio per risorse web solitamente si ha $c_{RA} \gg c_{SA}$, fino al limite in cui $c_{RA} = \infty$ nei cui casi gli accessi casuali sono impossibili. In maniera opposta, alcune risorse potrebbero non supportare l'accesso ordinato nei cui casi si ha $c_{SA} = \infty$, ad esempio per istanze in cui non abbiamo un indice per eseguire p_j.

Un concetto fondamentale necessario per capire in che senso TA "funziona bene" è il concetto di *ottimalità d'istanza.*

Per introdurre il concetto facciamo un esempio: immaginiamo di avere un array ordinato di N elementi a cui vogliamo applicare la ricerca. Come ben noto le ricerca binaria ha costo nel caso medio e costo nel caso peggiore uguale a $\lg_2 N$, mentre la ricerca sequenziale nel caso medio ha costo $\frac{N}{2}$ e le caso peggiore N. Supponiamo di voler mettere a confronto i due algoritmi per ogni istanza del problema, ovvero quando si cerca il primo elemento, il secondo elemento, ..., e l'ultimo elemento. Indicando con la coppia (X,Y) i costi della ricerca binaria (X) e ricerca sequenziale (Y) vediamo cosa accade:

1. Si cerca il primo elemento. Costi $(\log_2 N, 1)$
2. Si cerca il secondo elemento. Costi $(\log_2 N, 2)$
3. ...
4. Si cerca l'N-1-esimo elemento. Costi $(\log_2 N, N-1)$
5. Si cerca l'N-esimo elemento. Costi $(\log_2 N, N)$

All'aumentare di N ($N \to \infty$) il rapporto fra $\log_2 N$ e 1 (i costi della prima istanza) tende ad infinito, ovvero la ricerca binaria non è *ottimale per le istanze,* ovvero esistono istanze in cui la ricerca sequenziale non è "migliore" della ricerca sequenziale, dove per "migliore" significa il cui rapporto non tende a infinito.

Data una classe di algoritmi **A** e una classe **D** di database (ovvero gli input degli algoritmi), un algoritmo $A' \in A$ è *ottimo per le istanze* su **A** e **D** per un dato costo misurato, se per ogni $B \in A$ e ogni $DB \in D$ si ha

$$cost(A', DB) = O(cost(B, DB))$$

Ciò è equivalente a dire che esistono due costanti c e c' tale che

$$cost(A', DB) \leq c \cdot cost(B, DB) + c'$$

Se A è *ottimale per le istante*, allora qualsiasi algoritmo può essere migliore di A per solo un fattore costante c, il quale viene chiamato il *rapporto ottimale* di A.

Osserviamo che *l'ottimalità d'istanza* è una nozione molto più forte di ottimalità di caso peggiore o migliore (ad esempio la ricerca binaria è ottima nel caso peggiore, ma non è ottima per le istanze).

TA è *ottimale per le istanze* su tutti i Database e su tutti gli algoritmi che non facciano "scelte casuali", e il suo *rapporto ottimale* è m quando $c_{RA} = 0$ (ovvero TA è ottimale per le istanze se consideriamo solo gli accessi ordinati). Un algoritmo A fa scelte casuali se effettua un accesso casuale all'oggetto o senza aver mai visto prima l'oggetto attraverso un accesso ordinato. Si noti che questo tipi di algoritmi "jolly" sono d'interesse solo dal punto di visto teorico.

Dimostrazione: Assumiamo che TA si fermi dopo aver eseguito X round di accessi ordinati (alla "profondità X"), cioè ha eseguito $m \cdot X$ accessi ordinati. Consideriamo un qualsiasi algoritmo *corretto* B e assumiamo che su ogni lista B esegue un numero di accessi ordinati sulla lista L_j in numero pari alla profondità, indicata con $Depth(B, j)$, che raggiunge sulla lista j. Indicando con $MaxDepth(B) = max_j\{Depth(B, j)\}$ ovvero la massima profondità che raggiunge su una lista, se $MaxDepth(B) \geq X$ abbiamo allora concluso, in quanto TA risulta migliore di B effettuando un numero minore di accessi ordinati. Se invece B effettua un numero strettamente minore di X di accessi ordinati su ogni lista L_j, cioè sulla lista L_j si raggiunge la profondità $Depth(B, j) < X$ e di conseguenza $MaxDepth(B) = max_j\{Depth(B, j)\} < X$ ovvero l'altezza massima di B è strettamente minore all'altezza massima trovata in TA bisogna dimostrare che B non è *corretto*. Consideriamo ora l'esecuzione di TA per $MaxDepth(B)$ round. Poichè i round includono tutti gli accessi ordinati (e i corrispondenti accessi casuali) fatti da B, e B è corretto per ipotesi, allora TA potrebbe fermarsi all'altezza $MaxDepth(B) < X$, una contradizione: ovvero siccome l'esecuzione di TA per $MaxDepth(B)$ effettua tanti accessi almeno quanti ne ha fatti l'algoritmo B (poichè B su ogni lista potrebbe avere profondità diverse, invece TA ogni lista ha stessa profondità poichè utilizza una tecnica *round robin*) allora TA potrebbe fermarsi a $MaxDepth(B)$ invece che scendere in profondità fino a X, poichè se B è corretto per ipotesi, e sappiamo che TA lo è, allora il risultato di B deve coincidere con quello di TA, ma TA si è dovuta fermare a profondità X per riuscire a trovare il risultato (non poteva fermarsi prima, altrimenti lo avrebbe fatto), dunque B è un algoritmo non corretto. Ne segue che ogni algoritmo corretto B deve avere $MaxDepth(B) \geq X$ e quindi il suo costo deve essere maggiore o uguale a X, cioè per ogni Database si ha

$$cost(TA, DB) = m \cdot X \leq m \cdot cost(B, DB)$$

Quando $c_{RA} > 0$ (caso reale, ricordiamo che finora avevamo assunto $c_{RA} = 0$ per anche la dimostrazione di ottimalità) il costo d'arresto di TA è al massimo

$$cost(TA, DB) = m \cdot X \cdot c_{SA} + m \cdot X \cdot (m - 1) \cdot c_{RA}$$

poichè nel caso peggiore TA ritrova $m \cdot X$ oggetti distinti, e per ognuon di essi esegue $m - 1$ acessi casuali.

Come abbiamo visto, ogni altro algoritmo B dovrà pagare almeno un costo di

$$cost(B, DB) \geq X \cdot c_{SA}$$

Dunque, il *rapporto ottimale* divinete ora

$$\frac{m \cdot X \cdot c_{SA} + m \cdot (m-1) \cdot X \cdot c_{RA}}{X \cdot c_{SA}} = m + m \cdot (m-1) \cdot \frac{c_{RA}}{c_{SA}}$$

il quale diventa abbastanza pessimo quando $c_{RA} > (\gg) c_{SA}$ poichè il costo degli accessi casuali prevale: ad esempio con $\frac{c_{RA}}{c_{SA}} = 10$ e $m = 3$ il *rapporto ottimale* è 63.

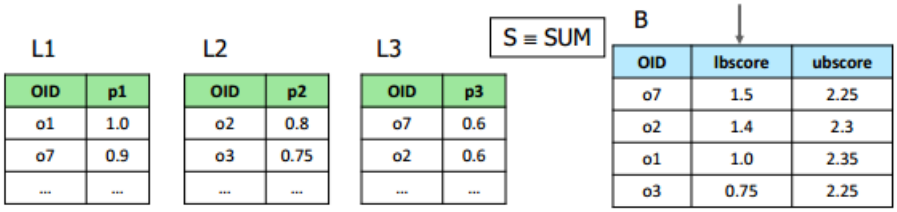
2.3.4 Algoritmo NRA

L'algoritmo *NRA* (No Random Access Algorithm) [FLN01, FLN03] è un algoritmo applicabile quando non è possibile eseguire accessi casuali (o se il loro costo è proibitivo). L'algoritmo ritorna i *top-k* oggetti, ma il loro punteggio potrebbe essere sbagliato, questo per limitare il costo dell'algoritmo stesso. L'idea dietro NRA è di mantenere per ogni oggetto o incontrato durante un accesso ordinato, un *lower bound* (*lbscore*) $S^-(o)$ e un *upper bound* (*upscore*) $S^+(o)$ sul loro punteggio.

- $S^-(o)$ (da ora in poi $S-(o)$ per semplicità) è ottenuto imponendo $p_j(o) = 0$ (o il minimo valore possibile di p_j) se l'oggetto o non è stato visto sull'input L_j
- $S^+(o)$ (da ora in poi $S+(o)$ per semplicità) è ottenuto imponendo $p_j(o) = \underline{p_j}$ se l'oggetto o non è stato visto sull'input L_j

NRA usa un buffer B di capacità illimitata, il quale è mantenuto ordinato secondo valori *decrescenti* di *lbscore*. Si veda la figura (Fig. 51), in cui $S \equiv SUM$ per una maggiore comprensione.

Figura 51:



Vediamo dunque l'algoritmo NRA. Chiamiamo Res le prime k posizioni del buffer B, cioè $Res = \{B[1], B[2], \ldots, B[k]\}$. L'idea dell'algoritmo è di fermarsi quando nessun oggetto $o' \notin Res$ può essere "migliore" di qualsiasi oggetto appartenente a Res, cioè quando

$$S^+(o') \leq S^-(o) \, \forall o' \notin Res, o \in Res$$

ovvero quando l'upper bound di un oggetto non presente nel risultato risulta minore o uguale al lower bound del k-esimo oggetto in Res (poichè Res è ordinata per lower bound non crescenti). L'idea, in pratica, è di fermare l'algoritmo quando nessun oggetto che non è tra i primi k-oggetti (quindi quelli appartenenti a Res) possa modificarmi il risultato Res.

Per verificare questa condizione è sufficiente considerare il massimo valore di $S + (o')$ fra gli oggetti in $B - Res$ (gli oggetti che sono nel buffer ma non in Res) e la soglia, la quale fornisce un limite superiore agli oggetti non ancora incontrati.

Algoritmo NRA

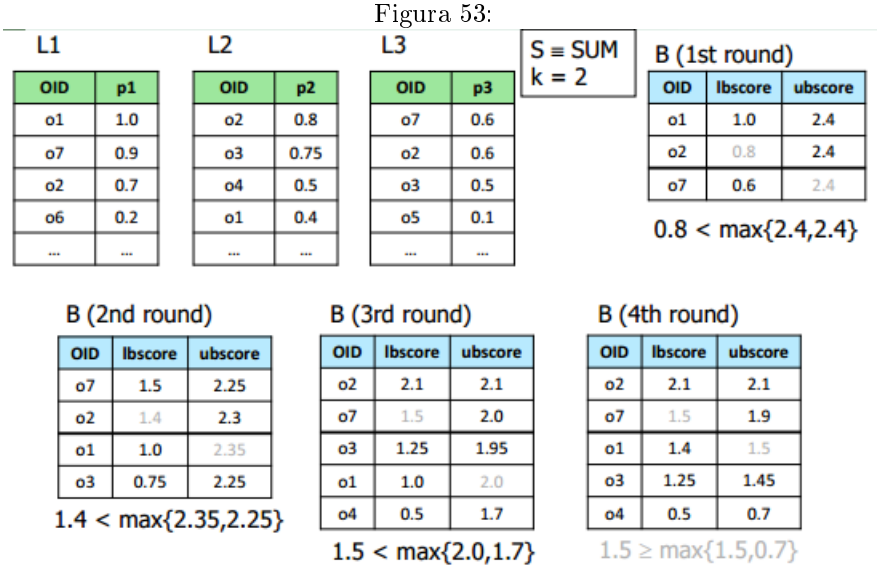
- **Input**: Ranked List L_j $(j = 1, \ldots, m)$, un intero $k \geq 1$, una *function score S monotona*
- **Output**: I *top-k* oggetti in accordo a S

Figura 52: Pseudo-Codice Algoritmo NRA

```
Input: ranked lists Lj (j=1,...,m), integer k ≥ 1, monotone scoring function S
Output: the top-k objects according to S
 1. B := ∅; // entry of type: [OID,lbscore,ubscore]; B is ordered by decreasing lbscore values
 2. for j = 1 to m: pj := 1;
 3.  while B[k].lbscore < max{max{B[i].ubscore, i > k},S(p1,p2,...,pm)}:
 4.   for j = 1 to m:
 5.       t := getNext_Lj(); o := t.OID; insert [o,S⁻(o),S⁺(o)] in B;
 6. return {B[1],...,B[k]};
 7. end.
```

E vediamo subito due un esempio (Fig. 53) in cui $S \equiv SUM$ e $k = 2$

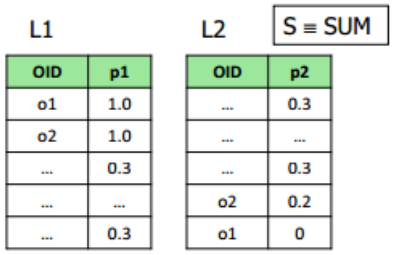
Figura 53:



Un interessante osservazione riguardo NRA è che il suo costo non cresce monotonicamente con k, cioè potrebbe richiedere un costo inferiore per i *top-k* oggetti rispetto che ai *top-(k-1)*.

Si veda il seguente esempio (Fig. 54) in cui $S \equiv SUM$.

Figura 54:



In caso si abbia $k = 1$ il vincitore è o2 ($S(o2) = 1.2$), poichè il punteggio di o1 è $S(o1) = 1.0$ e il punteggio di tutti gli altri oggetti è 0.6. In questo caso NRA deve raggiungere la profondità $N - 1$ per potersi fermare. Nel caso invece di $k = 2$ per trovare i *top-2* oggetti che sono o1 e o2 sono sufficienti solo 3 rounds.

Per quanto riguarda l'*ottimalità per le istanze*, si può dimostrare che *NRA* è *ottimo per le istanze* su tutti i Database e su tutti gli algoritmi che non eseguono accessi casuali, e il suo *rapporto di ottimalità* è m (cioè se NRA si ferma a profondità X, allora qualsiasi altro algoritmo B deve leggere X oggetti da almeno una lista).

Se risulta necessario avere i punteggi *esatti* dei top-k oggetti, allora l'algoritmo, che viene chiamato *NRA**, lavora come di seguito:

1. Esegui l'algoritmo *NRA* fino a che i *top-k* non sono determinati (ovvero *Res* è stabile)

2. Esegui tanti accessi ordinati quanti necessari fino a trovare tutti i punteggi parziali per gli oggetti che sono stati trovati in *Res*

Si noti che l'algoritmo NRA* esegue almeno tanti round di accessi ordinati quanti l'algoritmo FA (possibilmente anche di più), infatti FA si ferma a profondità X dopo aver visto k oggetti in tutte le liste (gli oggetti che stanno in M), mentre NRA* non può fermarsi a profondità $< X$ poichè per almeno un oggetto in *Res* non si conosce il punteggio esatto, inoltre non vi è alcuna garanzia che gli oggetti in M siano anche in *Res*.

Si può facilmente dimostrare che NRA* è *ottimale per le istanze* su tutti i Database e su tutti gli algoritmi che calcolano il punteggio esatto e non eseguono accessi casuali, e il suo *rapporto di ottimalità* risulta m.

2.3.5 Algoritmo CA

L'algoritmo *CA* (Combine Algoithm) [FLN01, FLN03] è un tentativo di ridurre l'influenza negativa degli alti costi degli accessi casuali. L'idea dietro CA è semplice: piuttosto che eseguire accessi casuali ad ogni round, vengono esguiti ogni $\frac{c_{RA}}{c_{SA}}$ round (più precisamente ogni $\left\lfloor \frac{c_{RA}}{c_{SA}} \right\rfloor$). In pratica CA si comporta come NRA, e come NRA mantiene *lower* e *upper bound* sui punteggi degli oggetti, ma ogni $\left\lfloor \frac{c_{RA}}{c_{SA}} \right\rfloor$ round di accessi ordinati esegue un accesso casuale. Il punto chiave è capire per quali oggetti bisogna richiedere un accesso casuale. Non sorprende che questi oggetti siano gli oggetti o per i quali mancano alcuni risultati parziali e per i quali $S^+(o)$ è massima (poichè per il solito concetto, se lo facessi per un altro oggetto questo non escluderebbe il dover eseguire successivamente il calcolo per l'oggetto con upper bound massimo).

Comparando CA all'algoritmo TA, CA esegui più accessi ordinati ma meno accessi casuali. Si può provare che CA è *ottimale per le istanze* con un *rapporto di ottimalità* indipendente da $\frac{c_{RA}}{c_{SA}}$ ma solo se

1. su ogni lista i punteggi sono tutti distinti (ovvero non ci sono casi di parità su p_j)

2. $S \equiv MIN$ o S è *strettamente* monotona per ogni argomento: ogni volta che p_j è incrementato e gli altri argomenti rimangono invariati, allora anche il valore di S si incrementa (ad esempio per scoring function SUM questo accade).

2.3.6 Riassunto

Si veda la seguente tabella (Fig. 55) per avere una visione generale di tutti gli algoritmi che si possono utilizzare per risolvere una *top-k join* 1:1 *query*.

Figura 55:

Algorithm	scoring f. S	Data access	Notes
B_0	MAX	sorted	instance-optimal
MaxOptimal	MAX	sorted	instance-optimal
FA	monotone	sorted and random	cost independent of S
TA	monotone	sorted and random	instance-optimal
NRA	monotone	sorted	instance-optimal, wrong scores
NRA*	monotone	sorted	instance-optimal, exact scores
CA	monotone	sorted and random	instance-optimal, optimality ratio independent of c_{RA}/c_{SA} in some cases

Esistono molti altri algoritmi per calcolare una *top-k join* 1:1 *query*, ed ognuno di questi si basa sull'assunzione che la *scoring function* S sia *monotona.* Il caso più semplice si ha quando $S \equiv MAX$: l'algoritmo base B_0 di Fagin può esere migliorato (*MaxOptimal*) sfruttando principi simili a quelli applicati per la ricerca k-NN. L'algoritmo FA è l'unico la cui condizione di stop considera semplicemente il *ranking* locale degli oggetti piuttosto che il loro punteggio parziale. La condizione di stop di TA si basa su una soglia T la quale fornisce un *upper bound* ai punteggi di tutti gli oggetti non incontrati. TA è *ottimale per le istanze* anche se il suo *rapporto di ottimalità* dipende da $\frac{c_{RA}}{c_{SA}}$, ovvero il rapporto tra il costo di accesso casuale e il costo di accesso ordinato. L'algoritmo NRA invece non esegue alcun accesso casuale. Infine l'algoritmo CA è una combinazione di TA e NRA, risulta essere *ottimale per le istanze* con un *rapporto di ottimalità* indipendente da $\frac{c_{RA}}{c_{SA}}$ solo per un sotto insieme di *scoring function* S e un sottoinsieme di Database.

2.4 Top-k Join Queries

Il caso generale di una *top-k join queries* in cui abbiamo $n > 1$ relazioni di input e una *scoring function* S definita sul risultato di join può essere espressa dalla seguente semantica

```
SELECT <some attributes>
FROM R1, R2, ..., Rn
WHERE <join and local conditions>
ORDER BY S(p1,p2,...,pm) [Desc]
STOP AFTER k
```

dove p1,p2,...,pm sono i criteri di *score* (le nostre "preferenze").

Consideriamo ora il caso generale in cui il *join* avviene *molti a molti*, ovvero un join M:N, come ad esempio

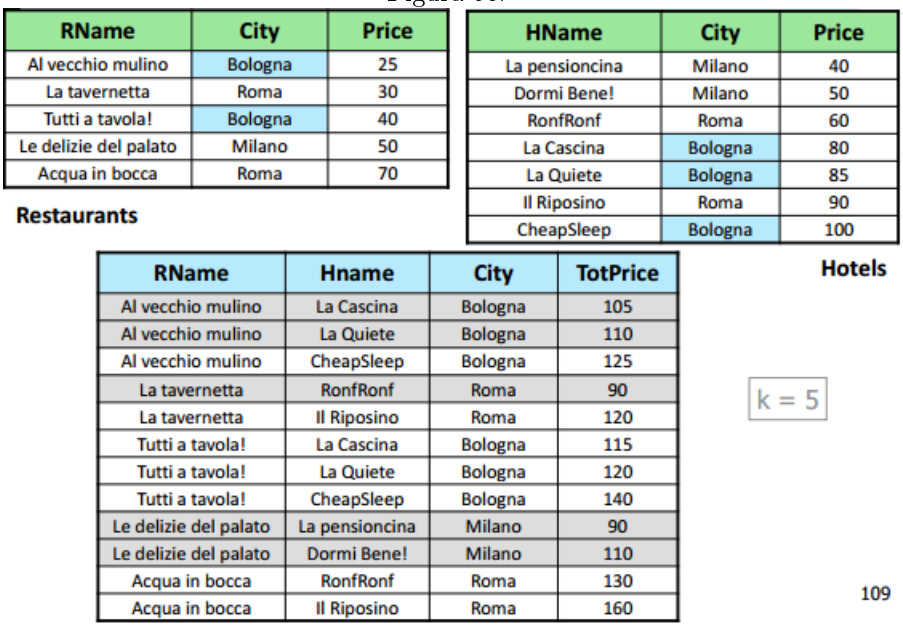
```
SELECT *
FROM RESTAURANTS AS R, HOTELS AS H
```

```
WHERE R.City=H.City
        AND R.Nation='Italy'
        AND H.Nation='Italy'
ORDER BY R.Price+H.Price
STOP AFTER 2
```

Notiamo che il join non avviene più sugli attributi chiave delle relazioni.

Vediamo in figura (Fig. 56) l'esempio della query appena scritta applicata a dei dati, in cui $k = 5$.

Figura 56:



Nei casi più favorevoli possiamo accedere a tutti gli input utilizzando *indici* sugli attributi di join (ad esempio su *city*).

In questo caso l'algoritmo risultante è molto simile all'algoritmo TA:

1. Esegui un round di accessi ordinati
2. Per ogni tupla recuperata:
   (a) usando l'indice sugli attributi di join, effettua un accesso casuale per ritrovare tutti i match sui restanti input
   (b) mantieni solo i migliori k risultati tra le combinazioni di join
   (c) se uno queste nuove combinazioni di join è fra le combinazioni *top-k* viste finora allora tienila, altrimenti scartala

3. Ripeti il passo 1 fintanto che la condizione di soglia non è verificata (cioè quando nessuna combinazione di join non ancora vista può essere migliore dell'attuale risultato *top-k*). La soglia T viene identificata alla stessa modo dell'algoritmo TA.

Vediamo dunque un esempio in cui l'algoritmo viene applicato (Fig. 57).

Figura 57:

RName	City	Price
Al vecchio mulino	Bologna	25
La tavernetta	Roma	30
Tutti a tavola!	Bologna	40
Le delizie del palato	Milano	50
Acqua in bocca	Roma	70

HName	City	Price
La pensioncina	Milano	40
Dormi Bene!	Milano	50
RonfRonf	Roma	60
La Cascina	Bologna	80
La Quiete	Bologna	85
Il Riposino	Roma	90
CheapSleep	Bologna	100

RName	Hname	City	TotPrice
Al vecchio mulino	La Cascina	Bologna	105
Al vecchio mulino	La Quiete	Bologna	110

k = 2

RName	Hname	City	TotPrice
Le delizie del palato	La pensioncina	Milano	90
Al vecchio mulino	La Cascina	Bologna	105

RName	Hname	City	TotPrice
Le delizie del palato	La pensioncina	Milano	90
La tavernetta	RonfRonf	Roma	90

2.4.1 Algoritmo Rank-Join

In caso invece non siano presenti indici sugli attributi di join, l'unica alternativa è di calcolare il risultato utilizzando solo gli accessi ordinati, sulla scia dell'algoritmo di NRA*.

L'algoritmo base per questo scenario è chiamato *Rank-Join* [IAE03], e la sua descrizione richiedere di aggiungere alcune notazioni:

- Per ogni ranked list L_j, indichiamo con pj^{max}il primo (più alto) punteggio visto su L_j
- Indichiamo con T il massimo fra i seguenti m valori

  $S(\underline{p1}, p2^{max}, \ldots, pm^{max})$, $S(p1^{max}, \underline{p2}, \ldots, pm^{max})$, $\ldots$, $S(p1^{max}, p2^{max}, \ldots, \underline{pm})$

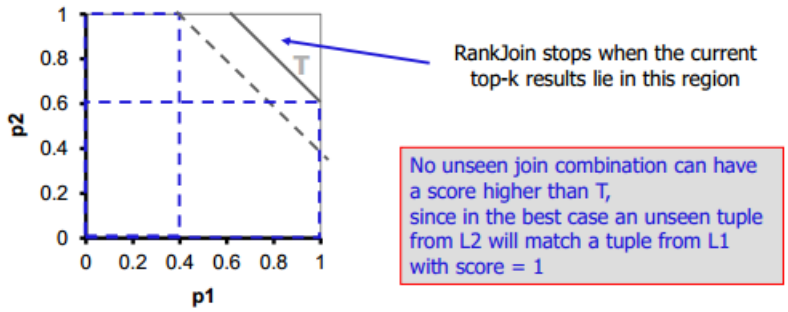
  il quale viene anche chiamato il *corner bound*. Si noti il significato di un singolo termine: si calcola per ogni coordinata la scoring function utilizzando il valore più basso incontrato sulla coordinata insieme ai valori massimi di tutte le altre.

- Indichiamo con j una generica combinazione di join,cioè $j = (t1, t2, \ldots, tm)$

L'osservazione ovvia è che ci si può fermare quando ci sono k combinazioni di join j tali che $S(j) \geq T$. Gli accessi ordinati possono essere eseguiti seguendo una strategia *round-robin* oppure accedendo alla lista per cui si ha $\underline{pj}$ massimo.

Tentiamo di visualizzare il *corner bound* in uno spazio grafico. Per semplicità assumiamo che $p1^{max} = p2^{max} = 1$ e che $S \equiv SUM$. Nella figura (Fig. 58) si ha che $\underline{p1} = 0.4$ e $\underline{p2} = 0.6$ e dunque $T = max\{(0.4 + 1), (1 + 0.6)\} = 1.6$

Figura 58:



Il *Rank-join* si ferma quando l'attuale risultato *top-k* giace sulla regione delimitata da T. Nessuna combinazione di join può avere un punteggio migliore di T, poichè nel miglior caso una tupla non incontrata su L2 farà il match con una tupla di L1 con punteggio uguale a 1.

Quando $m = 2$ (il numero di predicati di join) si può provare che il *Rank-join* è *ottimo per le istanze*, cioè il *corner bound* è stretto (potrebbe esserci una combinazione di join j tale che $S(j) = T$). D'altro canto quando $m > 2$ l'analisi diventa molto più complessa, e l'*ottimalità per le istanze* è garantita solo se le condizioni di join sono considerate come una "black box", cioè gli argomenti per dimostare l'*ottimalità di istanze* non considerano le vere condizioni di join. Quando, invece, i predicati di join sono presi effettivamente in considerazione, nessun algoritmo basato sul *corner bound* è *ottimale per le istanze* [SP08]. Lo stesso risultato negativo lo si ottiene anche se gli input n sono solamente due ma in cui almeno uno degli input ha 2 risultati parziali per ogni tupla (ovvero $n < m$).

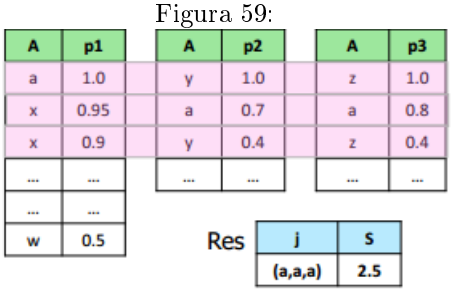
Vediamo un esempio di non ottimalità del *Rank-Join*.

Prendiamo la seguente query

```
SELECT *
FROM R1, R2, R3
WHERE R1.A=R2.A AND R1.A=R3.A
ORDER BY p1+p2+p3 DESC
STOP AFTER 1
```

in cui abbiamo $S \equiv SUM$. Il risultato è dato dalla seguente figura (Fig. 59)

Figura 59:



Dopo i primi 3 round di accessi ordinati, il *corner bound* prodotto $T = 0.9 + 1 + 1 = 2.9$. Tutta via, le tuple non incontrate da L1 possono portare a una combinazione di join j con $S(j) > 2.5$, cioè potremmo fermarci qui, con solo 9 accessi ordinati.

D'altro canto, su questa istanza il *Rank-Join* potrebbe incorrere in un costo arbitrariamente alto, dipende da come sono distribuiti i punteggi. Si noti che il numero di tuple in L1 fra $(x, 0.9)$ e $(w, 0.5)$ non è limitato.

Oltre che mostrare i limiti e le mancanze del *Rank-Join*, [SP08] ha inoltre introdotto uno schema di *bounding* più stretto che garantisce l'*ottimalità per le istanze*. Il metodo, qui non descritto, ha come maggiori caratteristiche di avere *complessità polinomiale* sui dati, ovvero di funzionare in tempo polinomiale sul numero delle tuple trovate dalle *ranked list*. È però un problema NP-hard se si considera la complessità della query, ovvero il suo tempo di calcolo cresce esponenzialmente con il numero di input. È interessante notare che si basa sul concetto di "dominazione" delle tuple, il quale è il cuore principale delle *query skyline*.

2.5 Ranking come prima classe

Una sfida interessante riguardante le *top-k queries* è come incorporare le tecniche basate su ranking in un *dbms* relazionale. Ciò è necessario per migliorare le performance per una *top-k query* del tutto generale. Il progetto RankDB ha fornito contributi fondamentali sulla soluzione a questo problema, proponendo un sistema prototipo, chiamato *RankSQL* [LCl+05], in cui il *ranking* è trattato come oggetto di "prima classe". Illustreremo i concetti base di RankSQL, in particolare:

- I requisiti di *splitting* e *interleaving*
- Il concetto di *rank-relation* e il *ranking principle*
- L'algebra di rank per *rank-relation*

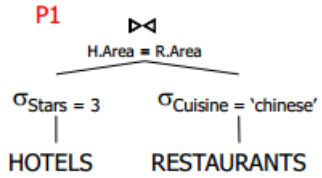
2.5.1 Splitting e Interleaving

Consideriamo la seguente query SQL

```
SELECT *
FROM RESTAURANTS AS R, HOTELS AS H
WHERE R.Area=H.Area
        AND H.stars=3
        AND R.Cuisine='chinese'
```

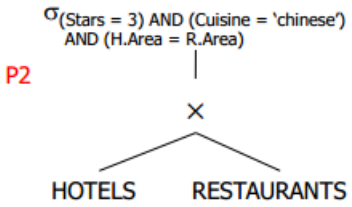
Un possibile piano di accesso P1 per la query è il seguente (Fig. 60)

Figura 60:



P1 è un piano d'accesso decisamente migliore del seguente "monolitico" piano P2 in cui $\times$ denota il prodotto cartesiano (Fig. 61)

Figura 61:



Ecco come funziona dunque lo *splitting* e l'*interleaving*: possiamo traformare P2 in P1 semplicemente effettuando lo *splitting* dei predicato booleani in *join* e *selezioni* e farne successivamente l'*interleaving*. Lo stesso dovrebbe essere fatto per le funzioni di *ranking*.

2.5.2 Rank-Relations e Ranking Principle

Per rendere un *ottimizzatore* di query "conscio" del *ranking* risulta necessario introdurre il concetto di *rank-relation*, cioè una relazione composta da tuple con punteggio. La definizione è strettamente legata al *ranking principle* il quale formalizza l'ormai ben noto fatto che la tupla più promettente dovrebbe essere processata per prima.

Diamo la definizione di **Rank-Relation**: data una relazione R e una *scoring function monotona* $S(p1, p2, \ldots, pm)$, la *rank-relation* R_P, con $P \subseteq \{p1, p2, \ldots, pm\}$, è una relazione R aumentata con un *ranking* così come segue:

- Il *punteggio* di una tupla t in R_P è il punteggio massimo possibile, denotato con $S_P^+(t)$, in accordo alla funzione S, dove $S_P^+(t)$ è calcolata sostituendo in S il valore di $p_j(t)$ se $p_j \in P$, 1 altrimenti (o il valore massimo possibile per p_j)
- Le tuple in R_P sono "ordinate" per valori decrescenti di $S_P^+(t)$ (*ranking principle*)

Si noti che $S_P^+(t) = S(t)$ quando $P = \{p1, p2, \ldots, pm\}$, e si noti che $R_\emptyset \equiv R$.

2.5.3 L'algebra RankSQL

L'algebra RankSQL estende la semantica di RA alle *rank-relations*. Essa introduce un nuove operatore di ranking μ, il quale applica alla *rank-relation* R_P una "preferenza" non ancora calcolata p (ovvero $p \notin P$), ottentendo la nuova *rank-relation* $R_{P\cup\{p\}}$.

L'operatore μ è la base per lo *split* della *scoring function* $S(p1, p2, \ldots, pm)$ poichè

$$R_{\{p1,p2,\ldots,pm\}} = \mu_{p1}(\mu_{p2}(\ldots(\mu_{pm}(R))))$$

ovvero il *ranking* finale può essere ottenuto applicando le "preferenze" una alla volta. Qualsiasi ordine di valutazione è ammissibile poichè $\mu_{p1}(\mu_{p2}(R)) = \mu_{p2}(\mu_{p1}(R))$.

L'*interleaving* con le selezioni e i join è ora possibile, ad esempio, per la selezione vale

$$\mu_p(\sigma_c(R_P)) = \sigma_c(\mu_p(R_P))$$

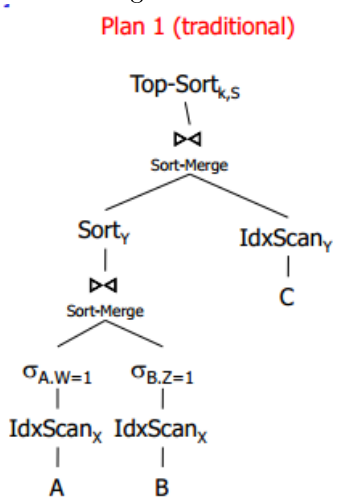
A causa del *ranking principle*, un operatore μ_p può ritornare una tupla t se e solo se è garantito che non vi sia alcuna tupla t' tale che $S_{P\cup\{p\}}^+(t') > S_{P\cup\{p\}}^+(t)$. Ciò può essere ottenuto appena μ_p calcola una tupla t'' tale che $S_{P\cup\{p\}}^+(t) \geq S_P^+(t'')$. La scansione per *indice* (e anche quella sequenziale) sono anch'essi trattati come operatori, poichè potrebbero essere usati al posto di μ per ordinare le tuple in accordo a una "prefenza" p.

Facciamo un esempio utilizzando la seguente query

```
SELECT *
FROM A,B,C
WHERE A.X=B.X AND B.Y=C.Y
        AND A.W=1 AND B.Z=1
ORDER BY A.p1+A.p2+B.p3+B.p4+C.p5 DESC
STOP AFTER k
```

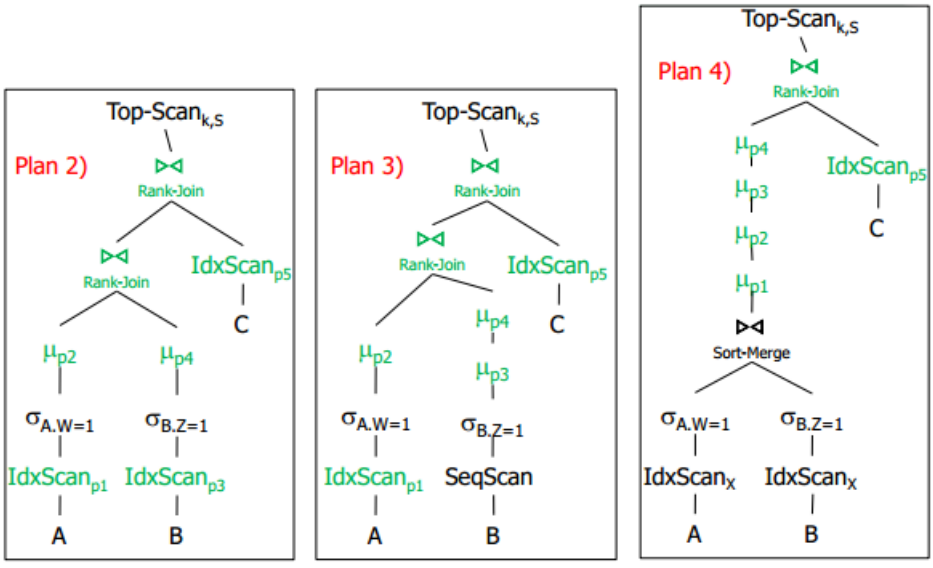
Un piano tradizionale (che non conosce il *ranking*) per riuscire a calcolare la query sfrutterebbe semplicemente l'operatore *Top-Sort* per evitare di ordinare tutte le tuple prodotte dal secondo (il più alto) join. Si veda la figura del piano d'accesso tradizione (Fig. 62)

Figura 62:



Invece con operatori consci del ranking esistono diversi piani, tutti possibili. Si veda la figura (Fig. 63)

Figura 63:



Risultati sperimentali mostano che nessun piano di accesso è il migliore, ha bisogno di ottimizzazione. Il piano (anche se prendiamo quello ottimale) tradizionale non è quasi mai il migliore.

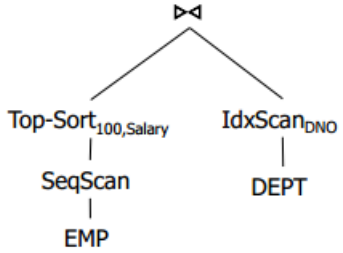
Come problema finale riguardante le *top-k queries*, consideriamo il problema di dove l'operatore *Top* può essere posizionato in un piano di accesso. La posizione di default dell'operatore è in cima al piano di accesso: notiamo che questo è anche il caso dei piani di accesso precedentemente mostrati (Fig. 63).

In alcuni casi è possibile scartare alcune tuple in eccesso anticipatamente, per esempio, nella seguente query

```
SELECT E.*, D.Dname
FROM EMP AS E, DEPT AS D
WHERE E.DNO=D.DNO
ORDER BY E.Salary DESC
STOP AFTER 100
```

In questo caso possiamo effettuare il join dei top-100 impiegati, e il piano di accesso risultante è il seguente (Fig. 64): questo perchè i top-100 impiegati risultano indipendenti dal dipartimento, ovvero il join su dipartimento serve solo per avere una informazione in più "D.Dname".

Figura 64:



Se anticipassimo la valutazione dell'operatore *Top*, dobbiamo essere sicuri che nessuna delle sue tuple di output possa essere successivamente eliminata dagli altri operatori. Questo può essere verificato controllando i vincoli di integrità del database (come FK, PK, NOT NULL, ...) e verificando che i predicati della query che rimangono da valutare successivi all'operatore *Top* vengano eseguiti.

Ad esempio con la seguente query

```
SELET E.*, D.Dname
FROM EMP AS E, DEPT AS D
WHERE E.DNO=D.DNO
ORDER BY E.Salary DESC
STOP AFTER 100
```

l'operatore *Top* può essere eseguito successivamente al join purchè E.DNO sia dichiarato come chiave esterna e con valori non nulli.

2.5.4 Sommario top-k join M:N

Per *top-k join* M:N *query* il caso più semplice da affrontare lo si ha quando gli accessi casuali sono possibili, in tal caso si applicano i principi dell'algoritmo

TA. L'operatore *Rank-Join* è stato creato per scenari in cui gli accessi casuali non sono possibili. Il *Rank-Join* con il *corner bound* è *ottimo per le istanze* solo quando le condizioni di join non sono tenute in considerazione o quando ci sono solo due input, ognuno con un solo punteggio parziale.

L'algebra RankSQL rappresenta un contributo rilevante nel creare *dbms* interamente consci del *ranking*: i suoi principi di design derivano dalla necessità di *splitting* e *interleaving* della valutazione della *scoring function*. Il RankSQL gestisce *rank-relations*, in cui le tuple sono ordinate in accordo al *ranking principle*. Il più recente operatore di *rank* valuta solo una singola "preferenza".