Tecnologie delle Basi di Dati M

Antonio Davide Calì

28 maggio 2014

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

Indice

Parte I
Skyline Queries

Sebbene le *scoring function* siano utilizzate per classificare un insieme di oggetti, è riconosciuto al giorno d'oggi che soffronoto di alcuni importanti problemi: in primo luogo hanno una capacità espressiva limitata, infatti possono solo catturare quei casi in cui le "preferenze" possono essere "trasformate" in numeri, cosa non sempre possibile (o al più non sempre naturale). Un secondo problema

riguarda quale *scoring function* usare, ovvero quale risulta "migliore" implementare? E quali pesi attribuire alle varie preferenze? La scelta è molto difficile e per nulla immediata.

In questa sezione studieremo un alternativa alle *scoring function*, le cosiddette *skyline query*, le quali hanno un rilevante utilizzo pratio e rappresentano il maggior passo verso modelli con preferenze più generali (e quindi più potenti).

Dominazione fra tuple Un concetto fondamentale che bisogna introdurre prima di definire le *skyline query* è il concetto di **dominazione fra tuple**. Data una relazione R $R(A_1, A_2, \ldots, A_m, \ldots)$ in cui gli attributi A_i sono gli attributi di *ranking*, assumiamo senza perdita di generalità che su ogni A_i valori più bassi sono migliori. Una tupla t *domina* una tupla t' rispetto agli attributi di ranking $A = \{A_1, A_2, \ldots, A_m\}$ e si scrive con $t \succ_A t'$ o semplicemente con $t \succ t'$ se il contesto è chiaro, se e solo se

$$\forall j = 1, \ldots, m \, : \, t.A_j \leq t'.A_j \, \wedge \, \exists j \, : \, t.A_j < t'.A_j$$

ovvero se la tupla t è non peggiore della tupla t' per tutti gli attributi, e risulta strettamente migliore di t' per almeno un attributo. Si noti che esistono i casi in cui non vale né $t \succ t'$ né $t' \succ t$ ovvero nessuna delle due tuple domina l'altra e in questo caso si dice che le tuple sono *indifferenti*.

La generalizzazione al caso in cui si vogliano massimizzare gli attributi di ranking o di renderli più vicinti a un qualisiasi punto target è immediata.

Facciamo un esempio (Fig. 1) in cui sia i *punti* che i *rimbalzi* debbano essere massimizzati.

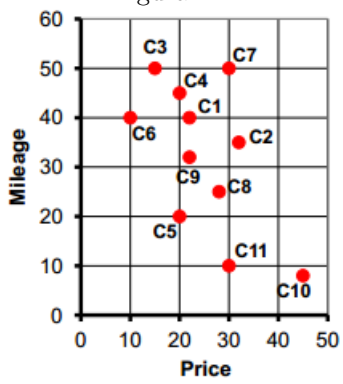
Figura 1:

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	...
...	...	...	...

In questo caso Tracy McGrady domina tutti i giocatori ad eccezion fatta di Yao Ming e Shaquille O'Neal. Shaquille O'Neal domina solo Yao Ming e Steve Nash. Yao Ming domina solo Steve Nash. Steve Nash non domina nessuno.

Un secondo esempio è il seguente (Fig. 2) in cui entrambi gli attributi devono essere minimizzati.

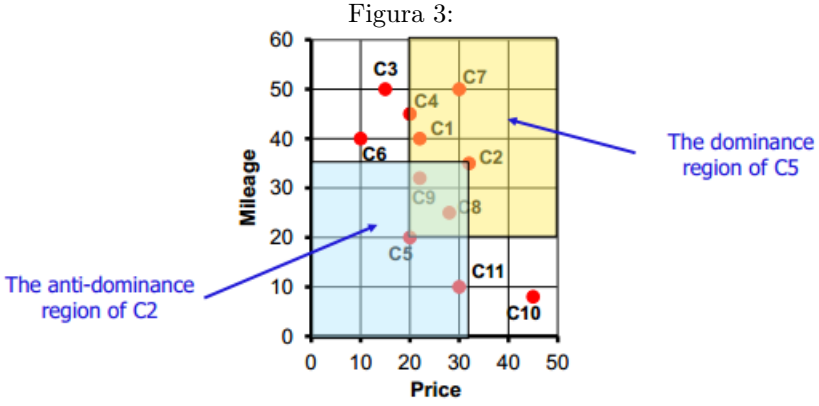
Figura 2:



La macchina C6 domina C1 (avendo lo stesso chilometraggio ma un prezzo inferiore), C3, C4 e C7. La macchina C5 domina C1, C2, C4, C7, C8 e C9. etc. . .

Il concetto di dominazione di tuple porta con se la creazione di *regioni di dominazione*. Si veda la seguente figura (Fig. 3) in cui vengono mostrate la *regione di dominazione* del punto C5 (rettangolo giallo) e la *regione di anti-dominazione* del punto C2 (rettangolo azzurro).

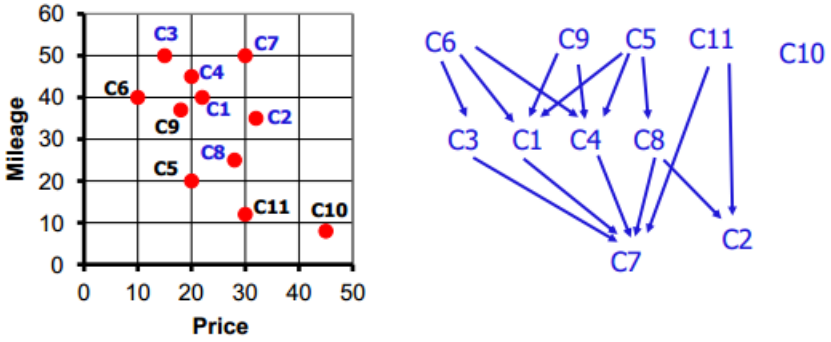
Figura 3:



La *regione di dominazione* di una tupla t è l'insieme dei punti nel dominio di lavoro che sono *dominati da t*. Similarmente, la *regione di anti-dominazione* della tupla t è l'insieme dei punti nel dominio di lavoro che *dominano t*. Chiaramente $t \succ t'$ se e solo se *t'* giace nella regione di dominazione di t e t giace nella regione di anti-dominazione di *t'*.

È possibile visualizzare la dominazione fra tuple attraverso un grafo, in cui per semplicità, vengono omesse le dominazioni transitive. Si controlli la figura (Fig. 4) per una maggiore comprensione.

Figura 4:



1 Skyline Query

Introduciamo la definizione di **skyline** di una **relazione** [BKS01]: data una relazione $R(A_1, A_2, \ldots, A_m, \ldots)$ in cui A_i sono gli attributi di *ranking*, la *skyline* della relazione R rispetto agli attributi di ranking $A = \{A_1, A_2, \ldots, A_m\}$, denotata con $Sky_A(R)$ o semplicemente con $Sky(R)$, è l'insieme delle tuple di R *non dominate*, ovvero

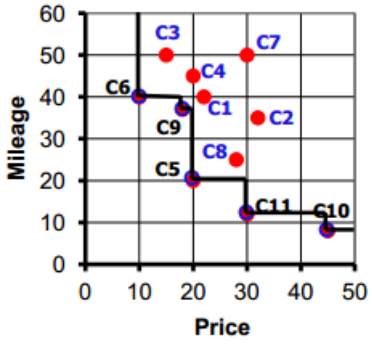
$$Sky(R) = \{t \,|\, t \in R,\ \nexists t' \in R : t' \succ t\}$$

Equivalentemente la tupla $t \in Sky(R)$ se e solo se nessun punto in R giace nella *regione di anti-dominazione* di t.

In geometria computazionale, le *skyline query* sono anche conosciute come "il problema dei massimi vettori"; per problemi di ottimizzazione multi-obiettivo il loro risultato è l'insieme delle soluzioni *ottime* cosiddette *Pareto*.

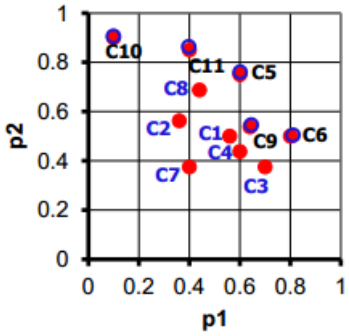
Prendiamo un esempio nello spazio degli attributi (Fig. 5): il "*profilo skyline*" è il risultato dell'unione delle regioni di dominazione dei punti skyline.

Figura 5:



Il seguente esempio (Fig. 6), invece, è mostrato nello spazio dei punteggi: come si vede non importa come i punteggi vengono definiti, lo *skyline* non cambia, ovvero lo *skyline* è invariante a qualsiasi "stiramento" di coordinate.

Figura 6:



Perchè sono così importanti le *skyline query*?

Indichiamo con **MD** l'insieme di tutte le *funzioni distanza monotone*. Dato un punto *target* **q**, vale la seguente relazione fra le *skyline query* e le 1-NN *query*

$$t \in Sky(R) \iff \exists d \in MD : \forall t' \in R, t' \neq t : d(t,q) < d(t',q)$$

ovvero se t è l'unica tupla risultante di una 1-NN query per una certa *funzione di distanza* d *monotona*, allora t appartiene allo *skyline*; viceversa se t è un punto *skyline* allora esiste (almeno) una funzione di distanza d monotona minimizzata solo da t.

Per questo motivo i punti *skyline* sono spesso chiamati "potenziali NN" poichè i punti skyline sono tutti i punti che sono potenzialmente vincitori in una 1-NN (dipende dalla funzione di distanza monotona applicata). Chiaramente il risultato rimane valido per *scoring function monotone*.

Dimostrazione

1) Se la tupla t è l'unica 1-NN per una funzione di distanza monotona d, allora t fa parte dello *skyline*

Dimostriamo per assurdo negando la conclusione, assumendo che t non faccia parte dello skyline, cioè esiste una tupla t' che domina t ovvero $\forall j\, t'.A_j \leq t.A_j \wedge \exists j * t'.A_{j*} < t.A_{j*}$ e siccome la funzione di distanza d è monotona avviene che $d(t',q) \leq d(t,q)$ che porta a una contraddizione

2) Se la tupla t è un punto dello skyline allora esiste (almeno) una fuzione di distanza monotona d minimizzata solo da t.

La dimostrazione è costruttiva: senza perdita di generalità poniamo il punto *target* $q = 0$ e assumiamo che tutti i valori degli attributi siano strettamente positivi. Consideriamo la distanza pesata $L_{\infty,w}$ con pesi uguali a $w_i = \frac{1}{t.A_i}$ per $i = 1, \ldots, m$. Si ha che $L_{\infty,w}(t,0) = max_i\{w_i \cdot t.A_i\} = 1$. Per qualsiasi altro

punto t' si ha invece che $L_{\infty,w}(t',0) = max_i\{w_i \cdot t'.A_i\} = max_i\{\frac{t'.A_i}{t.A_i}\} > 1$ e dunque t è un punto dello *skyline*.

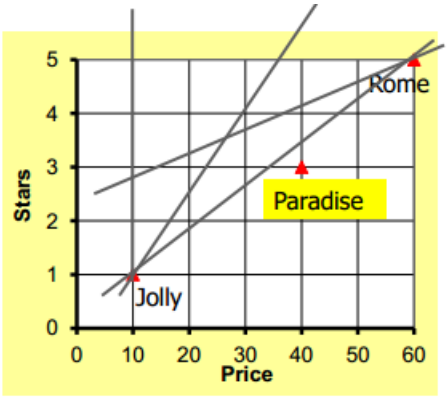
Prendiamo il seguente esempio (Fig. 7) e controlliamo l'accessibilità dei punto dello skyline, in cui la *scoring function* lineare è data da $S = Ws \cdot Stars - Wp \cdot price$

Figura 7:

S = Ws * Stars – Wp * Price

Hotels

Name	Price	Stars
Jolly	10	1
Rome	60	5
Paradise	40	3



Ciò che si vuole osservare è che in determinati dataset, un determinato *skyline point* (come Paradise nell'esempio) potrebbe non essere mai restituito come top-1 per qualsiasi scoring function lineare (anche al variare dei pesi applicati) scelta. Problemi simili si hanno con un valore arbitrario di k e/o con la maggior parte di *scoring function*.

Infatti, lo *Skyline* non ammette qualsiasi *funzione di distanza*. Lo *skyline* di R non corrisponde a qualsiasi risultato k-NN (o top-k), ovvero dato uno schema $R(\overline{A_1}, A_2, \ldots, A_m, \ldots)$ non esiste una funzione distanza d (o equivalentemente una *scoring function* S) che su tutte le possibili istanze di R mantiene nelle prime k posizioni i punti dello *skyline*. Si noti che stiamo permettendo a k di variare così da poter renderlo equivalente al numero di punti dello *skyline* in ogni istanza di R.

Dimostrazione: Per la dimostrazione utilizziamo un esempio (Fig. 8). Si ha $Sky(R') = \{t1, t4\}$ cioè $\{S(t1), S(t4)\} > S(t2)$. D'altro canto si ha anche $Sky(R'') = \{t2, t3\}$ cioè $\{S(t2), S(t3)\} > S(t4)$ il che porta a una contraddizione.

Figura 8:

R'

TID	p1	p2
t1	0.9	0.6
t2	0.8	0.4
t4	0.5	0.7

R''

TID	p1	p2
t2	0.8	0.4
t3	0.7	0.8
t4	0.5	0.7

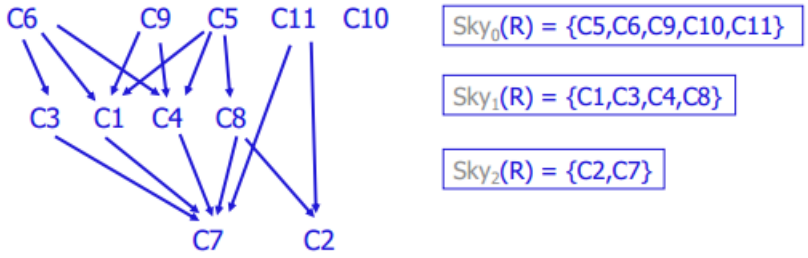
1.0.1 Ranking con Skyline

Il ranking di tuple può essere semplicemente ottenuto *iterando* l'operatore *skyline* così definito:

1. $Sky_0(R) = Sky(R)$
2. $Sky_1(R) = Sky(R - Sky_0(R))$
3. $Sky_2(R) = Sky(R - Sky_0(R) - Sky_1(R))$
4. ...

Così si ottiene che $Sky_0(R)$ sono le tuple "top", $Sky_1(R)$ sono le "seconde" scelte e così via. Si guardi la figura (Fig. 9) per una maggiore comprensione.

Figura 9:



2 Valutazione Skyline

Il problema di un calcolo efficiente delle *skyline query* è stato largamente discusso e molti algoritmi sono stati introdotto. Un motivo base è dovuto al fatto che il problema risulta "più difficile" delle *top-k query* poichè ha complessità $\Theta(N^2)$ nel caso peggiore per un database di N oggetti. Ciò che vedremo sono algoritmi che seguono uno dei due seguenti approcci base:

1. *Generico*: calcola lo *skyline* senza l'ausilio di alcun metodo di accesso (indici) cioè la relazione di input può essere anche l'*output* di qualche altra operazione (join, group by, etc...)
2. *Basati su indici*: nel quale si assume che sia disponibile un indice.

2.1 Algoritmo Nested-Loop (NL)

Il modo più semplice (ed inefficiente) di calcolare lo *skyline* di R è di confrontare ogni tupla con tutte le altre.

Nested Loop

- **Input**: Un dataset R, un insieme di attributi **A** che inducono $\succ$
- **Output**: $Sky(R)$ ovvero lo *skyline* di R rispetto ad **A**

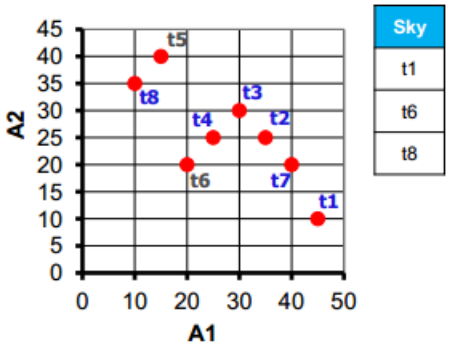
Figura 10: Pseudo-Codice Algoritmo NL

```
ALGORITHM NL (nested-loops)
Input: a dataset R, a set of attributes A inducing ≻
Output: Sky(R), the skyline of R with respect to A
1.  Sky(R) := ∅;
2.  for all tuples t in R:
3.      undominated := true;
4.      for all tuples t' in R:
5.                  if t' ≻ t then: {undominated := false; break}
6.      if undominated then: Sky(R) := Sky(R) ∪ {t};
7.  return Sky(R);
8.  end.
```

E vediamo subito un esempio (Fig. 11) in cui il punto *target* è l'origine. Si noti che se $t \in Sky(R)$ allora t sarà sempre confrontata con tutte le altre tuple.

Figura 11:

R
t1
t2
t3
t4
t5
t6
t7
t8



Sky
t1
t6
t8

TID	No. of comparisons
t1	7
t2	3
t3	3
t4	5
t5	7
t6	7
t7	6
t8	7
Total	**45**

2.2 Algoritmo Block-Nested-Loops (BNL)

L'algoritmo *BNL* [BKS01] migliora l'algoritmo *NL* poichè scarta immediatamente tutte le tuple che sono dominate da almeno un'altra tupla, cioè permette di evitare di confrontare due volte la stessa coppia di tuple (come NL fa).

BNL alloca un buffer (*window*) W in memoria principale, la cui grandezza è un parametro di *design*, e legge sequenzialmente il *data file*. Ogni nuova tupla t che viene letta dal *data file* è confrontata con solo quelle tuple che sono

attualmente in W. Si noti che l'algoritmo BNL proposto per le *skyline query* ha in realtà un'applicabilità molto più generale.

La logica che contraddistingue BNL è che quando si legge una nuova tupla t sono possibili 3 casi:

1. Se una qualche tupla t' nel buffer W domina la tupla t, allora t è immediatamente scartata

2. Se t domina alcune tuple t' in W allora tutte queste tuple vengono rimosse da W e vi viene inserita t (si noti che il caso 1 e il caso 2 sono mutuamente esclusivi).

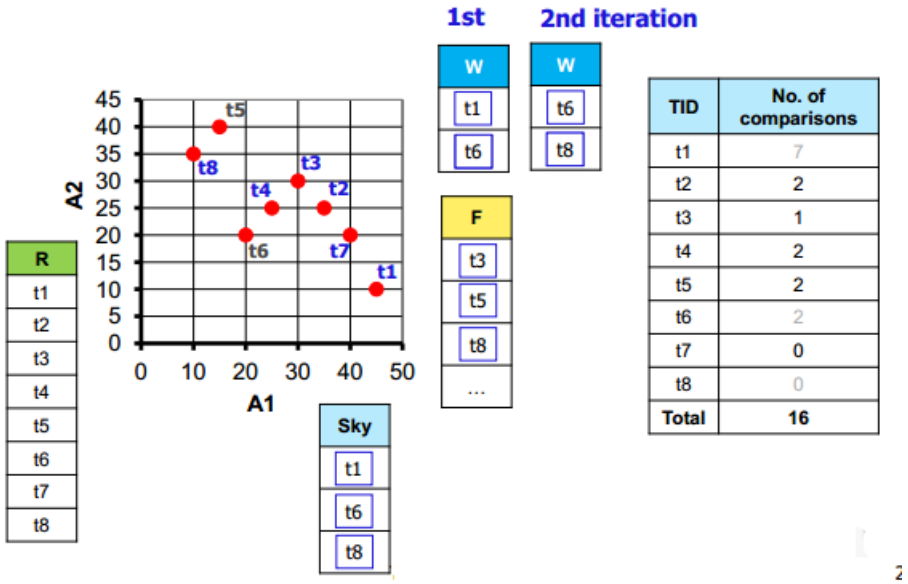
3. Se non accade nessuno dei due casi precedenti allora semplicemente si inserisce t nel buffer W. Se non vi è spazio sufficiente in W allora t viene scritta su un file temporaneo F

Quando tutte le tuple sono state processate, se F è vuoto allora l'algoritmo si ferma, altrimenti inizia una nuova iterazione dell'algoritmo usando F come nuovo input.

Le tuple che sono state inserite in W quando F era vuoto possono essere immediatamente restituite come output, poichè sono state confrontate con tutte le altre tuple. Le restanti tuple in W (ovvero quelle inserite quando F risulta non vuoto) possono essere restuite alla prossima iterazione: una tupla t può essere restituita come output quando viene trovata una tupla t' in F che segue t in ordine sequenziale, per far ciò è richiesto di aggiungere un *timestamp* (o un counter) ad ogni tupla.

Vediamo un primo esempio (Fig. 12) in cui si assume la grandezza di W pari a 2 ($|W| = 2$) e in cui il punto *target* risulta essere l'origine. Si vede che per ogni tupla t vengono conteggiati soltanto i confronti con tuple seguenti t in R.

Figura 12:



Considerazioni Risultati sperimenali su [BKS01] mostrano che *BNL* è *CPU-bound* e che le sue performance deteriorano se W cresce poichè con valori alti di W, BNL esegue più confronti. D'altro canto BNL ha un costo relativamente basso di I/O.

Facciamo un esempio limite per tentare di spiegare perchè al crescere di W le performance peggiorano. Si immagini di avere i due casi limite con W illimitata e con $|W| = 1$ e l'istanza del problema abbiamo N tuple di cui le prime N-1 tuple sono *indifferenti* fra loro mentre l'ultima tupla domina tutte le altre. Nel caso di *window illimitata* fintato che leggo una nuova tupla fra le prime N-1 viene inserita nella *window* e quindi ogni nuova tupla deve confrontarsi con ogni tupla già presente in window portando a $\binom{N-1}{2}$ confronti ed infine arriva l'ultima tupla t che dominando tutte le precedenti le elimina dalla window stessa. Nel caso invece in cui la window abbia capacità uguale a uno, la prima tupla viene inserita nella window, tutte le successive essendo indifferenti alla prima, vengono inserite nel file F (dunque effettuano solo il cronfronto con la prima tupla nella window e non con quelle del file). Infine arriva l'ultima tupla che domina l'unica tupla in window, la quale viene eliminata dal buffer per far posto alla nuova tupla dominante. All'iterazione successiva a questo punto vengono confrontate tutte le N-2 tuple nel file con la tupla presente nella window (che le domina tutte): il numero di confronti totali è $N - 1 + N - 2$ lineare in N.

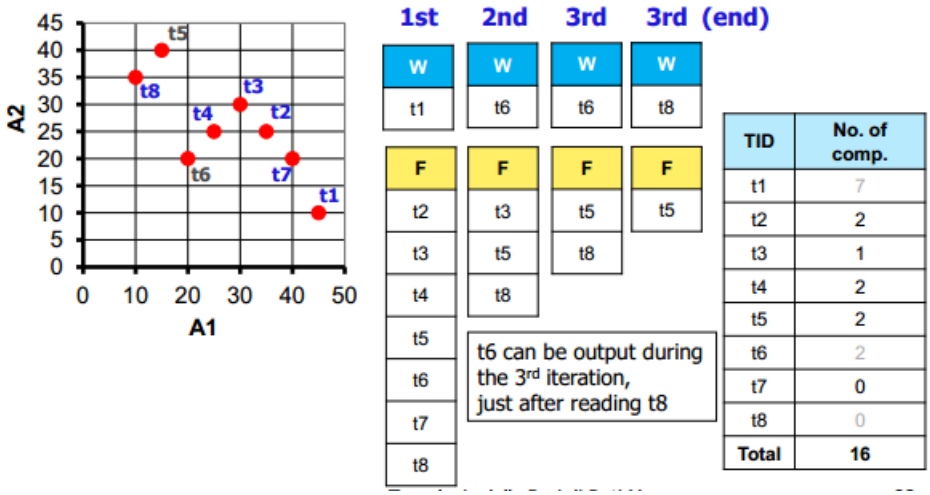
Le performance sono inoltre influenzate negativamente dal numeri di punti dello *skyline*. La cardinalità dello skyline dipende dal numero di attributi e dalle

loro correlazioni: gli attributi negativamente *correlati* (o anti-correlati), come *Price* e *Mileage*, tendono ad avere uno *skyline* pià grande.

[BKS01] ha inoltre indrodotto alcune varianti rispetto a BNL, come ad esempio BNL-sol che gestisce W come una *lista auto-organizzata* in cui l'idea è di confrantare prima gli oggetti in arrivo con quelli presenti in W (chiamati oggetti "killer") che sono stati trovati a dominare molti altri oggetti; oppure come ad esempio algoritmi D&C che si basano sull'approccio *divide et counquer*.

Mostriamo un ultimo esempio (Fig. 13) in cui impostiamo la cardinalità di W a 1 ($|W| = 1$): ciò rende minimo il numero di confronti per un dato ordine di input. Si noti come t6 può essere restituita come risultato durante la terza iterazione, subito dopo aver letto t8.

Figura 13:



2.3 Algoritmo Sort-Filter-Skyline (SFS)

L'algoritmo SFS ha come obiettivo quello di ridurre il numero di confronti: per riuscirci esegue un *ordinamento topologico* dei dati di input il quale rispetta il criterio di preferenza dello *skyline*.

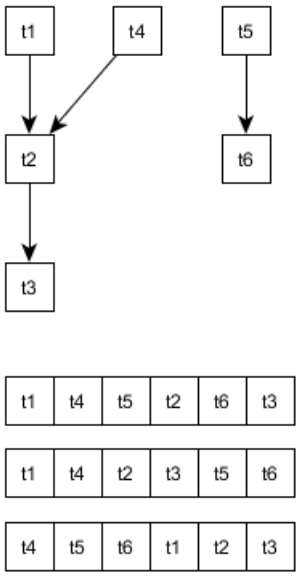
Diamo la definizione di **ordinamento topologico**: data $\succ$, un *ordinamento topologico* di R è un ordinamento completo (senza parità) definito con $<$ delle tuple in R tali che

$$t \succ t' \Rightarrow t < t'$$

cioè se la tupla t domina t' allora t precede t' nell'ordinamento completo.

Si veda la seguente figura (Fig. 14) per una maggiore comprensione

Figura 14:



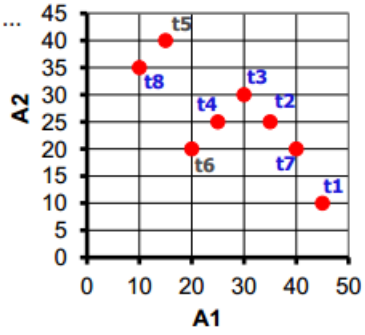
L'osservazione chiave è la seguente: se l'input è *ordinato topologicamente* allora una nuova tupla appena letta non può dominare alcuna tupla precedentemente letta (poichè $t > t' \Rightarrow t \not\succ t'$).

Nella seguente figura (Fig. 15) sono mostrati possibili risultati di ordinamenti topologici. In pratica un ordinamento topologico è ottenuto ordinando i dati utilizzando una funzione di distanza (o una scoring function) monotona (quali ad esempio la somma e il prodotto sono) applicata ai criteri di skyline.

Figura 15:

			sum		product
t6	t8	t1	t6 40	t8	350
t4	t5	t6	t8 45	t6	400
t2	t6	t4	t4 50	t1	450
t3	t1	t7	t5 55	t5	600
t1	t4	t8	t1 55	t4	625
t8	t7	t3	t2 60	t7	800
t7	t3	t2	t3 60	t2	875
t5	t2	t5	t7 60	t3	900

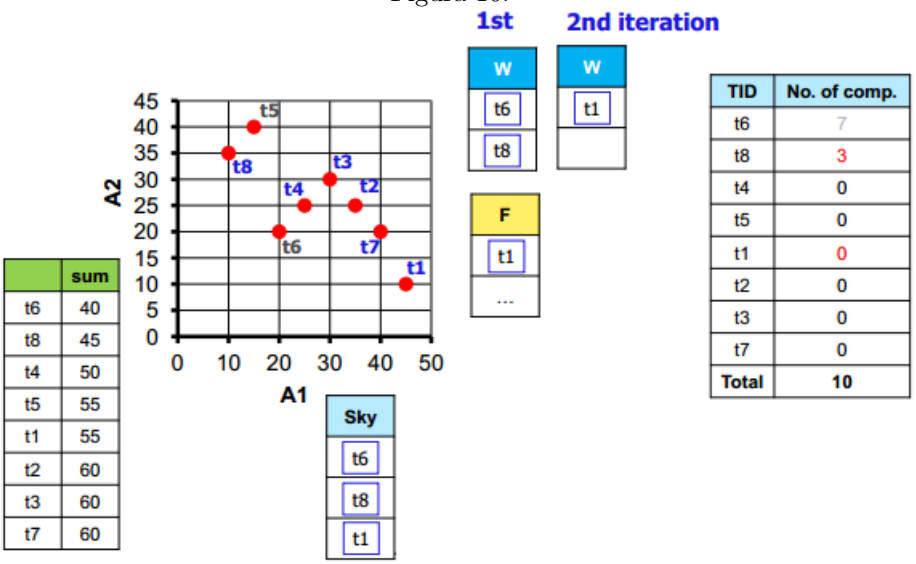
...



E mostriamo di seguito un esempio (Fig. 16) di come SFS funzioni assumendo $|W| = 2$ e punto target l'origine. Si noti come che per ogni tupla t vengono

conteggiati soltanto i confronti con tuple seguenti t in nell'input ordinato.

Figura 16:



SFS gode inoltre di ulteriori proprietà:

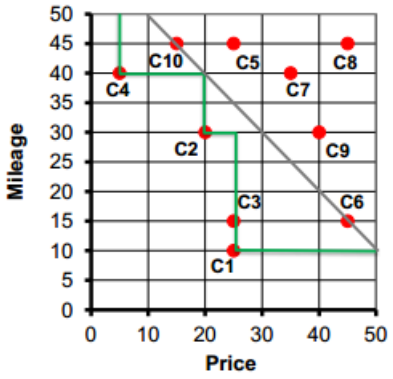
- Alla fine di ogni itereazione tutte le tuple in W possono essere restituite come output, poichè nessuna tupla in W può essere scartata da una successiva tupla
- Il numero di iterazioni è pertanto il minimo possibile: $\left\lceil \frac{|Sky(R)|}{|W|} \right\rceil$, a differenza di BNL che non garantisce nulla di ciò
- SFS può restituire una tupla appena viene inserita nel buffer W, ma in W si possono memorizzare solo i valori degli attributi *skyline* (con obiettivo di confrontare i successivi valori delle tuple per verificare se questi sono dominati), il che porta a ridurre di molto lo spazio richiesto
- Due tuple non-*skyline* non verranno mai confrontate, poichè in W sono presenti solo tuple skyline: infatti i confronti possibili possono essere solo fra due tuple presenti nello *skyline* e quindi ad aver diritto a stare nella window W, oppure confronto fra una tupla skyline e una no.
- Gestire la struttura del buffer (*window*) è ora molto più semplice poichè si suppongono possibili solo inserimenti e non vi è mai la cancellazione di tuple specifiche, quindi non bisogna gestire gli slot vuoti.

2.4 Algoritmo SaLSa

L'algoritmo *SaLSa* (Sort and Limit Skyline Algorithm) estende l'idea di SFS osservando che quando i dati sono ordinati topologicamente è possibile evitare di leggere tutte le tuple di input.

Prendiamo il seguente esempio (Fig. 17) in cui i dati sono ordinati utilizzando la *scoring function* SUM siffata $t.Price + t.Mileage$.

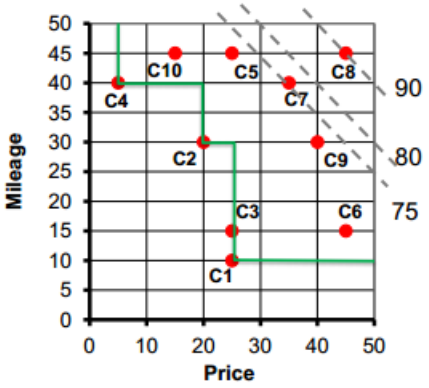
Figura 17:



Dopo aver letto C6 (o C10) la cui somma è pari a 60, sappiamo che non esistono ulteriori punti *skyline*, tutta via usare tutti gli attuali punti in Sky(R) per questo scopo è costoso, infatti il problema è *NP-hard* [BCP08].

Per risolvere il problema SaLSA fa uso di una *sola* tupla skyline, la cosiddetta *stop-point* denotata con t_{stop}, per determinare quando l'esecuzione può essere fermata. In questo caso è sufficiente controllare se ciò che bisogna ancora leggere giace nella *regione di dominazione* di t_{stop}. Si veda il seguente esempio (Fig. 18) per una maggiore comprensione.

Figura 18:



t_{stop}	halt when sum ≥
C1	75
C2	80
C4	90

Per funzioni di distanza (o scoring function) *simmetriche* (ovvero con pesi uguali o che scambiando gli argomenti la funzione non cambia), e assumendo che le coordinate lavorino tutte sullo stesso range ([0,1], [0,50], etc...) è possibile provare che la scelta *ottimale* per il *stop-point* è data dalla seguente regola

$$t_{stop} = argmin_{t \in Sky(R)} \{max_i \{t.A_i\}\}$$

ovvero la tupla per cui il *valore della coordinata massima è minimo*. Si noti che ciò è valido per qualsiasi funzione di distanza simmetrica. Il seguente esempio (Fig. 19) chiarirà le idee.

Figura 19:

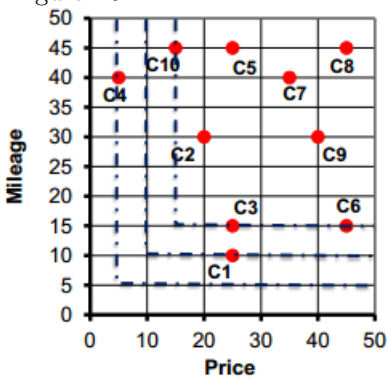
t_{stop}	Price	Mileage	halt when sum ≥
C1	25	10	75
C2	20	30	80
C4	5	40	90

Scelta dell'ordinamento ottimale Tra le molte alternative possibili di ordinare i dati di input, SaLSa utilizza un *criterio dimostrabilmente ottimale*, ovvero in ogni istanza ordinando i dati utilizzando un'altra funzione simmetrica non permette di scartare punti in più. Il criterio di ottimalità è chiamato **minC** (minimum coordinate), cioè per ogni tupla t viene utilizzato il valore di $min_i\{t.A_i\}$ (ovvero vengono ordinate utilizzando la coordinata convalore minimo). In caso di parità si utilizza il criterio secondario di "Sum". Questo perchè ricordando le curve a equi-punteggio, la curva del minimo è effettivamente quella che riesce a "cancellare" più tuple poichè a differenza della curva SUM rappresentata da una retta e ricordando che è possibile fermarsi quando ciò che bisogna ancora vedere giace *completamente* all'interno della regione di dominazione della tupla t_{stop}, allora possiamo capire che SUM scarta meno tuple poichè forma un "triangolo" con il grafico, mentre la funzione di MIN copia ugualmente la regione di dominazione.

Si controlli il seguente esempio (Fig. 20) per una maggiore comprensione.

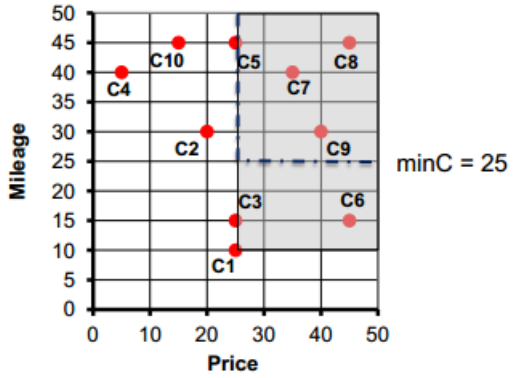
Figura 20:

	minC	sum
C4	5	
C1	10	
C3	15	40
C6	15	60
C10	15	60
C2	20	
C5	25	
C9	30	
C7	35	
C8	45	



Vediamo un esempio (Fig. 21) su come l'algoritmo funziona, utilizzando l'ordinamento minC per fermarsi. Lo *stop-point* è C1 per il quale si ha $max_i \{C1.A_i\} = 25$. Appena si ha che $minC \geq 25$ allora SaLSa può fermarsi. La condizione generale di stop è dunque la seguente $minC \geq max_i \{t_{stop}.A_i\}$.

Figura 21:



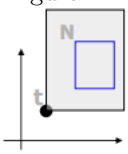
2.5 Algoritmo BBS

Se abbiamo un *indice* sugli attributi di ranking, possiamo usarlo per evitare di scandire tutto il database. L'algoritmo **BBS** (Branch and Bound Skylin) [PTF+03] è una reminiscenza sia del *k-NNOptimal*, nel quale si accede ai nodi dell'indice per valori crescenti di *MinDist* (da qui in seguito, il punto target coincide con l'origine) sia dell'algoritmo *next-NN*, nella cui coda PQ si mantengono sia tuple che nodi. Per migliorare i costi [PTF+03] calcola le distanze usando L_1 (la distanza di Manhattan).

L'obiettivo base dell'algoritmo è di evitare di accedere ai nodi dell'indice che non possono contenere alcun oggetto *skyline*. Per far ciò sfrutta la seguen-

te osservazione: se la regione $Reg(N)$ del nodo N giace completamente nella *regione di dominanza* di una tupla t, allora N non può contere alcun punto *skyline* (ovvero t domina N). Si controlli la figura (Fig. 22) per una maggiore comprensione.

Figura 22:



Inoltre sfrutta anche ormai il ben noto fatto che se $L_1(t', 0) \geq L_1(t, 0)$ allora $t' \not\succ t$. La coda PQ inoltre memorizza anche $key(N)$ ovvero la MBR (Minimum Bound Rectangle) di N per verificare se N è dominata da una qualche tupla t.

Algoritmo BBS

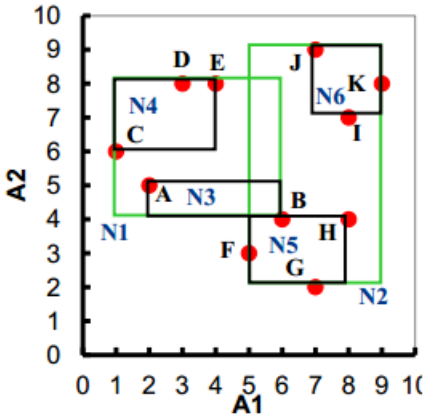
- **Input**: l'albero indice con nodo radice RN
- **Output**: *Sky*, lo *skyline* dei dati indicizzati

Figura 23: Pseudo-Codice Algoritmo BBS

```
Input: index tree with root node RN
Output: Sky, the skyline of the indexed data
 1. Initialize PQ with [ptr(RN),Dom(R),0];   // starts from the root node
 2. Sky := ∅;                  // the Skyline is initially empty
 3. while PQ ≠ ∅:                        // until the queue is not empty...
 4.    [ptr(Elem), key(Elem), dMIN(0,Reg(Elem))] := DEQUEUE(PQ);
 5.    If no point in Sky dominates Elem then:
 6.       if Elem is a tuple t then: Sky := Sky ∪ {t}
 7.          else: { Read(Elem);   // ...node Elem might contain skyline points
 8.             if Elem is a leaf then: { for each tuple t in Elem:
 9.                if no tuple in Sky dominates t then:
10.                                   ENQUEUE(PQ,[ptr(t), key(t), L1(0,key(t))]) }
11.                                else: { for each child node Nc of Elem:
12.                if no point in Sky dominates Nc then:
13.                                ENQUEUE(PQ,[ptr(Nc), key(Nc), dMIN(0,Reg(Nc))]) }};
14. return Sky;
15. end.
```

Vediamo di seguito un esempio in cui i dati iniziali sono dati da (Fig. 24) e l'algoritmo in esecuzione è mostrato in (Fig. 25) in cui la distanza è L_1.

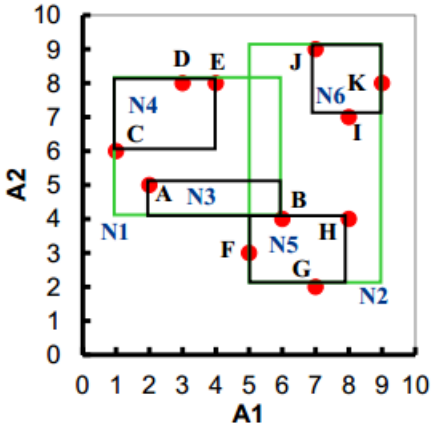
Figura 24:



Elem	d_{MIN}
A	7
B	10
C	7
D	11
E	12
F	8
G	9
H	12
I	15
J	16
K	17
N1	5
N2	7
N3	6
N4	7
N5	7
N6	14

Figura 25:

Action	PQ			
Read(RN)	(N1,5)	(N2,7)		
Read(N1)	(N3,6)	(N4,7)	(N2,7)	
Read(N3)	(A,7)	(N4,7)	(N2,7)	(B,10)
Return(A)	(N4,7)	(N2,7)	(B,10)	
Read(N4)	(C,7)	(N2,7)	(B,10)	
Return(C)	(N2,7)	(B,10)		
Read(N2)	(N5,7)	(B,10)		
Read(N5)	(F,8)	(G,9)	(B,10)	
Return(F)	(G,9)	(B,10)		
Return(G)	(B,10)			

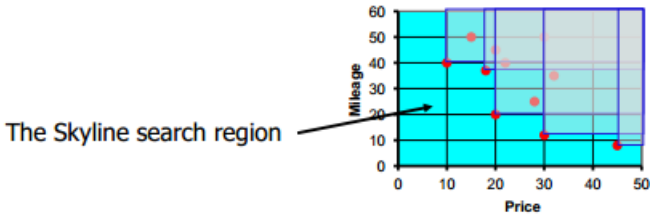


L'esempio mostra chiaramente perchè una tupla *attualmente* non dominata, come B, la quale è salvata nel nodo N3, necessita di essere inserita nella coda *PQ*.

La *corretezza* del'algoritmo **BBS** è molto semplice da provare, poichè l'algoritmo scarta solo i nodi trovati che sono dominati da un qualche punto dello *Skyline*.

Come *SFS* e *SaLSa* quando una tupla t è inserita nell'insieme *Sky* allora c'è la garanzia che t fa parte del risultato finale: questa è una diretta conseguenza

Figura 26:



dell'accesso ai nodi per valori incrementali di *MinDist* e dell'inserzione di una tupla in *Sky* solo quando diventa il primo elemento di *PQ*.

L'*ottimalità* di **BBS** (che non dimostriamo formalmente) significa che *BBS* legge solo quei nodi che intersecano la "*regione di ricerca Skyline",* la quale è ottenuta facendo il complemento dell'unione delle *regioni di dominazione* dei punti *skyline.* Si veda la figura (Fig. 26) per una maggiore comprensione.

2.6 Algoritmo LS-B

In molti scenari, molti (possibilmente tutti) gli attributi di interesse possono assumere solo un piccolo range di valori (ad esempio le valutazioni di film, la presenza/assenza di una feature, i "predicati di preferenza", discretizzazione del dominio). Un esempio è mostrato in figura (Fig. 27).

Figura 27:

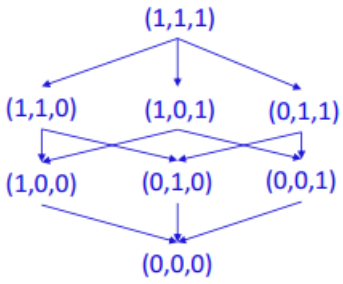
Hotel	Price	Stars	WiFi	Parking	Air Cond.
H1	35 €	*			
H2	30 €	**		✓	
H3	60 €	**			✓
H4	40 €	***	✓	✓	
H5	40 €	**		✓	

Nell'esempio $Sky(R) = \{H2, H3, H4\}$ poichè $H2 \succ H1$ e valgono contemporaneamente $H2 \succ H5$ e $H4 \succ H5$.

Gli algoritmi considerati finora sono incapaci di sfruttare la peculiarità di domini a bassa cardinalità.

L'algoritmo **LS-B** [MPJ07] assume che tutti gli attributi abbiamo bassa cardinalità. Senza perdita di generalità considereremo solo *m attributi booleani.* La corrispondente rete di booleani è composta da 2^m elementi che possono essere ordinati considerando che "1 è sempre migliore di 0". Si veda la figura (Fig. 28) per una maggiore comprensione.

Figura 28:



L'idea di LS-B è che solo le tuple nelle "migliori classi" (dove una classe è rappresentata da un nodo del reticolo) del reticolo fanno parte dello *skyline*. Vediamo ora come funziona.

L'algoritmo LS-B opera in due parti

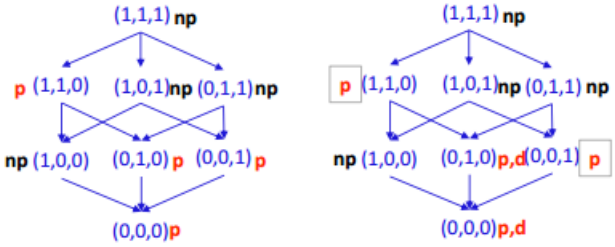
Fase 1 Legge tutte le tuple e marca come **presente** (p) il corrispondente elemento nel reticolo; i restanti elementi rimangono marcati come **non presenti** (np)

Fase 2 Leggi di nuovo tutte le tuple e restituisci in output quelle il cui elemento nel reticolo è *non dominato*

Si veda di seguito un esempio (Fig. 29) del funzionamento di LS-B.

Figura 29:

Hotel	WiFi	Parking	Air Cond.
H1			
H2		✓	
H3			✓
H4	✓	✓	
H5		✓	



2.7 Algoritmo LS

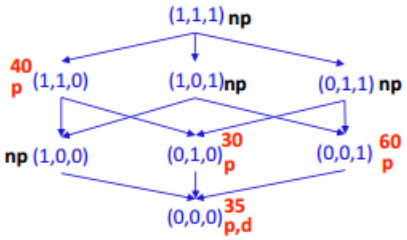
L'algoritmo **LS** [MPJ07] estende l'algoritmo *LS-B* permettendo la presenza di un attributo A_0 il cui dominio può essere arbitrariamente grande (ad esempio *Price*).

Nella *prima fase* l'algoritmo *LS* calcola anche il *valore locale ottimo* (*lov*) di A_0 per ogni elemento presente (ad esempio il prezzo più basso). Un elemento e è ora dominato se esiste un elemento *migliore e'* (ovvero la dominazione descritta in LS-B) nel reticolo il cui *lov* è non peggiore di *e.lov*.

Si veda di seguito un esempio della prima fase (Fig. 30)

Figura 30:

Hotel	Price	WiFi	Parking	Air Cond.
H1	35 €			
H2	30 €		✓	
H3	60 €			✓
H4	40 €	✓	✓	
H5	40 €		✓	



Nella *seconda fase* una tupla t il cui elemento e è *non dominato* può essere "tagliato" se e solo $t.A_0$ è peggiore di $e.lov$ (ovvero, una tupla di una determinata classe appartiene allo skyline se la tupla ha effettivamente il valore locale ottimo calcola precedentemente, altrimenti essa viene eliminata).

Si noti che non si conosce alcuna facile ed efficiente estensione quando più di un attributo ha un dominio grande (poichè per ogni elemento dovremmo calcolare uno *skyline* "locale").

3 Conclusioni

Esistono alcune varianti alle *skyline query*. [PTF+03] introduce alcune modifiche alle query *skyline* di base come ad esempio:

1. **Ranked Skyline Query** che ordinano lo *skyline* con una *scoring function*
2. **Constrained Skyline Query** che limitano la regione di ricerca
3. **K-Dominating Query** che restituiscono le k tuple che dominano il più grande numero di altre tuple

Molti altri problemi relativi allo *skyline* sono stati proposti e studiati finora, ad esempio le *Reverse Skyline Query* in cui dato un punto query q, si vuole sapere quali siano le tuple t tali che q sia nello *skyline* calcolatato rispettivamente a t (ovvero quando t è il target); oppure le *Representative Skyline Points* in cui si vuole sapere quali sono i k punti "più rappresentativi" dello *skyline*.

3.1 Sommario

Le *Skyline Query* rappresentano una valida alternativa alle *top-k query*, poichè non richiedono alcuna scelta su *scoring function* e su *pesi*. Lo *skyline* di una relazione R, $Sky(R)$, contiene tutte e sole le tuple *non dominate* in R, ovvero quelle tuple che rappresentano "interessanti alternative" da considerare. Il calcolo di $Sky(R)$ si può basare sia su algoritmi *sequenziali* sia su quelli basati su *indice*.

L'algortimo BNL lavora allocando un *buffer* (*window*) in memoria principale e confrontando le tuple in arrivo con quelle nella *window*.

SFS pre-ordina i dati mantenendo un *ordinamento topologico* che introduce diversi benefici se comparato a BNL.

SaLSa aggiunge una condizione di stop che permette di evitare di leggere tutti i dati.

BBS è un algoritmo dimostrabilmente ottimo per numero I/O per il calcolo di $Sky(R)$ utilizzando un *R-tree*.

Infine LS-B e LS sono stati creati per lavorare con domini di bassa cardinalità (e con al più un attributo con dominio arbitrariamente vasto).