Università
Ca'Foscari
Venezia

Corso di Laurea specialistica (*ordinamento ex D.M. 509/1999*)
in Informatica

Tesi di Laurea

—

Ca' Foscari
Dorsoduro 3246
30123 Venezia

# Cost Effective Semi-Automatic Algorithm for Materialized Views Selection in Data Warehousing in distribuite environment

**Relatore**
Ch. Prof. Renzo Orsini

**Laureando**
Matteo Brigo
Matricola 814728

**Anno Accademico 2015 / 2016**

## Indice

Abstract....3
1. Introduzione....4
1.1. Sviluppo del testo....4
1.2. Introduzione al problema....4
2. Related work....10
2.1 Terminologia e Metodologia....10
2.2 Un algoritmo greedy per la selezione delle viste materializzate...13
2.2.1. Esempio illustrato dell'algoritmo....15
2.3 Data Warehouse Configuration Problem....22
3. Algoritmo proposto....29
3.1 Euristica proposta....32
3.2 Esempio dell'algoritmo proposto....36
4. Verifica dell'algoritmo proposto, metodologia di stima....45
4.1. Stima della dimensione di una vista materializzata....46
4.2. Stima del tempo di costruzione di una vista materializzata....49
4.3. Stima del tempo di esecuzione di una query su viste materializzate....51
5. Caso di studio e raffronto tra gli algoritmi....52
5.1. Query frequenti....54
5.2. Dimensionalità e velocità del sistema....59
5.3 Tempo di esecuzione base delle query....60
5.4 Esempio di esecuzione dell'algoritmo 2.2....61
5.5 Esecuzione dell'algoritmo 2.3 e la versione proposta 3....66
5.6 Schema di confronto....69
6. Considerazioni finali e sviluppi futuri....71
7. Bibliografia....73

Università
Ca'Foscari
Venezia

## Abstract

L'analisi dei dati attraverso un sistema datawarehouse è sempre più presente e necessaria per permettere alle aziende una corretta politica decisionale.
Con l'aumentare della quantità di informazioni presenti nei sistemi informativi è cruciale concentrarsi sulla ottimizzazione delle query per il recupero di informazioni dai sistemi aziendali, inoltre sempre più spesso le aziende dispongono di sistemi distribuiti eterogenei.
Scopo di questa tesi è proporre un metodo per ottimizzare la velocità di reperimento delle informazioni, attraverso l'uso di viste materializzate, tenendo conto dei vincoli spaziali e della possibilità di avere sistemi con velocità di calcolo non omogenee.
Per far ciò abbiamo dapprima analizzato alcuni algoritmi già presenti in letteratura, concentrandoci su pregi e difetti, in secondo luogo abbiamo presentato una versione modificata di un algoritmo per tener conto dei vincoli aggiuntivi da noi premessi.
Infine abbiamo dato un esempio dell'algoritmo proposto e ne abbiamo confrontato il risultato con gli altri algoritmi descritti.

Università Ca'Foscari Venezia

# 1. Introduzione

## 1.1. Sviluppo del testo

Nello sviluppo del testo ci siamo concentrati nelle seguenti fasi: come prima cosa abbiamo analizzato il problema ( Cap 1.2 ) e abbiamo raccolto quante più informazioni storiche su come è stato affrontato in letteratura; si potrà ritrovare nella bibliografia il riferimento ai testi presi in considerazione.
In secondo luogo ci siamo concentrati nell'esplorazione di alcuni degli algoritmi storici, indicandone pregi e difetti e dandone un esempio di utilizzo (Cap. 2), proponendo in seguito una nostra soluzione evolutiva rispetto a quanto presente ad oggi in letteratura (Cap. 3), per proseguire poi nel dare un metodo per stimare l'efficacia della nostra soluzione e delle soluzioni precedenti (Cap. 4) ed infine portare un caso di studio realistico per testare la bontà dell'algoritmo proposto (Cap. 5).
Data la complessità di dar vita ad un algoritmo innovativo lasciamo a sviluppi futuri la dimostrazione della sua bontà, proponendo solo il metodo di valutazione di essa.

## 1.2. Introduzione al problema

Con il termine data warehouse (abbreviato d'ora in poi in DW) si intende un archivio in grado di memorizzare gradi volumi di informazioni [1], le quali verranno successivamente utilizzate a livello di analisi e come supporto per l'elaborazione di scelte strategiche da parte di un'azienda [2].

Data l'enorme mole di dati normalmente presente all'interno di un DW, una delle problematiche più sentite nel loro utilizzo è il costo di esecuzione delle query che vengono utilizzate per l'estrazione dati [3, 4, 5]: per ovviare a questo problema è possibile introdurre l'utilizzo delle viste materializzate.

L'idea di base è quella di rendere direttamente disponibili all'applicazione che fa uso del sistema DW una pre-elaborazione dei dati di cui necessita; le viste materializzate, disponibili nella maggior parte dei sistemi di database, prevedono di poter memorizzare su supporto fisico, o in alcuni casi in memoria per maggior efficienza, il risultato dell'elaborazione di una query, con lo scopo di poterla utilizzare successivamente senza eseguire nuovamente l'intera elaborazione [7].

Ma non essendo illimitato né lo spazio dove poter materializzare le viste, né il tempo per la loro costruzione [6], ci si pone di fronte ad un ulteriore problema : quello  della scelta su quali viste conviene materializzare.

Riguardo a quest'ultimo problema, in letteratura sono state proposte diverse soluzioni le quali hanno punti di partenza molto diversi tra loro: ad esempio alcune partono dai dati in modo da andare a determinare quali siano le migliori viste da creare per poi andare a ridefinire le query facendo si che queste viste vengano utilizzate, oppure un altro approccio si basa sull'analisi delle frequenza con cui le query vanno ad utilizzare le varie relazioni.

Per quanto riguarda la composizione del nostro sistema DW utilizzeremo il consueto schema a stella, ma tutti i sistemi di viste materializzate potranno essere evoluti per incorporare ulteriori tipi di configurazioni ( fiocco di neve, etc.. )

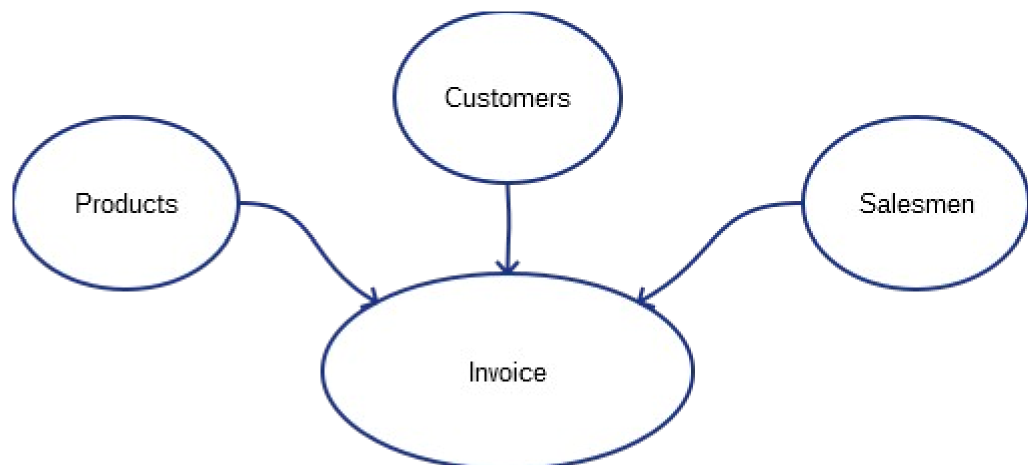



*Illustrazione 1: Esempio schema a stella*

Università Ca'Foscari Venezia

Portiamo un esempio delle caratteristiche dell'effetto della materializzazione delle viste basandoci sullo schema DW [ Illustrazione 1 ];

Abbiamo un semplice schema a stella composto da:

- Una tabella dei fatti “Invoice” contenente i dati di testata e riga delle fatture, contenente la maggior parte delle informazioni, di dimensione $D_i$
- Una tabella delle dimensioni “Product”, contenente le informazioni relative ai prodotti commercializzati, di dimensione $D_p$
- Una tabella delle dimensioni “Customers”, contenente le informazioni inerenti ai clienti, di dimensione $D_c$
- Una tabella delle dimensioni “Salesmen”, contenente le informazioni inerenti agli agenti, di dimensione $D_s$

Al fine dell'esempio che andremo a proporre, consideriamo per semplicità di disporre di un unico raggruppamento per ogni tabella di dimensioni, quindi ad esempio provincia del cliente, categoria merceologica dell'articolo etc..., e di un unico dato da raggruppare nella tabelle dei fatti ( che nomineremo V ).
Supponiamo quindi, di voler andare ad analizzare quali viste materializzare su un DW con la struttura a stella presentata in figura 1 ed avente le caratteristiche precedentemente esposte.
Se volessimo materializzare tutte le possibili viste di cui potremmo aver potenzialmente bisogno per analizzare il nostro DW, avremmo bisogno delle seguenti:

1. Una vista che raggruppi la tabella dei fatti per la categoria del prodotto, avente dimensione:

   $$D_{ip} = D_p$$

   Con $D_{ip}$ pari alla dimensione richiesta per materializzare la vista data dal prodotto cartesiano tra Invoice e Product

2. Una vista che raggruppi la tabella dei fatti per i dati relativi alla clientela, avente dimensione:

   $$D_{ic} = D_c$$

   Con $D_{ic}$ pari alla dimensione richiesta per materializzare la vista data dal prodotto cartesiano tra Invoice e Customer

Università
Ca'Foscari
Venezia

3. Una vista che raggruppi la tabella dei fatti per i dati relativi agli agenti, avente dimensione:

   $$D_{is} = D_c$$

   Con $D_{is}$ pari alla dimensione richiesta per materializzare la vista data dal prodotto cartesiano tra Invoice e Salesmen

4. Una vista che raggruppi la tabella dei fatti per i dati relativi a prodotti e clienti, avente dimensione:

   $$D_{pc} = D_p \text{ X } D_c$$

   Con $D_{pc}$ pari alla dimensione richiesta per materializzare la vista data dal prodotto cartesiano tra Invoice, Product e Customer

5. Una vista che raggruppi la tabella dei fatti per i dati relativi a prodotti ed agenti, avente dimensione:

   $$D_{ps} = D_p \text{ X } D_s$$

   Con $D_{ps}$ pari alla dimensione richiesta per materializzare la vista data dal prodotto cartesiano tra Invoice, Product e Salesmen
6. Una vista che raggruppi la tabella dei fatti per clienti ed agenti, di dimensione:

   $$D_{cs} = D_c \text{ X } D_s$$

   Con $D_{cs}$ pari alla dimensione richiesta per materializzare la vista data dal prodotto cartesiano tra Invoice, Customer e Salesmen

7. Infine di una vista che raggruppi la tabella dei fatti per ciascuna dimensione, di dimensione:

   $$D_{pcs} = D_p \, x \, D_c \, x \, D_s$$

   Con $D_{pcs}$ pari alla dimensione richiesta per materializzare la vista data dal prodotto cartesiano tra tutte le tabelle di dimensione $D_p$ x $D_c$ x $D_s$

Pertanto la dimensione totale delle viste materializzate sarà dato da :

$D_{tot} = D_p + D_c + D_s + D_{pc} + D_{ps} + D_{cs} + D_{pcs}$

Università
Ca'Foscari
Venezia

Il vantaggio delle viste materializzate consiste nel tempo di risposta del sistema; prendiamo per esempio questo set di query:

- Q1: SELECT $R_p$,sum(Val) FROM Invoice JOIN Products GROUP BY $R_p$ ( eseguita 1000 volte )
- Q2: SELECT $R_c$,sum(Val) FROM Invoice JOIN Customers GROUP BY $R_c$ ( eseguita 5000 volte )

Senza viste materializzate avremo un costo totale di esecuzione di 1000 volte la join (raggruppata) tra prodotti e fatture, e 5000 volte la join (raggruppata) tra clienti e fatture.

Stimando il costo di una join, in maniera molto semplicistica, come la moltiplicazione tra le dimensioni delle tabelle coinvolte, abbiamo un costo pari a

$$C_{tot} = D_i \text{ X } D_p \text{ X } 1000 + D_i \text{ X } D_c \text{ X } 5000$$

Mentre se ci trovassimo tutte le viste già materializzate, il costo totale sarebbe pari alla semplice lettura dalle viste già create:

$$C_{tot} = D_{ip} \text{ X } 1000 + D_{ic} \text{ X } 5000$$

Riducendo così drasticamente il costo totale, in quanto il prodotto tra Invoice e dimensioni viene fatto un'unica volta in fase di creazione della vista materializzata.

Torniamo però a sottolineare l'assunzione fatta all'inizio, ovvero quella per cui si è ipotizzato di di non avere raggruppamenti molteplici che coinvolgano le tabelle: se così non fosse infatti vedremmo che lo spazio necessario alla materializzazione di tutte le possibili viste crescerebbe molto rapidmente, così come il tempo per fare la loro materializzazione.

Quindi, pur essendo lo spazio disponibile sul disco potenzialmente molto elevato, nel caso considerassimo tutte le possibili combinazioni di tabelle, lo spazio occupato sarebbe troppo; così come, pur essendo il tempo disponibile per la creazione delle viste potenzialmente elevato, ad esempio facendo elaborazioni notturne, se volessimo materializzare tutte le possibili combinazioni sarebbe troppo poco: ecco perché è di

Università Ca'Foscari Venezia

vitale importanza a livello strategico scegliere quali viste materializzare in modo da rimanere dentro i vincoli del sistema, pur avendo i benefit visti in precedenza dovuti all'aver dati pre costruiti.

Come sviluppo innovativo verrà proposta una funzione semi-automatica per il calcolo delle viste da materializzare in modo da poter successivamente riscrivere le query tenendo a mente che:

1. Dobbiamo mantenere un vincolo su spazio e tempo di esecuzione
2. Dobbiamo cercare di minimizzare il tempo di esecuzione delle query eseguite sul sistema DW, aggiungendo però un parametro configurabile dall'utente, in cui potrà aumentare il peso delle query che per lui sono strategiche
3. Inoltre sarà prevista la possibilità di distribuire le viste materializzate in più tablespace aventi caratteristiche di dimensione e velocità differenti, in modo da incrementarne ulteriormente l'efficienza dell'analisi dei dati del nostro sistema DW.

Università Ca'Foscari Venezia

# 2. Related work

Nelle prossime sezioni introdurremo una serie di algoritmi già presenti in letteratura e una breve introduzione sulla terminologia e metodologia utilizzate per comprenderli e strutturarli.

## 2.1 Terminologia e Metodologia

Partiamo con l'andare a definire la metodologia di selezione delle viste materializzate.
Lo schema che verrà utilizzato sarà il seguente:

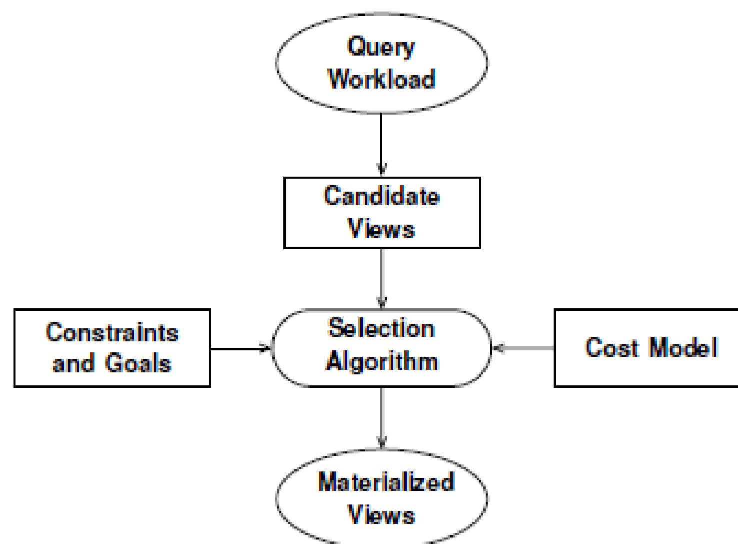


*Illustrazione 2: Selezione delle viste materializzate*

La selezione delle viste da materializzare si basa dunque sui seguenti elementi:

- Query Workload – ovvero i dati storici relativi alle query eseguite sul sistema DW
- Candidate Views – ovvero sia l'assieme di tutte le possibili viste che portano un beneficio
- Constraint and goals – ovvero sia gli scopi e i vincoli ammessi dall'algoritmo di selezione

- Cost model – ovvero sia un modello sulla base del quale andare a calcolare il costo di esecuzione delle query e di creazione delle viste materializzate
- Selection Agorithm – ovvero sia un algoritmo che permette di selezionare le viste da materializzare in base a quanto visto precedentemente.
- Il tutto alla fine genererà un assieme di Materialized Views.

Definiamo, dunque:

- V l'insieme di tutte le possibili viste che è possibile materializzare ( Candidate views )
- M l'insieme delle viste materializzate che andremo ad individuare ( Materialized views )
- $v_i$ l'i-ma vista materializzata
- $C(v_i)$ il costo in termini di spazio della vista i-ma ( la funzione C determinerà il nostro Cost Model )
- Q sarà l'insieme di tutte le query storiche nel nostro DW ( Query Workload )

Per maggior chiarezza è possibile rappresentare V attraverso uno schema a lattice come in figura (schema riferito all'esempio presente nella introduzione – Si veda illustrazione 1):

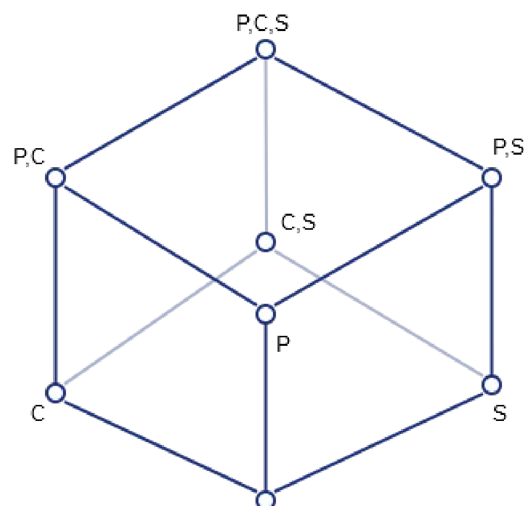


*Illustrazione 3: Lattice delle viste possibili*

Dove :

- Il nodo radice rappresenta la tabella dei fatti raggruppata per Customer (C), Product(P) e Salesmen (S) {(P,C,S}
- Il secondo livello contiene raggruppamenti più specifici per ogni combinazione di 2 delle 3 dimensioni {(P,C),(P,S),(C,S)}.
- Il terzo livello contiene raggruppamenti di una sola dimensione {(P),(C),(S)}
- Infine l'ultimo livello (o chiusura) sarà la semplice tabella dei fatti raggruppata, contenente quindi un unico record {}.

Questo tipo di schema ci aiuterà in particolar modo a seguire l'andamento dell'algoritmo 2.2 che seguirà, in quanto si basa principalmente sulla relazione di dipendenza data dai legami tra i vari livelli;
Infatti se materializziamo la vista che tiene conto di un raggruppamento con due dimensioni, possiamo utilizzarla per ottimizzare tutte le query più “ in basso “ nello schema, o più specifiche, semplicemente utilizzando un raggruppamento sui dati della vista già creata.

Università
Ca'Foscari
Venezia

## 2.2 Un algoritmo greedy per la selezione delle viste materializzate

Il primo, e più semplice, algoritmo che andremo a presentare e a commentare è chiamato "algoritmo greedy per la selezione delle viste materializzate" [14].
Questo algoritmo è un algoritmo greedy il quale andrà a selezionare k delle V possibili viste materializzate in modo da andare a minimizzare il costo totale delle query eseguibili.
L'algoritmo si basa sulla formula matematica [8]:

$$\tau(V, M) = \sum_{i=1}^{|Q|} C(q_i, M)$$

La quale sta a indicare che di deve arrivare a trovare t(V,M) tali per cui la somma dei costi di esecuzione di tutte le query eseguibili, dato un insieme M di viste materializzate, sia minimo.
E' stato dimostrato che questo problema di ottimizzazione è NP-Completo.

Proprio per questo motivo l'algoritmo greedy non andrà a scorrere tutte le possibili soluzioni per andare a selezionare in seguito quella ottimale, ma si baserà sulla seguente assunzione : "ad ogni iterazione si andrà a calcolare il beneficio che ognuna delle viste da materializzare rimaste può portare, e la vista che porterà ad un maggiore beneficio verrà materializzata."
L'algoritmo si compone di un numero fissato k di iterazioni, fino ad esaurimento dello spazio disponibile oppure al completamento di tutto il lattice di possibili viste da materializzare; ad ogni iterazione si andrà ad aggiungere all'insieme M una vista (quella appunto che in quel preciso istante porta un maggior beneficio).

Come si può evincere, il cuore di questo algoritmo risiede nel calcolo dei benefici che le varie viste materializzabili possono portare. Definiamo con B(vi,M) il beneficio che l'aggiunta della vista materializzabile i-ma a M porterebbe.
Il procedimento per definire questo beneficio è schematizzato attraverso il seguente algoritmo:

- per ogni vista $w <= v$ ( ovvero con w o uguale a v o un suo discendente ), prendiamo $v_m$ la vista in M con costo $C(v_m)$ minimo, tale per cui $w <= v_m$.
$B_w = \max\{C(v_m) – C(v),0\}$
- il beneficio B(v,M) sarà definito come $B(v,m)= \Sigma_{w<=v} B_w$

In altre parole, il beneficio di aggiungere una vista al sistema è dato da quanto incrementa le prestazioni sulle query sulla stessa vista, e delle possibili viste da essa derivabili, ad esempio la creazione ipotetica di una vista che incorpori clienti(cutomers) va ad avere un beneficio sul sistema dato dal miglioramento di prestazioni relativo alle query su clienti, ma anche su clienti/articoli, in quanto parte dei dati è già calcolata.

Università Ca'Foscari Venezia

### 2.2.1. Esempio illustrato dell'algoritmo

Passiamo ora a dare una dimostrazione grafica dell'algoritmo basandoci sull'esempio di base riportato all'esempio 1 :

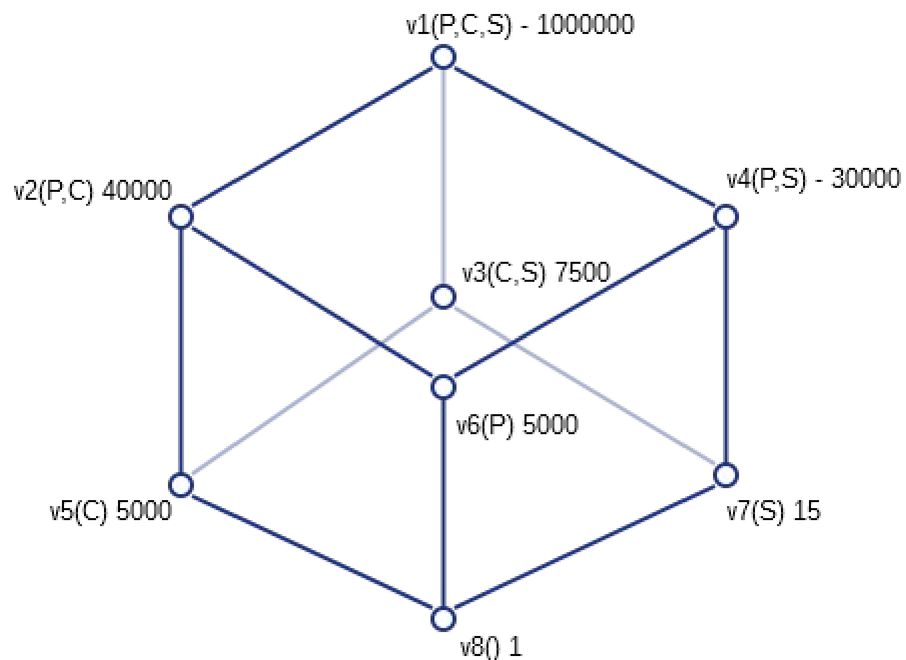


*Illustrazione 4: Lattice iniziale*

Dove siamo andati indicare le varie dimensioni (o costi) delle varie viste che potrebbe essere materializzate.

Consideriamo come punto di partenza la prima interazione M = { v1 }, calcoliamo per ogni vista il benefit $B(v_i)$.

Per v2(P,S) la situazione sarà la seguente, dove i nodi segnati in rosso non sono direttamente coinvolti da v2, mentre i nodi segnati in verde sono quelli già inseriti all'interno dell'insieme M.

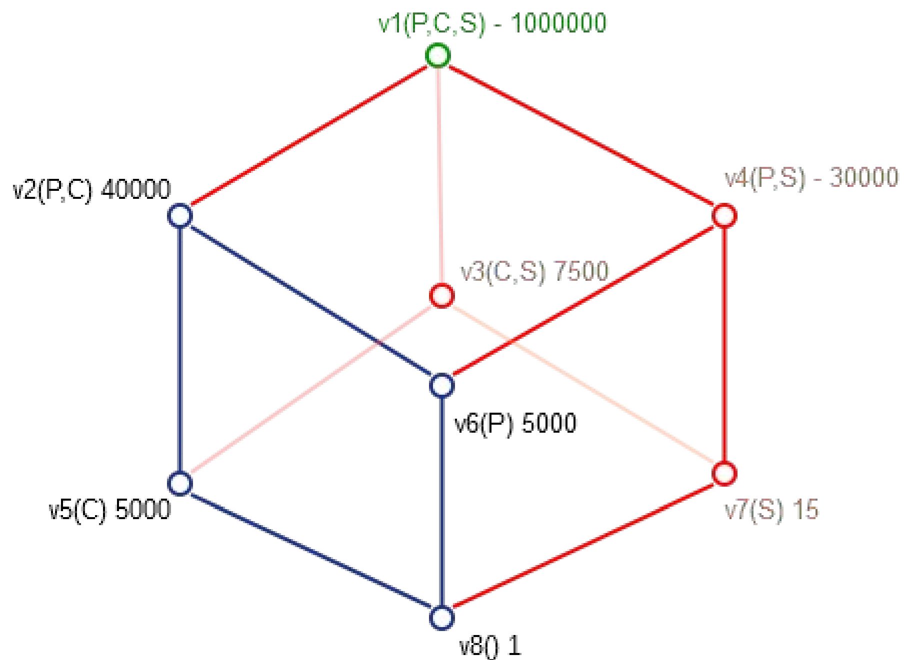



*Illustrazione 5: Lattice 1 - v2 con M={v1}*

In questo caso le viste coinvolte da v2 saranno le seguenti:
w = v2,v5,v6,v8

Dato che l'unica vista appartenente a M è v1, per ogni v di w l'unico costo è quello di v1:
vm1 = v1 = 1.000.000

Il beneficio è dato dalla differenza tra C(v2) ed il minimo costo appartenente a M di ogni v di w, ma essendo solo v1 presente in M avremo:
Bw = max { 1.000.000 – 40.000 , 0 } = 960.000 per ogni v appartenente a w.

Il beneficio di aggiungere v2 a M è dunque:
B(v2,M)=960.000 x 4 = 3.840.000

Università
Ca'Foscari
Venezia

Che andrebbe ad indicare che l'aggiunta di v2 a M va ad incrementare tutte quelle query che agiscono su v1 inerenti a P e/o C, ovvero 4 volte il beneficio di avere v2 materializzata.

Proviamo ora a calcolare che beneficio porterebbe l'inserimento della vista V3 all'interno di M :

Il caso è del tutto similare al precedente, con:
w = v3,v5,v7,v8
vm1 = v1 = 1.000.000
Bw = max { 1.000.000 – 7,500, 0 } = 992.500 ( per v3,v5,v7 e v8 )
B(v3,M)=992.500 x 4 = 3.970.000

Da cui deduciamo che tra le due viste materializzabili prese finora in considerazione, la materializzazione di v3 porterebbe il maggiore beneficio.

Di seguito riportiamo brevemente l'analisi dei benefici portati da ogni vista materializzabile dopo la prima iterazione (k=1):

B(v2,M) = 3.840.000
B(v3,M) = 3.970.000
B(v4,M) = 3.880.000
B(v5,M) = 1.990.000
B(v6,M) = 1.990.000
B(v7,M) = 1.999.970
B(v8,M) = 999.999

Pertanto l'algoritmo sceglierà di materializzare la vista V3 in quanto è quella che porta un maggior beneficio.

Alla seconda iterazione dell'algoritmo, considerando di partire ad analizzare i benefici partendo dalla vista v2, avremo la seguente situazione :

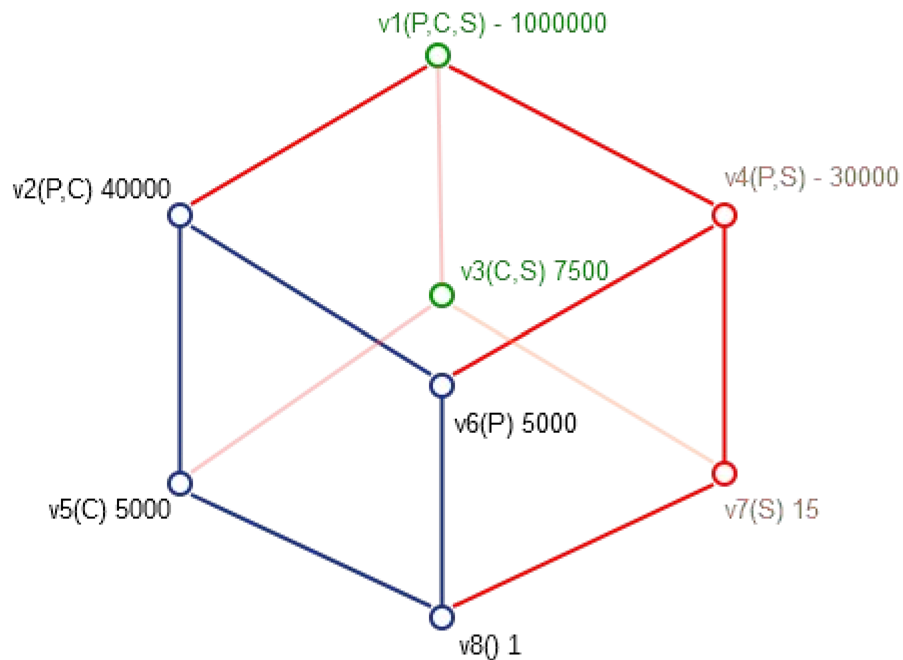


*Illustrazione 6: Lattice 2 – v2 con M={v1,v3}*

A questo punto, con M aggiornato a {v1,v3}, ricominciamo ad analizzare per K=2 quale sia la vista che porta il maggior beneficio.

Iniziamo da v2.

Avremo, come per l'esempio precedente:
w = v2,v5,v6,v8

Calcoliamo il beneficio che la stessa v2 porterebbe alla materializzazione di v2
vm1 = C(v1) = 1.000.000
Bw= max { 1.000.000 – 40.000 ,0 } =960.000
In quanto totalmente disgiunta dall'aggiunta di v3 al sistema;

Calcolando invece il beneficio che v5 porterebbe alla materializzazione di v2 invece la situazione varia, infatti avremo:
vm1 = C(v1) = 1.000.000
vm2 = C(v3) = 7.500

Dato che il minimo costo dei predecessori di v5 è quello di v3 e non di v1, il beneficio è il seguente:
Bw = max { 7.500 – 40.000 , 0 } = 0
In altre parole, non porta nessun beneficio alle query su v5 la materializzazione di v2 in quanto si può usare più efficientemente la già materializzata v3.

Proseguendo, per v6:
vm1 = C(v1) = 1.000.000
Bw = max { 1.000.000 – 40.000 , 0} = 960.000

Per v8
vm1 = C(v1) = 1.000.000
vm2 = C(v3) = 7.500
Bw = max { 7.500 – 40.000 , 0 } = 0

Per un totale di
B(v2,M) = 1.920.000

Vediamo che rispetto alla prima iterazione il beneficio di aggiungere v2 ad M è calato, infatti per tutte le query che potrebbero usare v3 già materializzata il beneficio si riduce.

Diamo ora un riassunto dei benefici calcolati:

B(v4,M) = 1.940.000
B(v5,M) = 5.000
B(v6,M) = 997.500
B(v7,M) = 14.970
B(v8,M) = 7.499

Al termine della seconda iterazione dunque decidiamo di materializzare v4.

Andiamo a dare solo i risultati della terza e quarta iterazione:

per k=3

B(v2,M) = 960.000
B(v5,M) = 5.000
B(v6,M) = 25.000+2.500=27.500

Università Ca'Foscari Venezia

B(v7,M) = 14.970
B(v8,M) = 7.499

aggiungendo dunque v2 ad M.

per k=4

B(v5,M)=5.000
B(v6,M)=27.500
B(v7,M)=14.970
B(v8,M)=7.499

aggiungendo dunque v6 ad M.

con k = 4 l'algoritmo termina qui, materializzando v2,v3,v4 e v6.

Nonostante questo algoritmo trovi delle buone soluzioni, non ne offre di ottimali: infatti soffre di diversi punti deboli.
Primo tra tutti è quello relativo al tempo di ricerca di una soluzione ottimale, l'algoritmo scandisce molte volte il lattice in cerca della soluzione che porta il maggior beneficio in quel momento preciso; il quale risulta un costo computazionale $O(kn^2)$.
Inoltre non tiene conto degli attributi delle relazioni, ma solo del costo delle letture da una relazione piuttosto che da un altra, infatti normalmente crea viste semplicemente date dalle giunzioni tra le tabelle.
Altri importanti punti deboli di questo algoritmo risiedono nel fatto che non si tiene conto nè della frequenza con cui le query vengono eseguite né del tempo necessario alla generazione delle viste materializzate, a meno di non calcolare dinamicamente il costo C di ciascuna vista in base a tali condizioni.
Sebbene esistano delle ottimizzazioni per questo algoritmo (dette "*Polynomial Greedy Algorithm*"), le quali prevedono un sensibile miglioramento delle performance dal punto di vista computazionale andando a ridurre il numero di viste da prendere in considerazione, noi andremo a concentrarci su algoritmi più complessi che partono da una base diversa: invece che partire da una possibile combinazione di tabelle che

andrebbero a ottimizzare le query sul sistema DW, questi partono dalle query da utilizzare in modo da poter andare a ottimizzare la gestione delle viste da materializzare con conseguente possibilità di andare a indicare un peso o una priorità di esecuzione di ciascuna query.

## 2.3 Data Warehouse Configuration Problem

I limiti dell'algoritmo presentato al paragrafo 2.2 sono stati affrontati tramite l'algoritmo *"Data Warehouse Configuration"*[10], il quale terrà conto sia del tempo di costruzione delle viste da materializzare, sia della frequenza con cui le query vengono eseguite.

Integriamo quindi la nostra lista terminologica con questi nuovi concetti :

- Sia la funzione $f(q_i,M)$ la funzione andrà a indicare la frequenza di esecuzione di un query $q_i$ con l'insieme delle viste materializzate M, la quale potrebbe anche tener conto del peso della query $q_i$ nell'insieme di tutte le query eseguibili
- Sia l'insieme $T = \{t_1, t_2, \ldots, t_q\}$ l'insieme di tutti i tipi di aggiornamento che possono avvenire all'interno della base dati (inserimento, aggiornamento, cancellazione)
- Sia la funzione $C_t(t_i)$ la funzione che andrà a indicare il costo di aggiornamento di $t_i$
- Sia la funzione $f_t(t_i)$ la funzione che andrà a indicare la frequenza con cui ti viene aggiornata

Pertanto il nostro scopo diventa quello di minimizzare il costo pesato delle query eseguibili all'interno del nostro insieme di viste materializzate, unitamente all'ottimizzazione del loro tempo di aggiornamento.

Per fare ciò partiremo dall'andare a definire l'insieme M per cui avremo che sia il tempo totale di esecuzione delle query t(V,M) e il tempo totale di aggiornamento delle viste tt(V,M) [24] sia minimo:

$$t(V,M) = \sum_{i=1}^{|Q|} f(q_i,M)$$

$$tt(V.M) = \sum(_{i=1}^{|T|} ft(t_i,M)Ct(t_i,M)$$

Per unificare queste due regole sommeremo t(V,M) e tt(V,M) pesando tramite il parametro c l'importanza che ha

l'aggiornamento delle viste rispetto al tempo di esecuzione delle query.

Sia dunque
$\tau(V,M) = t(V,M) + c*tt(V,M)$

La funzione da minimizzare che sarà lo scopo del nostro algoritmo.

L'algoritmo si basa sulle transizioni di stato: consideriamo uno stato iniziale $s_0$, vediamo quali sono le possibili transizioni di stato ammesse dall'algoritmo, e finché si possono creare nuovi stati, memorizzeremo stato e relativo costo; lo stato con costo minimo sarà la nostra ottimizzazione.

Abbiamo anche bisogno di un modo grafico di rappresentare le query $q_i$; utilizzeremo per far ciò dei “multiquery graph”. Questi grafici rappresentano join e selezioni in questo modo:

1. I nodi del grafico sono le relazioni che appaiono tra le viste V
2. Ogni join appartenente ad una vista di V è rappresentata da un arco che unisce due nodi, come etichetta avrà la condizione di join
3. Ogni selezione è un arco che unisce un nodo a se stesso con come etichetta il predicato di selezione

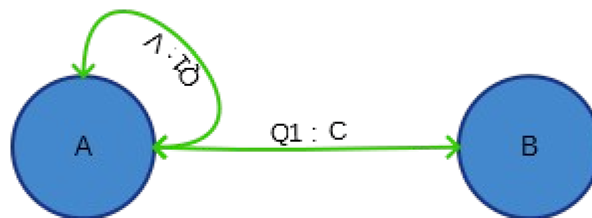


*Illustrazione 7: Multiquery graph, esempio*

Rispetto alla illustrazione, è rappresentata una query Q1 che interessa due relazioni ( “A” e “B” ), unite da una join con predicato “C” e seleziona il dato “V” di “A”, sintatticamente è la seguente query:

Q1: SELECT V FROM A JOIN B ON C;

Diamo la possibilità di avere tre possibili transizioni di stato:

**a) Selection edge cut**

Se esiste una selezione *e* in un nodo *R* appartenente alla vista *V*,
a.1) se è anche l'unica selezione della vista *V* in *R*, allora si può creare una nuova vista $V_i$ che sostituisce la selezione, rimuovendo la selezione *e* e rimpiazzando ogni occorrenza di *V* con *Vi*,
a.2) in alternativa si possono unire le due selezioni di V in R creando $V_i$.

In altre parole selection edge cut dice che si possono sostituire le selezione di una vista, con un'altra vista senza tali selezione, che va semplicemente a leggere la tabella.

Seguitiamo ad indicarne un esempio:

Supponiamo di avere le seguenti query:

Q1 = SELECT V FROM A;
e
Q2 = SELECT V1, V2 FROM A;

Rappresentiamole attraverso un multiquery graph:

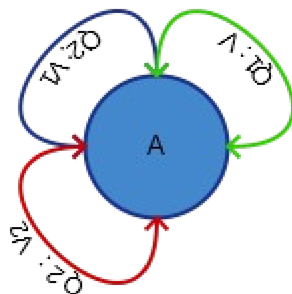


*Illustrazione 8: Multiquery graph: Selection edge cut 1*

Dopo "selection edge cut" di Q1 avremo la seguente situazione:

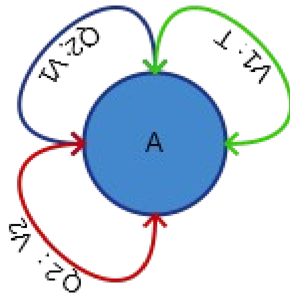


*Illustrazione 9:*
*Multiquery graph:*
*Selection edge cut 2*

La procedura di selection edge cut ci consente anche di riscrivere semplicemente le query, infatti man mano che “tagliamo” le selezioni, basta infatti sostituire alla selezione limitata sulla tabella, la selezione sulla vista corrispondente.

V1=SELECT * FROM A
Q1=SELECT V FROM V1
Q2=SELECT V1, V2 FROM A

Nel caso di selection edge cut avremo una occupazione di spazio normalmente minore, perdendo una vista materializzata con la velocità di esecuzione quasi invariata, infatti la velocità di lettura da una vista con un solo campo o da una tabella con molti campi è normalmente indifferente.

**b) Join edge cut**

Nel caso ci sia un arco  e che unisce due nodi V : Fp, si può eliminare costruendo un nuovo grafico come segue:

a) Se la rimozione di e non divide il grafico in due componenti distinte, allora sostituiamo V con $_{V_i}$ ed eliminiamo l'arco di join
b) Altrimenti sulla prima componente sostituiamo con $V_{i1}$ e sulla seconda con $V_{i2}$ ;se quello che rimane è un componente senza archi, allora rinominiamo anche l'arco come $V_{i1}$ : T , $V_{i1}$ sarà una semplice relazione.

Il significato del “join edge cut” è che da una unica vista materializzata che esegue una join, possiamo crearne due, una per ogni parte di join e demandare alla query l'esecuzione della join.

Un esempio è il seguente:

Consideriamo due nodi A e B, uniti da una join con condizione C, con questa query che dà origine alla vista materializzata iniziale:
Q1: SELECT V1 FROM A JOIN B ON C

ed il seguente multiquery graph corrispondente:

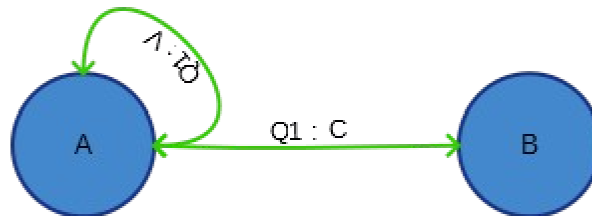


*Illustrazione 10: Multiquery graph: Join edge cut 1*

Dopo l'applicazione di “join edge cut” avremo la seguente situazione:

*Illustrazione 11: Multiquery graph: Join edge cut 2*

La riscrittura della query segue la creazione delle due viste distinte in questo modo:
V1=SELECT V1 FROM A

V2=SELECT * FROM B
Q1: SELECT V1 FROM V1 JOIN V2 ON C

Lo spazio occupato normalmente diminuisce, con la creazione di una o due nuove viste separate, aumenta sicuramente la frammentazione: infatti se avessimo materializzato Q1 avremmo avuto una tabella unica che conteneva tutto il risultato dato dalla join, mentre nel caso in esame si vengono a separare due viste distinte di dimensione singola sicuramente inferiore dalla dimensione della join.
Il tempo di esecuzione invece aumenta notevolmente, infatti non avremo più una vista predisposta per Q1, ma la stessa query dovrà eseguire la computazione ogni volta che sarà eseguita.

**c) View merging**

Se esistono due viste V1 e V2 con gli stessi nodi e con predicati l'uno che implica l'altro, si possono fondere in un'unica vista V3, avente un solo arco al posto di due ogni qualvolta si trovino due archi uguali, uno per V1 e uno per V2 e come condizioni la più generica delle condizioni di selezione e join ( quella che implica l'altra )

Il significato è che se esistono due viste che condividono anche in parte un predicato di join, se ne può creare una più generica che tiene conto del predicato meno restrittivo e riscrivere le query sulla base di questa nuova vista materializzata.

L'esempio è il seguente:

V1=SELECT A.K1 FROM A JOIN B ON A.id=B.id AND a.test=true
V2=SELECT B.K2 FROM A JOIN B ON A.id=B.id

Con il seguente query graph:

*Illustrazione 12: Multiquery graph: View merging 1*

Università Ca'Foscari Venezia

Dopo l'applicazione di View – Merging avremo:

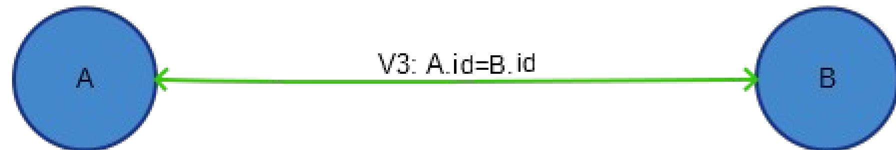


*Illustrazione 13: Multiquery graph: View merging 2*

Con la riscrittura delle query che avviene in questo modo:

V3=SELECT A.K1,B.K2,B.test FROM A JOIN B ON A.id=B.id
V1=SELECT V3.K1 FROM V3 WHERE V3.test=true
V2=SELECT V3.K2 FROM V3

View Merging sicuramente diminuisce lo spazio totale, eliminando una parte di dati replicati; le query in generale saranno più lente, in quanto almeno una di esse esegue una selezione su una vista e non è invece totalmente materializzata.

In generale dunque ogni iterazione dell'algoritmo riduce lo spazio totale utilizzato creando viste più generiche, allo scotto di aumentare normalmente il tempo totale di esecuzione delle query, infatti si parte dal fatto che tutte le query abbiano una propria vista materializzata, e poi man mano si tolgono delle particolarità di ciascuna query, unificandole.
La situazione finale dell'algoritmo sarà l'avere le sole tabelle materializzate.

# 3. Algoritmo proposto

L'algoritmo proposto è una variante di quanto già indicato nel capitolo 2.3 e ne va a tentare di migliorare l'aspetto spaziale: pur rimanendo lo scopo di trovare la combinazione di viste materializzate che minimizzino il tempo totale, pesato, delle query eseguite, andiamo però ad aggiungere un vincolo: ammettiamo che il sistema preveda più spazi per la memorizzazione delle viste materializzate, $K_1,K_2,\dots,K_n$ con velocità di accesso decrescenti: $V(K_1)<V(K_2)<\dots<V(K_n)$.

Vogliamo a questo punto stabilire la combinazione di viste materializzate, residenti ognuna in uno dei K spazi disponibili, che minimizzi il tempo totale, pesato, delle query eseguite, tenendo conto della differenza di prestazioni tra i vari tablespace.

Sempre più infatti i sistemi server prevedono dischi o memorie con capacità di elaborazione più varie: dischi o memorie veloci per le elaborazioni che richiedono maggiori prestazioni e normalmente leggono da dati meno numerosi, dato che hanno dimensioni contenute, mentre dischi più capienti per elaborazioni che richiedono meno velocità e che normalmente agiscono su più dati.

Il vincolo aggiuntivo varierà la nostra funzione di calcolo del beneficio dato da una vista che diventa materializzata, infatti non solo determina la possibilità che sia creata ( deve essere disponibile dello spazio per la sua creazione ), ma anche che questa non infici la possibilità di creazione di due o più viste più piccole, che magari da sole non portavano lo stesso beneficio, ma che sommate hanno un beneficio maggiore.
In particolare per l'algoritmo esaustivo (2.2) ne aumenta incredibilmente la complessità in quanto, per ogni possibile cambio di stati, deve anche andare a verificare l'occupazione di spazio del nuovo stato, ottimizzando il più possibile la posizione nei vari tablespace delle viste materializzate ( problema di ottimizzazione riconducibile al problema np completo del knapsack )

Aggiungiamo dunque le seguenti definizioni:
Siano $K_1, K_2, \ldots, K_n$ le dimensioni di n spazi disponibili con $K_1 < K_2 < \ldots < K_n$,
Sia $V(K_i)$ la velocità ( relativa ) dello spazio $K_i$, potremmo definirla come Kb/s se si tratta di una vera e propria partizione del disco, oppure semplicemente un valore che determina quanto la partizione $K_i$ sia veloce in confronto con gli altri spazi disponibili ( ad esempio con due spazi disponibili, sapendo che il primo risulta 2 volte più rapido del secondo, possiamo anche indicare semplicemente $V(K_1) = 2$ , $V(K_2) = 1$;
Sia inoltre MK una disposizione delle viste materializzate M sulle K partizioni disponibili; sia $k(q_i,MK)$ la maggiorazione del costo di esecuzione di qi su MK, intesa, ad esempio, come il minimo tra tutti i $V(K_i)$ dove $K_i$ è utilizzato in qualche parte di $q_i$.

Dovremmo trovare dunque l'insieme MK per cui sia:

$t(V,MK) = \Sigma_{i=1}^{|Q|} f(q_i,MK)C(q_i,MK)/k(q_i,MK)$ , ovvero il tempo totale di esecuzione delle query tenendo conto di dove sono presenti, ma anche
$tt(V,MK) = \Sigma_{i=1}^{|T|} ft(t_i,M)Ct(t_i,M)/kt(t_i,MK)$ , ovvero il tempo totale di aggiornamento delle viste sia minimo.

Per unificare le due sopra citate regole, sommeremo t(V,M) e tt(V,M), pesando con un parametro c l'importanza che ha l'aggiornamento delle viste rispetto al tempo di esecuzione delle query.
Sia dunque $\tau(V,MK) = t(V,MK) + c*tt(V,MK)$ la funzione che determina come strutturare il nostro algoritmo.

Possiamo scrivere l'algoritmo esaustivo come segue:

Partiamo da uno stato $s_0$ che rappresenta la materializzazione di tutte le query eseguite e da $k_0$ che consiste nella disposizione di tutte le viste su un qualunque tablespace.
Nota: La funzione Costo è valorizzata ad infinito o invalido qualora lo spazio disponibile non sia sufficiente.

$C[s_0,k_0]=Costo(s_0,k_0)$
$L1= \{s_0\}; L2=\{\}$
while (L1 != {} )

```
  s=pop(L1)
  for s' in Transazioni(s,s’)
    if s’ non appartiene a L1 U L2 then
      for k' in possibili posizioni delle viste s' in K
        C[s',k']=Costo(s’,k')
      end for
      L1 = L1 U {s’}
    end if
  end for
  L1 = L1 - {s} ; L2 = L2 U {s}
end while
return (s,k) avente C[s,k] minimo
end.
```

Naturalmente tale algoritmo esaustivo, pur applicabile, comporta un tempo di esecuzione molto elevato. Questo normalmente non è un problema dato che la ricostruzione delle viste su un DW è eseguita molto di rado; ma ne dobbiamo offrire una versione che, pur perdendo la sicurezza di ottenere la soluzione migliore, sia più efficiente.

Esistono ovviamente anche ulteriori algoritmi per la ricerca delle viste da materializzare alcuni riassunti ad esempio in [20], di cui rimandiamo alla bibliografia ([16],[17],[18],[19],[21],[23]) per ulteriori approfondimenti

## 3.1 Euristica proposta

Come prima cosa ci concentriamo sullo stato iniziale $s_0$:
in generale si tratta dello stato in cui tutte le query giacciono su un vista preposta; questo garantisce il minor tempo di esecuzione possibile ( ho tutto materializzato ), ma prevede anche l'occupazione massima di spazio disponibile ed il massimo tempo di aggiornamento e creazione delle viste;
La prima considerazione quindi è di andare a non considerare le combinazioni su K per tutti quei stati in cui “ la somma dell'occupazione di spazio di tutte le viste v' appartenenti a V è maggiore della somma di tutto lo spazio disponibile, inoltre non valutiamo nemmeno il costo dello stato s' per cui anche solo una vista v' appartenente a V ha uno spazio maggiore del massimo spazio disponibile in K”, eliminando quindi a priori alcune combinazioni non possibili, questo lo facciamo quando calcoliamo le combinazioni possibili dello stato s' su k, ritornando l'insieme vuoto.
Aggiungiamo inoltre una funzione CostoMinimo(s') che considera il costo che avrebbero tutte le viste se posizionate su K0; se questo costo è maggiore del minimo costo trovato finora, non continuare la ricerca.
L'algoritmo ottimizzato ( non ancora euristico ) diventa dunque:

```
C[s0,k0]=CostoMinimo(s0)
L1= {s0}; L2={}
while (L1 != {} )
  s=pop(L1)
  for s' in Transazioni(s,s’)
    if s’ non appartiene a L1 U L2 then
       C[s',km]=CostoMinimo(s')
       if (C[s',km] < minimo(C[s,k])) then
         for k' in possibili posizioni delle viste s' in K - {km}
           C[s',k']=Costo(s’,k')
         end for
       end if
       L1 = L1 U {s’}
    end if
  end for
  L1 = L1 - {s} ; L2 = L2 U {s}
```

Università
Ca'Foscari
Venezia

end while
return (s,k) avente C[s,k] minimo
end.

Come soluzione euristica al problema aggiungiamo una funzione Beneficio, B(s,k) che ne determina l'importanza, ovvero quanto vantaggio porta tale configurazione;
La funzione B(s,k):
1) Dipende direttamente dal Costo(s,k).
2) Dipende dall'occupazione totale di spazio : più una configurazione occupa spazio, più potrebbe inficiare la creazione di ulteriori viste.

In altre parole una configurazione che occupa poco spazio, ma che porta un gran beneficio è la migliore, se occupa lo stesso spazio, ma con beneficio minore è peggiore, ma anche una configurazione occupa molto spazio e porta lo stesso gran beneficio ha il medesimo valore.

Quindi la definizione della funzione B(s,k) è la seguente

B(s,k) = min { (Costo senza nessuna vista - Costo(s,k)) / Costo senza nessuna vista , (SpazioDisponibile(K)-Occupazione(s,k))/SpazioDisponibile(K) }

Quindi una configurazione che migliora le performance del 10% ci aspettiamo occupi circa il 10% dello spazio totale disponibile, se ne occupa una porzione maggiore, allora lo spazio dà il limite al valore, altrimenti è il beneficio inteso come rapporto tra costo con la vista materializzata diviso il costo senza tale vista che ne limita il valore.

Basandoci sulla funzione quindi andiamo a “tagliare” tutte quelle configurazioni che non portano benefici sotto una soglia Vmin costante.

L'aumentare di Vmin ci consentirà di avere un algoritmo più efficiente che però elimina alcune soluzioni possibilmente ottimali, al ridursi di Vmin saranno controllate più combinazioni, ma molte che non porteranno mai risultati ottimali.

Usiamo anche una funzione ValoreMinimo(s) che calcola la funzione valore in cui tutte le viste giacciono nella partizione più veloce.

Il “tagliare” incide nell'algoritmo in questo modo:

```
C[s0,k0]=CostoMinimo(s0)
L1= {s0}; L2={};
while (L1 != {} )
  s=pop(L1)
  for s' in Transazioni(s,s’)
    if s’ non appartiene a L1 U L2 then
        V[s']=ValoreMinimo(s')
        if V[s']>=Vmin
        then
        C[s',km]=CostoMinimo(s')
        if (C[s',km] < minimo(C[s,k])) then
          for k' in possibili posizioni delle viste s' in K - {km}
            C[s',k']=Costo(s’,k')
          end for
         end if
       end if
       L1 = L1 U {s’}
    end if
  end for
  L1 = L1 - {s} ; L2 = L2 U {s}
end while
return (s,k) avente C[s,k] minimo
end.
```

Possiamo inoltre anche fare un altro tipo di considerazione sulle transazioni:
- Selection edge cut:
a) Caso in cui viene sostituita la vista rimuovendo la restrizione sulla selezione:
In questo caso l'occupazione di spazio non potrà che aumentare ( rimuoviamo una restrizione ), mentre il costo non si ridurrà molto in quanto il tempo di selezione su una vista che tiene un campo o l'intera relazione non varia di molto
b) Caso in cui viene sostituita la vista raggruppando le selezioni: unificando due selezioni l'occupazione di spazio subisce una

forte variazione decrescente, anche il costo totale diminuisce data la sua componente dovuta al tempo di aggiornamento e creazione.

- Join edge cut:

In questo caso si possono avere delle viste più piccole, l'occupazione totale di spazio delle viste normalmente diminuisce, mentre il costo normalmente aumenta

- View merging:

Normalmente è il caso in cui l'occupazione subisce una variazione più brusca e, normalmente, anche il costo decrementa dato che si riduce la componente dovuta al tempo di aggiornamento e costruzione delle viste.

Detto questo quindi possiamo utilizzare tali dati per ottimizzare anche la selezione delle transizioni di stato in questo modo:

Se la transizione di stato che ha portato allo stato prima(s) è stata scartata perché troppo costosa in termini di occupazione totale, allora non consideriamo neppure le transizioni di tipo “Selection edge cut (a)”, mentre se il limite è dovuto al costo totale di esecuzione troppo elevato, allora non consideriamo “Join edge cut”.
Diamo il maggior peso in ogni caso a “View merging” e “Selection edge cut (b)” che portano il maggior beneficio sia come occupazione che come costo.

Università Ca'Foscari Venezia

## 3.2 Esempio dell'algoritmo proposto

Immaginiamo tre dischi ( o server distinti ) l'uno che ha una latenza di accesso di 100 (K2), uno di 10 (K1) e uno di 1 (K0).

Siano dunque:

$K_0$ = 1000 , $V(K_0)$ = 100
$K_1$ = 5000 , $V(K_1)$ = 10
$K_2$ = 10000, $V(K_2)$ = 1

Avremo il seguente lo star schema del nostro DW:

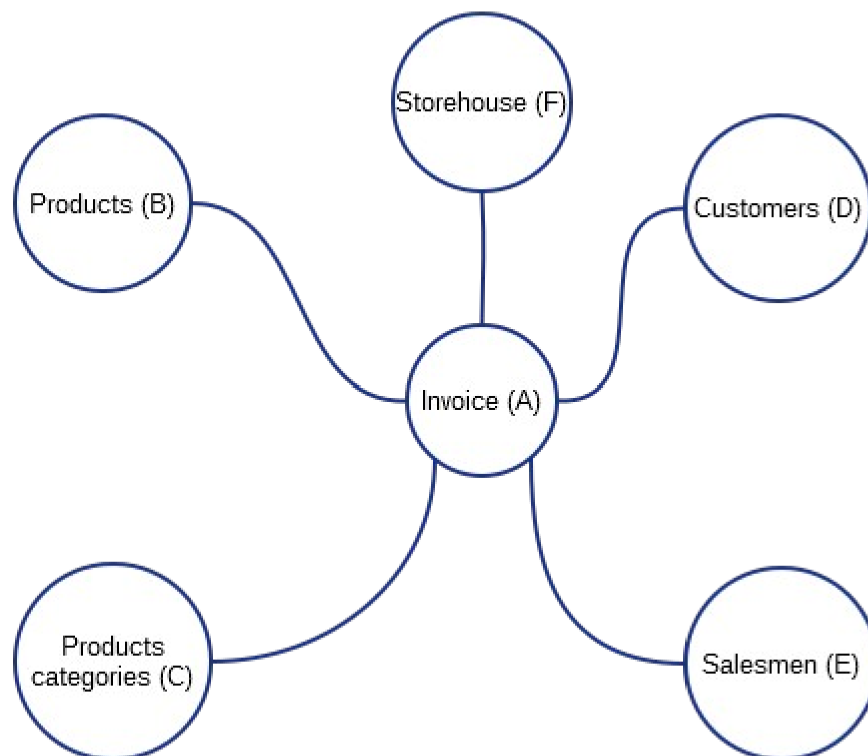


*Illustrazione 14: Star Schema algoritmo proposto*

Consideriamo le seguenti 3 query:

Q1)
SELECT sum(A.qta)

FROM A
INNER JOIN B ON A.Prod_id=B.Prod_id AND
B.Prod_color='GREEN'
INNER JOIN C ON A.ProdCat_id=C.ProdCat_id AND
C.Cat_Name='CAT'
INNER JOIN D ON A.Customer_id=D.Customer_id AND
D.Customer_Nation='IT'

Q2)
SELECT sum(A.val)
FROM A
INNER JOIN B ON A.Prod_id=B.Prod_id AND
B.Prod_color='GREEN' AND B.Prod_Active=true
INNER JOIN E ON A.Sale_id=E.Sale_id AND E.Sale_name='SALE'

Q3)
SELECT sum(A.val)
FROM A
INNER JOIN B ON A.Prod_id=B.Prod_id AND B.Prod_Active=true
INNER JOIN F ON A.Storehouse_id=F.Storehouse_id AND
F.Storehouse_cat='31059'

Avremo il seguente multiquery graph s0:

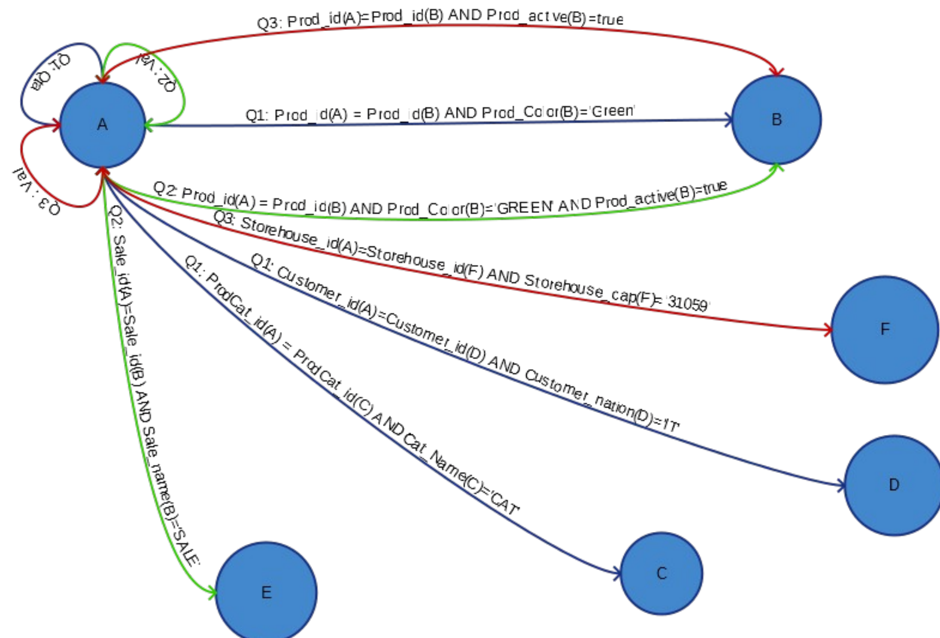


*Illustrazione 15: Esempio proposto 1 - s0*

Università Ca'Foscari Venezia

Partendo dalla situazione iniziale immaginiamo a questo punto 3 viste:
Q1,Q2 e Q3 e le query avverranno tramite selezione dei campi ( Qta e Val) su tali viste.

Consideriamo anche i seguenti costi:
- Costo di A: 100
- Costo di A ( solo qta) : 90
- Costo di A ( solo val) : 90
- Costo di B(prod_active=true): 10
- Costo di B(prod_color='Green'): 7
- Costo di B(prod_color='Green' AND prod_active=true): 6
- Costo di C(Cat_name='CAT'): 2
- Costo di D(Customer_nation='IT'): 4
- Costo di E(Sale_name='SALE'): 2
- Costo di F(Storehouse_cap='31059'): 3

- Costo di esecuzione di Q1,Q2 e Q3 senza viste materializzate: 4788

Infine usiamo le seguenti costanti: (Vmin=2%, una frequenza di 1,ed un peso di aggiornamento  c di 0)

Calcoliamo il costo di $s_0$, ammettendo tutte le viste sulla partizione più rapida;
per semplicità di esempio, per simulare il calcolo del costo utilizziamo la moltiplicazione dei costi di tutte le tabelle coinvolte, vedremo in un capitolo successivo come poter calcolare in maniera corretta il costo di esecuzione ed aggiornamento:

Costo($s_0$,$k_0$) = Costo(Q1,$k_0$) + Costo(Q2,$k_0$) + Costo (Q3,$k_0$) =
100*7*2*4/100 + 100*6*2/100 + 100*10*3/100 =
5600/100+1200/100+3000/100=98

Passando alle transazioni di stato:

Possiamo utilizzare la regola
- Selection Edge cut: rimuovendo le selezioni Q1:Qta, Q2:Val, Q3:Val.

Proviamo per Q1:Qta, dato che non è l'unica selezione di Q1 su A, dobbiamo sostituire Q1 con V1, con V1 equivalente alla selezione della Qta su Q1.

Otteniamo il seguente schema:

*Illustrazione 16: Algoritmo proposto 2*

Vediamo che Q1 non è più citato, al suo posto è presente V1.
Vediamo ora se il guadagno supera la soglia del minimo di resa:
Costo(s1,km) = 90*7*2*4/100 + 100*6*2/100 + 100*10*3/100 = 5040/100+1200/100+3000/100=92,4
Tenendo conto della nostra costante a zero, l'occupazione di spazio per il calcolo del costo minimo è ininfluente, per cui:
V[s1] = (98-92,4)/98= 5,7%
Superando la soglia del 2%.

Il costo minimo è inferiore del minimo costo calcolato finora, quindi si può passare a impostare nelle varie partizioni le viste V1,Q2 e Q3.

La combinazione minima è V1 in K3, V2 e V3 in K2, nessuna vista in K1.

Quindi C(s1,K) =
5040/1+1080/10+3000/10=5040+108+300=5448

Proseguendo con Q2 avremo:
Costo(s2,km) = 90*7*2*4/100 + 90*6*2/100 + 100*10*3/100 =
5040/100+1080/100+3000/100=91,2
con
V[s2] = (92,4-91,2)/9,24= 1,2% che non soddisfa la nostra soglia di 2% di guadagno minimo, quindi non calcoliamo neanche il costo delle combinazioni.

Infine con Q3 avremo:
Costo(s3,km) = 90*7*2*4/100 + 100*6*2/100 + 90*10*3/100 =
5040/100+1080/100+2700/100=88,2
con
V[s3] = (92,4-88,2)/92,4= 4,5% che soddisfa la nostra soglia di 2% di guadagno minimo.

La combinazione minima è V1 in K3, V2 e V3 in K2, nessuna vista in K1.
Quindi C(s1,K) =
5040/1+1080/10+2700/10=5040+108+270=5378

Ottenendo il seguente grafico dopo i selection edge cut:

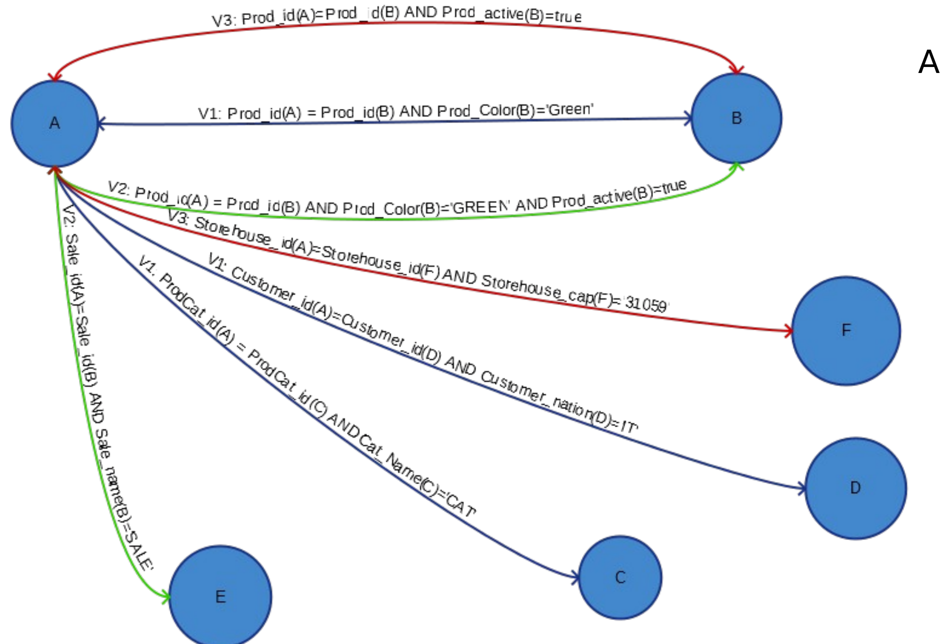


*Illustrazione 17: Algoritmo proposto 3*

questo punto possiamo utilizzare Join edge cut sull'arco di V2: da A ad E

Dato che togliendo questo arco il grafo si divide in due componenti distinte, e che in E non rimangono archi, possiamo creare V4 e V5, dove V5 è la sola selezione dei dati in E (relazione ).

Andiamo a vedere subito il grafico risultante:

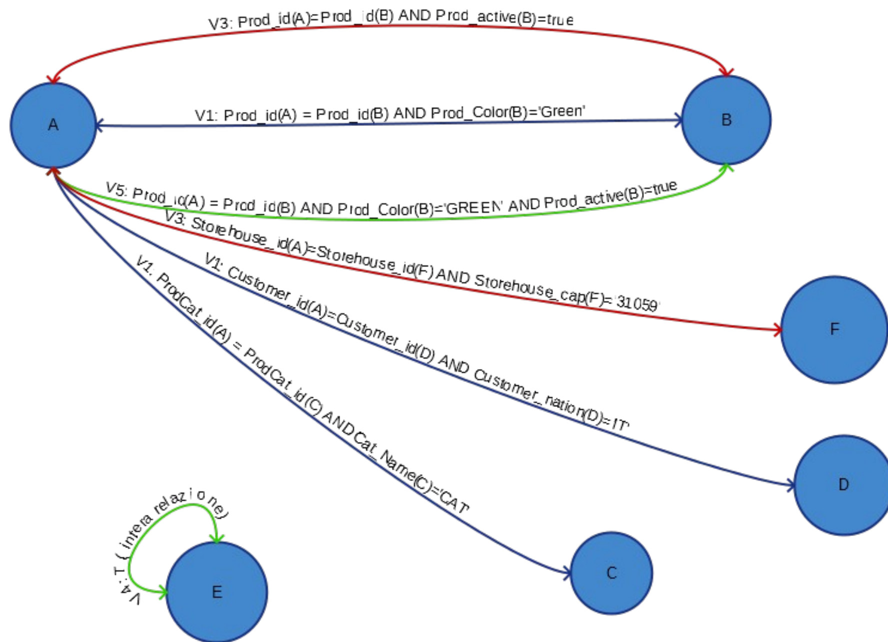


*Illustrazione 18: Algoriitmo proposto 4*

Nota:
Il costo in questo caso, potrebbe anche aumentare rispetto al minimo costo possibile , in effetti Q2 sarà più lenta ( dovrà eseguire una join tra V4 e V5 ), l'algoritmo non ottimizzato avrebbe considerato anche questa eventualità e si sarebbe accorto della possibilità di mettere V4 nella partizione più rapida, ottenendo un calcolo di un costo in ogni caso, ma la nostra euristica si basa sul fatto di ottenere comunque un costo minimo maggiore al minimo dei costi già calcolati, quindi ci obbliga a verificare il calcolo del costo minimo e confrontarlo.

Otteniamo:

Costo(s1,km) = 90*7*2*4/100 + (90*6*2/100+DJ) + 90*10*3/100 = …

Al passo successivo usiamo di nuovo join edge cut sull'arco di V1 A → C

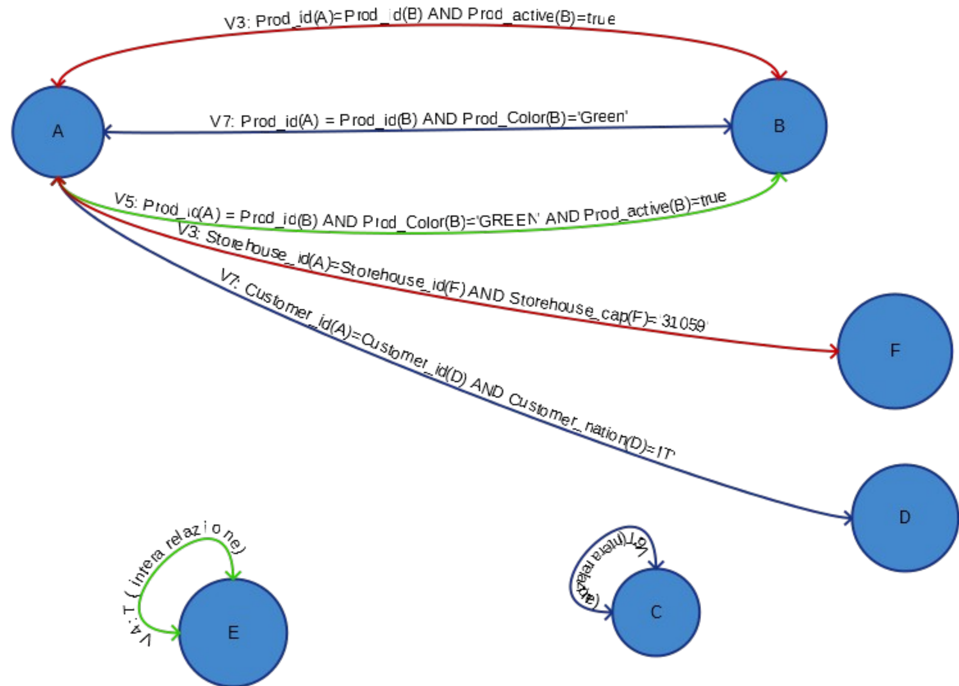


*Illustrazione 19: algoritmo proposto 5*

Ne calcoliamo il costo e poi applichiamo nuovamente il join edge cut A→D:

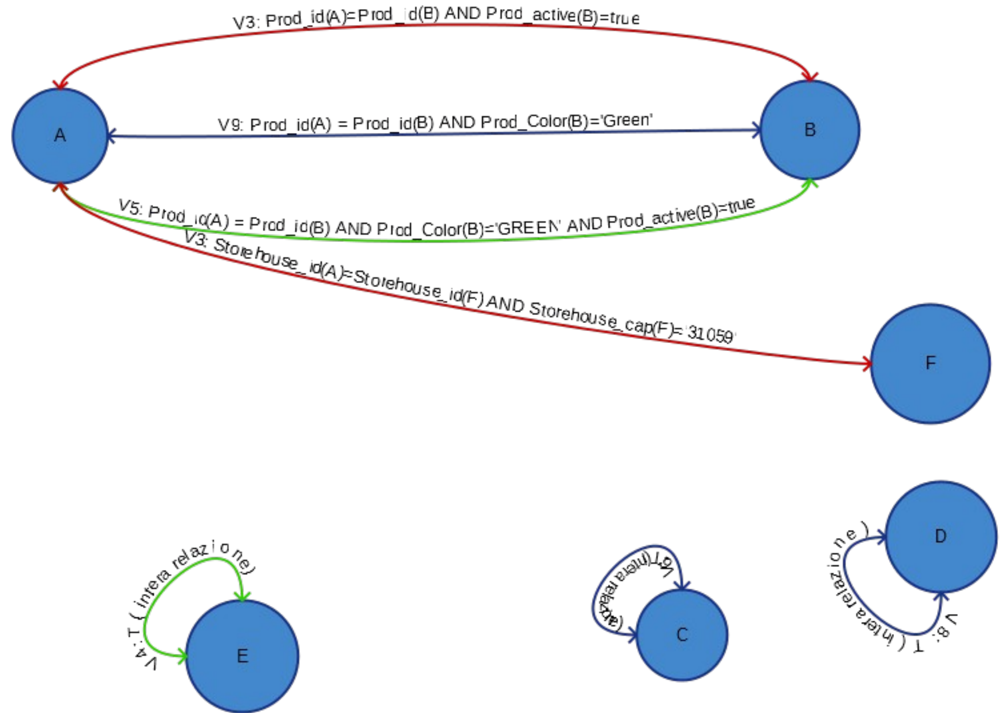


*Illustrazione 20: Algoritmo proposto 6*

A questo punto abbiamo due strade: o applichiamo nuovamente join edge cut A->F oppure view merging V9+V5

Se applichiamo view merging abbiamo questo stato:

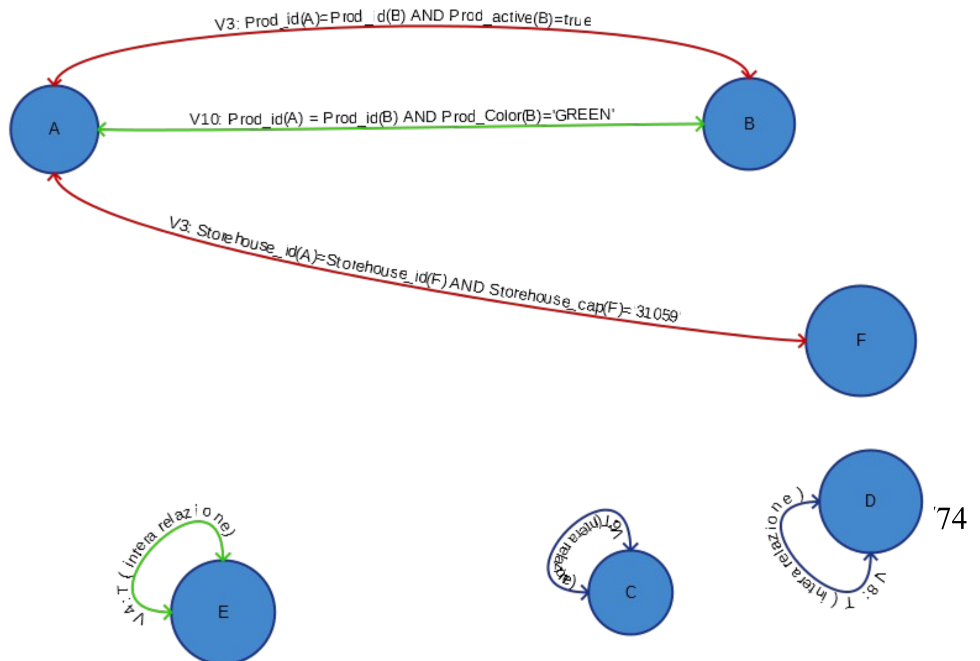


*Illustrazione 21: Algoritmo proposto 7*

Altrimenti questo:

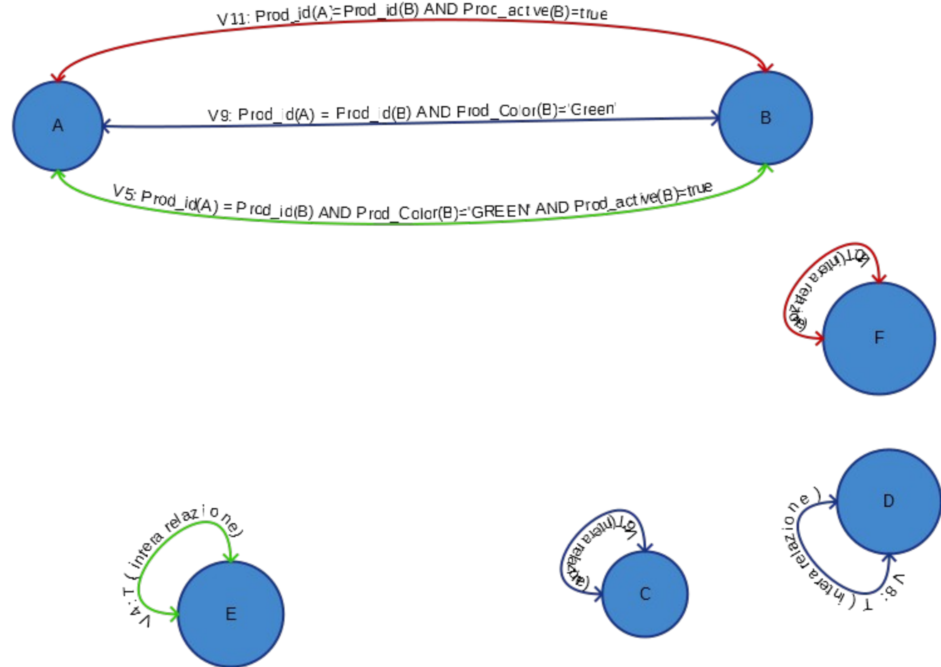


*Illustrazione 22: Algoritmo proposto 8*

A cui far seguire un view merging ottenendo lo stato finale:

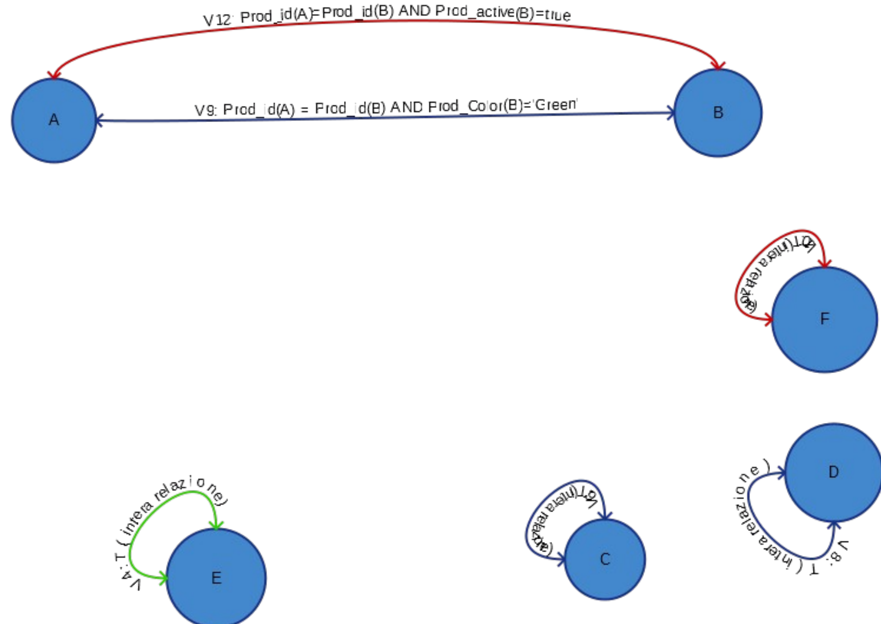


*Illustrazione 23: Algoritmo proposto 9*

## 4. Verifica dell'algoritmo proposto, metodologia di stima

Seguendo quanto visto nei capitoli precedenti, vediamo di valutare ed inquadrare il nostro algoritmo rispetto ai precedenti.

Innanzitutto, dato che l'algoritmo proposto, a differenza dei precedenti, tiene in considerazione la possibilità di un ambiente distribuito in cui ogni vista giaccia su un tablespace con una determinata latenza di accesso, per raffrontarlo agli altri considereremo per essi una latenza media ponderata rispetto alle dimensioni dei tablespace.

In secondo luogo dobbiamo stimare dimensione, tempo di costruzione e costi delle query sulle viste materializzate.

Università
Ca'Foscari
Venezia

## 4.1. Stima della dimensione di una vista materializzata

Possiamo stimare la dimensione di una vista materializzata in questo modo:
- Sia |V| il numero di record nella vista materializzata
- Siano $c_1, c_2, \ldots, c_n$ n colonne della vista materializzata
- Sia $||c_i||$ la dimensione in byte del tipo di dato della colonna i

Nel caso di datawarehouse sono poco importanti le colonne di tipo testo libero o dati binari, in caso di presenza di tali colonne bisognerà stimarne una lunghezza media.

La dimensione dunque di una vista materializzata sarà data da

$$Dim[V] = |V| * \sum_{i=1..n} ||c_i||$$

Nel caso di viste raggruppate, per stimare il numero di record distinti, si può utilizzare una campionatura per determinare il numero di risultati e riproporzionarla sul totale dei record; ad esempio, se la vista (non raggruppata) contiene 100000 record, se ne possono leggere il 5% (5000), verificare il numero di valori distinti rispetto al gruppo e poi moltiplicarlo per 20.

Segue un esempio:

Consideriamo la seguente vista:

```
V1:SELECT codice_cliente::integer,sum(valore)::double precision
FROM documenti group by codice_cliente;
```

Mettiamo che
||integer|| = 4,
||double precision|| = 8,
La lettura dei primi 5% record dei documenti ha dato un numero di codice_cliente distinti di 200.

Avrà una dimensione stimata di Dim[V1] = |V1| * (||integer|| + ||double precision|| ) = 4000 * (4+8) = 48.000

Nel caso di viste aventi condizioni di join, per calcolarne il numero di record, si potranno moltiplicare le stime fatte su ogni singola tabella / vista di join.

Ad esempio consideriamo la seguente query:

SELECT b.provincia,a.anno,sum(a.valore) FROM documenti a join clienti b on a.cod_cliente=b.cod_cliente group by b.provincia,a.anno;

In questo caso consideriamo il 5% dei record di a e vediamo quanti anni diversi otteniamo, poi il 5% dei record di b e vediamo quante provincie diverse otteniamo.
Se ad esempio abbiamo una stima di 10 provincie diverse e 4 anni, stimiamo il numero di record in 10*4 = 40 record.

Una ottimizzazione che si può apportare a tale stima, è di considerare un numero di record statistici a scalare in base alla dimensione della tabella; in questo modo abbiamo stime più accurate;

Ad esempio, se abbiamo solo 10 depositi interni di cui vogliamo vedere la regione di appartenenza, è possibile scandirli tutti;
Se si hanno 10000 clienti diversi in cui vogliamo trovare quante provincie distinte sono presenti, possiamo ad esempio scandirne il 50% ( 5000 record ).
Infine se abbiamo 50.000.000 di documenti di cui dobbiamo trovare gli anni distinti, vorremo scandirne solo una piccola parte, altrimenti il tempo necessario per il calcolo della dimensione della vista materializzata sarebbe troppo elevato.

Quindi consideriamo una tabella così composta:
Dimensione massima della tabelle - Dimensione in percentuale del campione – Dimensione massima del campione.

Ad esempio

1000 - 100% -
10000 - 75% -
100000 - 50% -
400000 - 25% -

>400000 - 10% - 1000000

Con il significato che, per le tabelle fino a 1000 record, campioniamo l'intera tabella, sopra i 10000 record il 75%, etc… , fino ad arrivare a tabelle che superano i 400.000 record, per le quali campioniamo il 10% dei record, fino ad un massimo di 1000000 di record.

Naturalmente più il campione sarà elevato, più la stima della dimensione della vista / tabella risulterà corretta.

Università
Ca'Foscari
Venezia

## 4.2. Stima del tempo di costruzione di una vista materializzata

Per stimare il tempo di costruzione di una vista materializzata dovremmo considerare tre fattori:

1) Dimensione della vista,
2) Latenza di scrittura del tablespace dove è presente la vista.
3) Tempo di esecuzione della vista,

Per quanto riguarda il punto 1 l'abbiamo già trattato nel capitolo 4.1.

Per quanto riguarda il punto 2, per gli algoritmi che non ne tengono conto consideriamo la latenza media ponderata su tutti i tablespace, per il nostro algoritmo la latenza del tablespace dove è creata la vista.

Per quanto riguarda il punto 3, abbiamo due strade:
a) Ci possiamo avvalere, se disponibile, della stima data dal planner del nostro sistema database
b) Possiamo apportare noi una stima del tempo di esecuzione, data dal numero di record scanditi.

Esploriamo il caso (b):

Possiamo calcolare il numero di record scanditi dalla query sulla vista come la totalità dei record delle tabelle coinvolte dopo l'applicazione dei criteri di filtro.

Ipotizziamo di avere la seguente query:

```
SELECT b.provincia,a.anno,sum(a.valore) FROM documenti a
join clienti b on a.cod_cliente=b.cod_cliente group by
b.provincia,a.anno;
```

Stimiamo il tempo di esecuzione in |documenti| + |clienti|, ovvero la totalità dei record presenti in tutte le tabelle coinvolte.

Se la query invece presenta condizioni di filtro:

Università Ca'Foscari Venezia

```
SELECT b.provincia,a.anno,sum(a.valore) FROM documenti a
join clienti b on a.cod_cliente=b.cod_cliente where b.regione =
'VENETO' group by b.provincia,a.anno;
```

Possiamo applicare il principio di campionatura, nel medesimo modo di quanto già visto nel campitolo 4.1, per stimare il numero di record presenti in b, una volta applicato il criterio di filtro.

Per i nostri esempi, ci avvaleremo del peggior sistema database possibile, infatti considereremo l'assenza di planner, l'esecuzione iniziale di tutte i prodotti cartesiani ed infine l'applicazione dei criteri di filtro, quindi per semplicità stimeremo il tempo di esecuzione come la moltiplicazione della dimensionalità di tutte le tabelle / viste coinvolte.

Il tempo di creazione di una vista materializzata sarà dunque dato da:
Tempo di esecuzine della vista + Dimensione della vista * Latenza di accesso in scrittura del tablespace dove la vista giace

## 4.3. Stima del tempo di esecuzione di una query su viste materializzate

Per quanto riguarda la stima del tempo di esecuzione di una query con sorgenti viste materializzate, ci avvaliamo di quanto già visto nel capitolo 4.2 per la stima del tempo di esecuzione di una vista materializzata, ovvero consideriamo di calcolare il numero di record scanditi dalla query per stimarne il tempo di esecuzione.

Utilizzeremo ancora una volta il peggior sistema possibile, in cui prima avvengono tutti i prodotti cartesiani ed infine tutte le procedure di filtro.

Università Ca'Foscari Venezia

# 5. Caso di studio e raffronto tra gli algoritmi.

In questo capitolo conclusivo, vediamo la differenza tra i vari algoritmi, applicati ad un caso di studio.
Utilizzeremo le stime approntate nel capitolo 4 per applicare gli algoritmi storici visti nel capitolo 2 ed il nostro algoritmo visto nel capitolo 3, raffrontando poi i risultati sulle stime di esecuzione di un certo numero di query.

Prendiamo come caso il seguente classico schema a stella:

Per ogni tabella, verrà indicata una stima degli elementi distinti di ogni colonna in cui tale stima è necessaria e della dimensionalità dell'intera tabella

Tabella dei fatti:
- F: Fact_Sales – |F| = 50.000.000 – Record di dimensione 1kb – Totale 50GB
   1. cddoc - Codice documento – 5.000.000 – 8 byte
   2. tpdoc - Codice tipologia di documento – 30 – 4 byte
   3. dtdoc - Data documento – 4.500 – 4 byte
   4. cdcli - Codice cliente – 5.000 – 4 byte
   5. cdage - Codice agente – 25 – 4 byte
   6. cdart - Codice articolo – 18.000 – 4 byte
   7. qta - Quantità – 8 byte
   8. val - Valore – 8 byte
   9. …. - Altri dati non considerati da questa analisi – 980 byte

- D1: Dimensione - Tabella tipologia documento - |D1| = 30 – Record da 124 byte – Totale 3kb
   1. tpdoc - Codice tipo documento – 4 byte
   2. dsdoc - Descrizione tipo documento – 120 byte

- D2: Dimensione - Tabella data documento - |D2| = 4.500 – Record da 20 byte – Totale 90kb
   1. data - Data – 4.500 – 4 byte
   2. giorno - Giorno – 31 – 4 byte
   3. mese - Mese – 12  4 byte
   4. quadr - Quadrimestre – 3 – 4 byte
   5. anno - Anno – 15 – 4 byte

- D3: Dimensione – Clienti - |D3| = 5.000 – Record da 512byte – Totale 2,5MB
   1. cdcli - Codice cliente – 5.000 – 4 byte
   2. ragso - Ragione sociale  - 240 byte
   3. cdcat - Categoria – 504 – 4 byte
   4. cdnaz – Nazione – 20 – 6 byte
   5. cdpro – Provincia – 100 – 4 byte
   6. …. - Altri dati – 250 byte

- D4 : Dimensione – Articoli - |D4| = 18.000 – Record da 768 byte – Totale 14MB
   1. cdart - Codice articolo – 4 byte
   2. descr – Descrizione – 240 btye
   3. cdca1 - Categoria 1 – 6 byte
   4. cdca2 - Categoria 2 – 6 byte
   5. cdca3 - Categoria 3 – 6 byte
   6. attiv – Articolo attivo – 1 byte
   7. … - altri dati – 505 byte

Università Ca'Foscari Venezia

## 5.1. Query frequenti

Prendiamo ora in considerazione alcune query frequenti sulla base dati (come ad esempio in [22]),

a) Calcolo del Churn rate

Ovvero sia di quanti clienti ci sono stati il mese prima e non sono più presenti il mese successivo.

Una query per il calcolo del churn rate potrebbe essere ad esempio la seguente:

Q1:
```
WITH attivita_mensile AS
(
SELECT DISTINCT
   D2.anno as anno,
   D2.mese as mese,
   F.cdcli as cdcli
FROM F
JOIN D2 ON F.dtdoc=D2.data
)
SELECT
   mese_corrente.mese as mese,
   COUNT(DISTINCT mese_precedente.cdcli) as num_clienti
FROM attivita_mensile mese_precedente
LEFT JOIN attivita_mensile mese_corrente ON
mese_corrente.cdcli=mese_precedente.cdcli AND
mese_corrente.mese=mese_precedente.mese+INTERVAL '1
month' AND CASE WHEN mese_corrente.mese=1 THEN
mese_corrente.anno=mese_precedente.anno+1 ELSE
mese_corrente.anno=mese_precedente.anno END
WHERE mese_corrente.cdcli IS NULL AND
mese_corrente.anno=2015
GROUP BY 1;
```

Che calcola il numero di clienti presenti nel mese precedente e non nel successivo, per tutti i mesi della nostra tabella dei fatti.

Università
Ca'Foscari
Venezia

Consideriamo che la nostra azienda voglia anche calcolare il churn rate per linea di prodotto (cdca1 – Categoria 1 degli articoli), avremo una seconda tipologia di query così composta:

```
Q2:
WITH attivita_mensile AS
(
SELECT DISTINCT
   D2.anno as anno,
   D2.mese as mese,
   F.cdcli as cdcli,
   D4.cdca1 as cdca1
FROM F
JOIN D2 ON F.dtdoc=D2.data
JOIN D4 ON F.cdart=D4.cdart
)
SELECT
   mese_corrente.cdca1 as cdca1,
   mese_corrente.mese as mese,
   COUNT(DISTINCT mese_precedente.cdcli) as num_clienti
FROM attivita_mensile mese_precedente
LEFT JOIN attivita_mensile mese_corrente ON
mese_corrente.cdcli=mese_precedente.cdcli AND
mese_corrente.cdca1=mese_precedente.cdca1 AND
mese_corrente.mese=mese_precedente.mese+INTERVAL '1
month' AND CASE WHEN mese_corrente.mese=1 THEN
mese_corrente.anno=mese_precedente.anno+1 ELSE
mese_corrente.anno=mese_precedente.anno END
WHERE mese_corrente.cdcli IS NULL AND
mese_corrente.anno=2015
GROUP BY 1,2;
```

Aggiungiamo anche una Q3 che calcola il numero di clienti nei mesi totali e una Q4 che lo faccia per categoria articolo.

```
Q3:
SELECT D2.anno as anno, D2.mese as mese,
   COUNT(DISTINCT F.cdcli ) as totcl
FROM F
JOIN D2 ON F.dtdoc=D2.data
GROUP BY 1
```

Q4:
```
SELECT D2.anno as anno,D2.mese as mese,
   D4.cdca1 as cdca1,
   COUNT( DISTINCT F.cdcli ) as totcl
FROM F
JOIN D2 ON F.dtdoc=D2.data
JOIN D4 ON F.cdart=D4.cdart
GROUP BY 1,2
```

b) Basket analysis

La seconda tipologia di query che andiamo ad implementare sono per determinare i set frequenti di articoli acquistati.

Vediamo innanzitutto le query generiche da lanciare, poi simuliamo il lancio con alcune varianti ( filtri per codici articolo o categorie )

La prima che vediamo è la query che calcola gli articoli più frequentemente acquistati ed ancora attivi

Q5:
```
SELECT F.cdart,COUNT(DISTINCT F.cddoc) FROM F JOIN D4 ON
F.cdart=D4.cdart WHERE D4.attiv
```

Passando alle coppie di articoli più frequentemente acquistati assieme:

Q6:
```
SELECT F1.cdart,F2.cdart,COUNT(DISTINCT F1.cddoc) FROM F
AS F1 JOIN F AS F2 ON F1.cddoc=F2.cddoc AND F1.cdart!
=F2.cdart JOIN D4 AS D4_F1 ON F1.cdart=D4_F1.cdart JOIN D4
AS D4_F2 ON F2.cdart=D4_F2.cdart WHERE D4_F1.attiv AND
D4_F2.attiv GROUP BY 1,2 ORDER BY 1,2,3 DESC
```

Aggiungiamo anche le varianti per categoria di articolo, nel caso in cui non siamo interessanti ad avere le corrispondenze tra tutti i possibili articoli, ma solo, ad esempio, per quelli di categoria 2 XYZ ( solo per le coppie )

Università
Ca'Foscari
Venezia

Q7:

```
SELECT F1.cdart,F2.cdart,COUNT(DISTINCT F1.cddoc) FROM F
AS F1 JOIN F AS F2 ON F1.cddoc=F2.cddoc AND F1.cdart!
=F2.cdart JOIN D4 AS D4_F1 ON F1.cdart=D4_F1.cdart JOIN D4
AS D4_F2 ON F2.cdart=D4_F2.cdart WHERE D4_F1.attiv AND
D4_F2.attiv        WHERE D4_F1.cdca2 ='XYZ' GROUP BY 1,2
ORDER BY 1,2,3 DESC
```

c) Query statistiche

Aggiungiamo infine analisi basilari basate su query statistiche

Q8.1:

Statistica sul venduto ( quantità e valore ) rispetto alle categorie (3) di articoli nell'anno attuale

```
SELECT D4.cdca3,sum(F.val),sum(F.qta) FROM F JOIN D4 ON
F.cdart=D4.cdart JOIN D2 ON F.dtdoc=D2.data WHERE
D2.anno=2015 GROUP BY 1
```

Q8.2:

Statistica sul venduto ( quantità e valore ) rispetto alle categorie (3) di articoli nell'anno attuale, per mese

```
SELECT D4.cdca3,D2.mese,sum(F.val),sum(F.qta) FROM F JOIN
D4 ON F.cdart=D4.cdart JOIN D2 ON F.dtdoc=D2.data WHERE
D2.anno=2015
GROUP BY 1,2;
```

Q9:

Statistica sul venduto (valore) per provincia dei clienti italiani nell'anno attuale divisa per quadrimestre:

```
SELECT D3.cdpro,D2.quadr,sum(F.val) FROM F JOIN D3 ON
F.cdcli=D3.cdcli JOIN D2 ON F.dtdoc=D2.data WHERE
D2.anno=2015 AND D3.cdnaz='IT'
GROUP BY 1,2;
```

Q10:

Statistica sul venduto (valore) per provincia dei clienti italiani nell'anno precedente divisa per quadrimestre:

```
SELECT D3.cdpro,D2.quadr,sum(F.val) FROM F JOIN D3 ON
F.cdcli=D3.cdcli JOIN D2 ON F.dtdoc=D2.data WHERE
D2.anno=2014 AND D2.cdnaz='IT'
GROUP BY 1,2;
```

Q11:

Statistica mensile sul venduto (valore) per agente nell'anno corrente

```
SELECT F.cdage,D2.mese,sum(F.val) FROM F JOIN D2 ON
F.dtdoc=D2.data WHERE D2.anno=2015
GROUP BY 1,2;
```

Naturalmente queste sono solo alcune delle possibili analisi sul DW, ma ci fermeremo qua per il nostro studio.

Università Ca'Foscari Venezia

## 5.2. Dimensionalità e velocità del sistema

Per poter raffrontare l'algoritmo proposto con gli algoritmi storici, dobbiamo dimensionare il sistema, terremo in considerazione che le tabelle e viste degli algoritmi, che non tengono conto della posizione delle viste e tabelle, vengano ordinate e impostate sul primo schema disponibile fino ad esaurimento dello spazio, sul secondo fino ad esaurimento dello spazio e così via.

Identifichiamo le seguente partizioni dati con le corrispettive velocità ( in rapporto alla più lenta)

Tipologia – Velocità - Dimensione
Memory ( tabelle persistenti in memoria) – 1000 - 4GB (“it's askin to comparing a 1,200-mph F/A-18 fighter jet to a garden slug. “)
HDD 1 (SSD) - 50 – 200GB
HDD 2 (SSD) - 50 - 200GB
HDD 3 ( Remoto ) – 1 – 100TB

## 5.3 Tempo di esecuzione base delle query

In comune con tutti gli algoritmi visti stimiamo il tempo di esecuzione delle query di esempio, utilizziamo impropriamente la moltiplicazione come stima del tempo di join, naturalmente i sistemi database non eseguono un semplice prodotto cartesiano per ricreare i collegamenti delle join, ma ci servirà come stima molto grossolana in modo da creare il raffronto tra le velocità delle query, inoltre i sistemi normalmente ottimizzano le query in modo da renderle il più efficienti possibile, noi nella nostra stima considereremo un sistema non dotato di nessuna ottimizzazione, quindi che esegua tutti i prodotti cartesiani e alla fine faccia le condizioni di filtro.

Ricordando che
$|F|=5*10^7$
$|D2|=4.500$
$|D3|=5.000$
$|D4|=18.000$

Q1: $(|F|*|D2|) * (|F|*|D2|) = (50*10^6*4.500) * (50*10^6*4.500) = 5*10^{21}$
Q2: $(|F|*|D2|*|D4|) * (|F|*|D2|*|D4|) = (50*10^6*4.500*18.000) * (50*10^6*4.500*18000) = 1{,}6*10^{31}$
Q3: $|F|*|D2| = 2{,}25*10^{11}$
Q4: $|F|*|D2|*|D4| = 4*10^{15}$
Q5: $|F|*|D4| = 9*10^{11}$
Q6: $|F|*|D4|*|F|*|D4| = 8*10^{23}$
Q7: $|F|*|D4|*|F|*|D4| = 8*10^{23}$
Q8.1: $|F|*|D4|*|D2| = 4*10^{15}$
Q8.2: $|F|*|D4|*|D2| = 4*10^{15}$
Q9: $|F|*|D3|*|D2| =1*10^{15}$
Q10: $|F|*|D3|*|D2| =1*10^{15}$
Q11: $|F|*|D2| = 2{,}25*10^{11}$

Università
Ca'Foscari
Venezia

## 5.4 Esempio di esecuzione dell'algoritmo 2.2

Utilizziamo ora l'algoritmo 2.2 per determinare il set di viste con il maggior beneficio e stimiamo infine il tempo totale di esecuzione delle nostre query.

Costruiamo per prima cosa il lattice delle nostre possibili view e ne stimiamo la dimensione e il tempo di esecuzione.

Per semplicità divideremo tutti i costi di esecuzione per la dimensione della tabella dei fatti ( che comunque è comune a tutte le possibili viste materializzate )

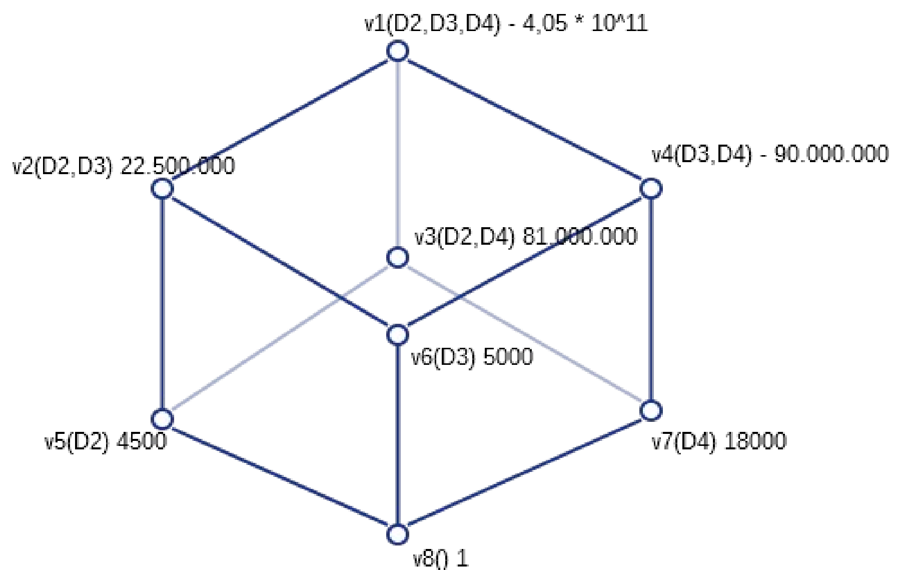


*Illustrazione 24: Esempio completo lattice*

Costruiamo il lattice delle possibili viste che raggruppano tabella dei fatti e dimensioni:

Andiamo a iterare una prima volta per ottenere i benefici:

Consideriamo la situazione iniziale M= {v1}, verifichiamo per v2:

w = v2,v5,v6,v8
vm1 = v1 = 4,05*10^11
Bw = max { 4,05*10^11 – 2,22*10^7 , 0 } = 4,049778*10^11 ( per ogni w )
B(v2,M)= 4,049778*10^11 x 4 = 1,6199112*10^12

Ovvero, avendo il solo nodo v1 in M, il beneficio di aggiungere v2 equivale al suo stesso beneficio moltiplicato per il numero di nodi sottostanti.

Fino ad ottenere il seguente schema
B(v2,M) = 1,6199112*10^12
B(v3,M) = 1,619676*10^12
B(v4,M) = 1,61964*10^12

Per v5,v6,v7,v8 le differenze sono troppo basse e quindi verranno ignorati.

Consideriamo quindi la materializzazione di v2 dato che porta il beneficio più alto.

Andiamo ad iterare:

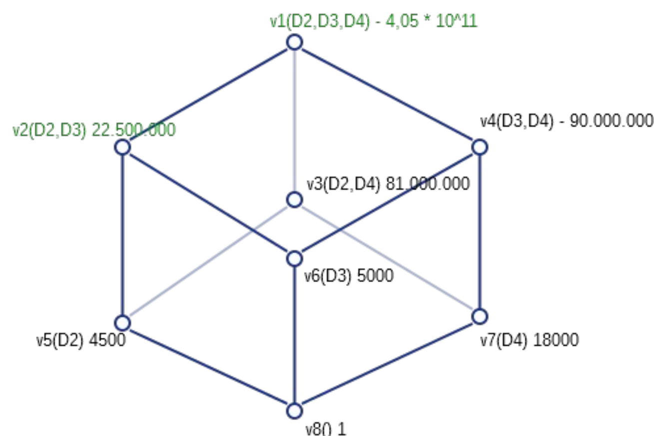


*Illustrazione 25: Esempio lattice v1*

Calcoliamo il beneficio per v3
w = v3,v5,v7,v8
per v3
vm1 = C(v1) = 4,05*10^11
Bw= max { 4,05*10^11 – 8,1*10^7 ,0 } = 4,04919*10^11
per v5
vm1 = C(v1) = 4,05*10^11
vm2 = C(v2) = 2,25*10^7
Bw = max { 2,25*10^7 – 4.500 , 0 } = 2,24955e+7
Dato che il minimo costo dei predecessori di v5 è quello di v2 di 2,25*10^7
per v7 e v8 il beneficio è quasi ininfluente.

Per un totale di
B(v3,M) = 4,04919*10^11

Concludendo i calcoli abbiamo:
B(v3,M) = 4,04919*10^11
B(v4,M) = 1,61964*10^12
B(v5,M) = 4,49955*10^7
B(v6,M) = 4,4995*10^7
B(v7,M) = 4,05022482*10^11
B(v8,M) = 2,25*10^7

Andiamo quindi a materializzare v4.

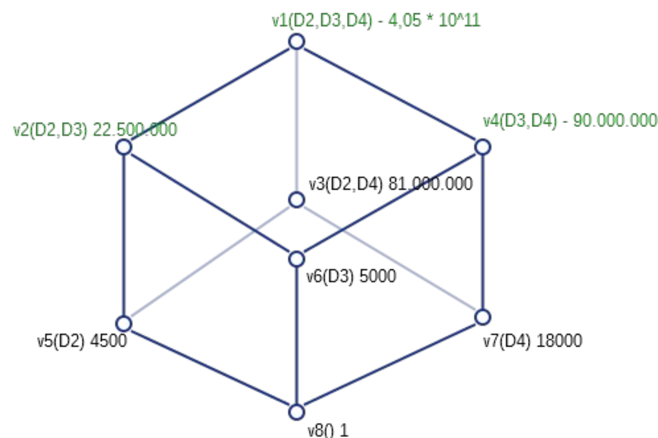


*Illustrazione 26: Esempio lattice 2*

Facciamo una quarta iterazione andando a considerare i benefici delle restanti viste:

B(v3,M) = 4,04919*10^11 + 2,24955e+7 + 2,25*10^7 = 4,0496399*10^11
B(v5,M) = 4,49955*10^7
B(v6,M) = 4,4995*10^7
B(v7,M) = 1,12482*10^8
B(v8,M) = 2,25*10^7

Andando a materializzare la vista v3.

Finiamo con una quarta ed ultima iterazione dell'algoritmo.

B(v5,M) = 4,49955*10^7
B(v6,M) = 4,4995*10^7
B(v7,M) = 3,05982*10^8
B(v8,M) = 2,25*10^7

Andando dunque a materializzare v7.

Completando dunque con il seguente schema:

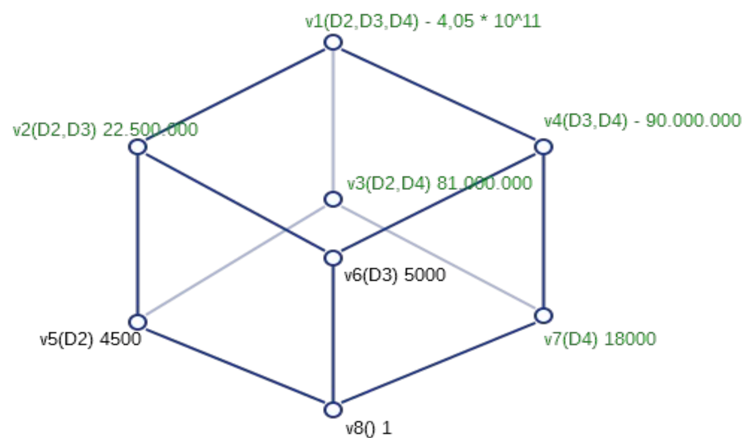


*Illustrazione 27: Esempio lattice 3*

Con significato:

Andiamo a materializzare le seguente viste:

FxD4, FxD4xD2, FxD2xD3,FxD3xD4, FxD2xD3xD4

Il tempo di esecuzione totale delle query risulta dunque:

Q1: $(|F|*|D2|) * (|F|*|D2|) = (50*10^6*4.500) * (50*10^6*4.500) = 5*10^{21}$
Q2: $(|F|) * (|F|) = (50*10^6) * (50*10^6) = 2,25*10^{15}$
Q3: $|F|*|D2| = 2,25*10^{11}$
Q4: $|F| = 5*10^6$
Q5: $|F| = 5*10^6$
Q6: $|F|*|F| = 2,25*10^{15}$
Q7: $|F|*|F| = 2,25*10^{15}$
Q8.1: $|F| = 5*10^6$
Q8.2: $|F| = 5*10^6$
Q9: $|F| =5*10^6$
Q10: $|F| =5*10^6$
Q11: $|F|*|D2| = 2,25*10^{11}$

Come vediamo molte query si sono ridotte alla sola scansione di |F| record, evitando le join in quanto già materializzate.

## 5.5 Esecuzione dell'algoritmo 2.3 e la versione proposta 3

Come ultimo capitolo vediamo come sarebbe l'esecuzione dell'algoritmo base (2.3) e dell'algoritmo proposto 3 e li andiamo a confrontare assieme e con il risultato che si avrebbe materializzando le viste del capitolo precedente.

Per far partire l'algoritmo dobbiamo innanzitutto rappresentare le query Q in un multi-query graph:

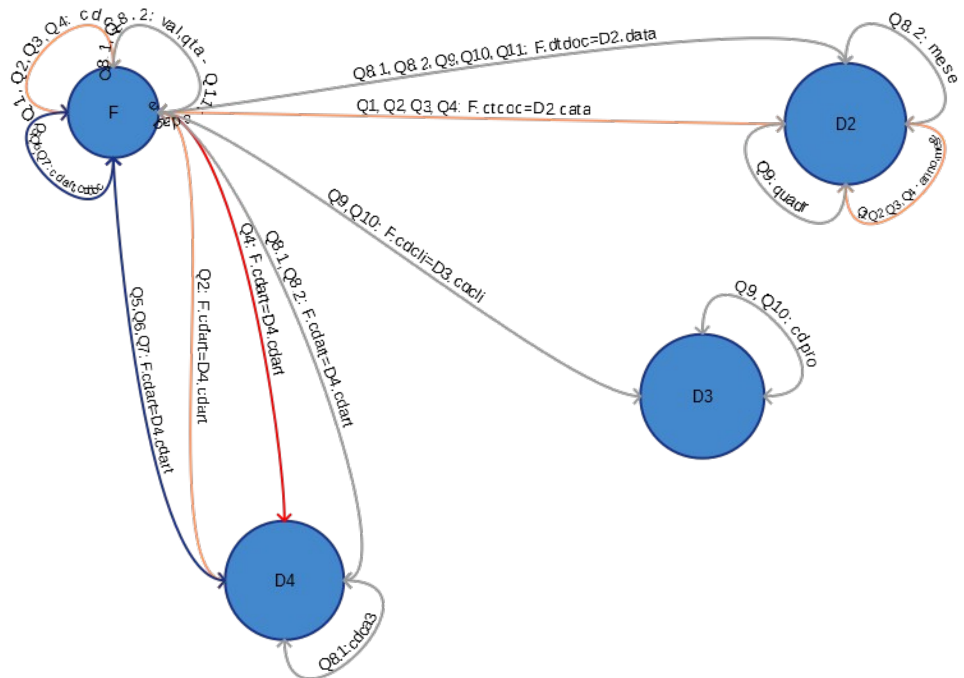


*Illustrazione 28: Esempio completo algoritmo proposto 1*

Dopo di questo dobbiamo cominciare ad applicare una delle tre trasformazioni possibili ( selection edge cut, join edge cut e view merging )

Tralasciando tutti i passaggi algoritmici, dato che esporli uno ad

uno sarebbe troppo gravoso, ci soffermiamo su alcune particolarità:

1) Selection edge cut è fondamentale per l'algoritmo proposto, in quanto allargare la dimensione di una vista in numero di colonne ne incrementa incredibilmente lo spazio occupato sul tablespace, mentre per l'algoritmo base è quasi ininfluente in quanto sia tempo di costruzione che velocità di esecuzione ne risentono poco.

2) Mentre per l'algoritmo visto nel capitolo precedente l'ottimizzazione massima che si può ottenere è leggere |F| record ( tutte le join eseguite ed una lettura su una vista grande come la tabella dei fatti ) per quanto riguarda l'algoritmo proposto la massima ottimizzazione è |V| con V pari al numero di risultati della query, infatti non ottimizzo le join, ma le singole query sul database.

3) L'ordine con cui si fanno selection edge cut, view merging e join edge cut è fondamentale per l'algoritmo proposto, infatti mentre l'algoritmo base ne risente in quanto la velocità di esecuzione varia a seconda di cosa è materializzato, l'algoritmo proposto ne fa anche dipendere cosa è possibile o meno materializzare nello spazio proposto e come configurare la posizione delle viste nei vari tablespace.
Risulta più efficiente iniziare a fare view merging se possibile, join edge cut come seconda possibilità ed infine selection edge cut per ridurre il numero di viste create.

Dobbiamo inoltre fissare alcune costanti per permettere l'esecuzione dei nostri algoritmi.
Primo: c ovvero la costante che determina il peso di costruzione delle viste rispetto all'esecuzione delle query.
Possiamo indicarla nel nostro esempio ad 1, ovvero ipotizziamo che il tempo di costruzione sia influente quanto quello di esecuzione.
Secondo ipotizziamo una nostra costante di % minima di guadagno pari al 20%, ovvero non vogliamo soluzioni che riducano meno del 20% le prestazioni del nostro sistema.

In conclusione supponendo che il risultato finale sia il seguente:

Università Ca'Foscari Venezia

V1: SELECT F.cdcli,F.dtdoc,D4.cdca1 FROM F JOIN D4
Di dimensione 12byte x 5 x 10^7 = 600MB
Di tempo di costruzione pari a 5 x 10^7 x 18000 = 9 x 10^11
Posizionata in memoria ( residuo 3,4GB )

V2:
SELECT F.cdart,F.cddoc FROM F JOIN D4 WHERE D4.attiv
Di dimensione 8byte x 5 x 10^7 = 400MB
Di tempo di costruzione pari a 5 x 10^7 x 18000 = 9 x 10^11
Posizionata in memoria ( residuo 3,0GB )

V3:
SELECT D4.cdca3,F.val,F.qta FROM F JOIN D4
Di dimensione 20byte x 5 x 10^7 = 1GB
Di tempo di costruzione pari a 5 x 10^7 x 18000 = 9 x 10^11
Posizionata in memoria ( residuo 2,0GB )

V4:
SELECT D4.cdca3,F.dtdoc,F.val,F.qta FROM F JOIN D4
Di dimensione 16byte x 5 x 10^7 = 800MB
Di tempo di costruzione pari a 5 x 10^7 x 18000 = 9 x 10^11
Posizionata in memoria ( residuo 1,2GB )

V5:
SELECT D3.cdpro,F.dtdoc,F.val FROM F JOIN D3
Di dimensione 16byte x 5 x 10^7 = 800MB
Di tempo di costruzione pari a 5 x 10^7 x 5000 = 2,5 x 10^10
Posizionata in memoria ( residuo 400MB )

V6:
SELECT F.cdage,D2.mese,F.val FROM F JOIN D2
Di dimensione 16byte x 5 x 10^7 = 800MB
Di tempo di costruzione pari a 5 x 10^7 x 4500 = 2,25 x 10^11
Posizionata sul disco 1

La velocità di esecuzione delle query risulta quindi la seguente:

Q1: (|F|*|D2|) * (|F|*|D2|) = (50*10^6*4.500) * (50*10^6*4.500) = 5*10^21
Q2: (|F|) * (|F|) = (50*10^6) * (50*10^6) = 2,25*10^15

Q3: |F| = 2,25*10^11
Q4: |F| = 5*10^6
Q5: |F| = 5*10^6
Q6: |F|*|F| = 2,25*10^15
Q7: |F|*|F| = 2,25*10^15
Q8.1: |F| = 5*10^6
Q8.2: |F| = 5*10^6
Q9: |F| =5*10^6
Q10:|F| =5*10^6
Q11:|F|*|D2| = 2,25*10^11 *50 (Su disco più lento)

## 5.6 Schema di confronto

| Query | Esec. Su 5.4 ( Algoritmo non ottimizzato su disco 50 volte più lento ) | Esec. su 5.5 | Diff. |
|---|---|---|---|
| Q1 | 5*10^21*50 | 5*10^21 | **-98%** |
| Q2 | 2,25*10^15*50 | 2,25*10^15 | **-98%** |
| Q3 | 2,25*10^15*50 | 2,25*10^11 | **-99.99%** |
| Q4 | 5*10^6*50 | 5*10^6 | **-98%** |
| Q5 | 5*10^6*50 | 5*10^6 | **-98%** |
| Q6 | 2,25*10^15*50 | 2,25*10^15 | **-98%** |
| Q7 | 2,25*10^15*50 | 2,25*10^15 | **-98%** |
| Q8 | 5*10^6*50 | 5*10^6 | **-98%** |
| Q9 | 5*10^6*50 | 5*10^6 | **-98%** |
| Q10 | 5*10^6*50 | 5*10^6 | **-98%** |
| Q11 | 2,25*10^15*50 | 2,25*10^11 *50 | --- |

Naturalmente lo schema superiore tiene conto che tutte le viste materializzate dell'algoritmo 5.4 siano presenti sul disco 50 volte più lento della memoria, da questo la variazione del 1/50 del tempo medio di esecuzione; tranne che per Q11 che anche nel nostro algoritmo ottimizzato dovrà essere presente sul disco

lento per mancanza di spazio e per Q3 che ha una notevole variazione di tempo di esecuzione.

# 6. Considerazioni finali e sviluppi futuri

Con l'algoritmo proposto si è cercato di aggiungere alla già complessa scelta delle viste da materializzare anche la necessità sempre più pressante di gestire dati presenti su più dispositivi fisici e logici di natura molto diversa.

Questa ottimizzazione è stata resa possibile partendo dalla base dell'algoritmo già presentato in [10] per la scelta delle viste da materializzare, aggiungendo un vincolo spaziale dato dalla posizione fisica delle viste create ed indicando un modo per escludere le ottimizzazioni non necessarie attraverso una euristica.

Rimandiamo a sviluppi futuri alcune considerazioni:
come prima cosa la verifica dell'algoritmo proposto, confrontandolo con altri algoritmi storici come possiamo trovare ad esempio in [ 11 ] e [ 12 ].

La versione completa dell'algoritmo non richiede molte verifiche, dato che segue tutto l'albero delle possibili combinazioni, semplicemente escludendo le non ammissibili, calcolando il costo di tutte le altre ed alla fine ricavandone quella con costo minimo.

Per quanto riguarda invece la versione euristica, rimane da determinare che come effetto indesiderato del taglio di soluzioni sotto la soglia richiesta di ottimizzazione non si vada a tagliare nella maggior parte delle volte anche la soluzione ottimale ricercata.

Un ulteriore sviluppo futuro riguarda la possibilità di ottimizzare anche gli indici sulle viste materializzate, indicando anche per essi la posizione fisica ( qualora il sistema DW ne consenta una locazione diversa da quella della vista materializzata associata) si veda come esempio [ 13 ].

Infine si può anche aggiungere una ottimizzazione per tenere “vicine” le viste materializzate che riguardano la stessa query, infatti una join tra due tabelle è molto più efficiente se esse

Università
Ca'Foscari
Venezia

sono presenti nella stessa unità di archiviazione, molto spesso infatti il sistema DW caricherà contestualmente entrambe le viste in memoria leggendole dalla stessa unità logica in una volta sola.

Università
Ca'Foscari
Venezia

# 7. Bibliografia

[1] J. Widom, editor. Data Engineering, Special Issue on Materialized Views and Datawarehousing, volume 18(2). IEEE, 1995.

[2] J. Widom. Research problems in data warehousing. In Proc. CIKM, pages 25-30, Nov. 1995.

[3] S. Dar, H. V. Jagadish, A. Y. Levy, and D. Srivastava. Answering SQL Queries with Aggregation using
Views. In Proc. of VLDB, 1996.

[4] A. Gupta, V. Harinarayan, and D. Quass. Aggregate query processing in data warehousing environments.
In Proc. of VLDB, 1995.

[5] V. Harinarayan, A. Rajaraman, and 3. D. Ullman. Implementing Data Cubes Efficiently. In Proc. ACM
SIGMOD, 1996.

[6] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In Proc. ACM SIGMOD, 1996.

[7] A. Levy, A. 0. Mendelson, Y. Sagiv, and D. Srivastava.Answering Queries using Views. In Proc. ACM PODS, 1995.

[8] H. Gupta. Selection of Views to Materialize in a DataWarehouse. In Proc. ICDT, 1997.

[9] Antonio Albano, *"A Greedy Algorithm for the Selection of Materialized Views" - Decision Support Databases Essential 1, 2015*.

[10] Dimitri Theodoratos and Timos Sellis Data Warehouse Configuration, 1997

[11] Imene Mami and Zohra Bellahsene, A Survey of View Selection Methods

[12] Mr. P. P. Karde, Dr. V. M. Thakare, Selection & Maintenance of Materialized View and It's Application for Fast Query Processing: A Survey

[13] Kamel Aouiche and Jerome Darmont, Data Mining-based Materialized View and Index Selection in Data Warehouses

[14] Agrawal, S., Chaudhuri, S., and Narasayya, V. (2000). Automated selection of materialized views and indexes for SQL databases. In Proceedings of the International

Conference on Very Large Data Bases (VLDB), pages 496–505, Cairo, Egypt.

[15] Michael Teschke, Achim Ulbrich, Using Materialized Views To Speed Up Data Warehousing.

[16] Goretti K.Y. Chan, Qing Li, Ling Feng, Design and Selection of Materialized Views in a Data Warehousing Environment: A Case Study,

[17] Sanjay Agrawal,Surajit Chaudhuri,Vivek Narasayya, Automated Selection of Materialized Views and Indexes for SQL Databases,

[18] Hoshi Mistr y, Prasan Roy,S. Sudarshan, Krithi Ramamritham,Materialized View Selection and Maintenance Using MultiQuery Optimization

[19] Weifa Liang, Hui Wang, Maria E.Orlowska, Materialized views selection under the maintenance time constraint

[20] Hema S. Botre, Prof. M. S. Chaudhari, Materialized View Selection Algorithm: A Survey

[21] Jonathan Goldstein and Per-Åke Larson, Optimizing Queries Using Materialized Views: A Practical, Scalable Solution

[22] Foto Afrati1, Rada Chirkova, Shalu Gupta, and Charles Loftis., Designing and Using Views To Improve Performance of Aggregate Queries

[23] Chuan Zhang, Xin Yao, An Evolutionary Approach to Materialized Views Selection in a Data Warehouse Environment

[24] Mr. P. P. Karde1 and Dr. V. M. Thakare, selection of materialized view using query optimization in database management: an efficient metodology

[25] José Maria Monteiro, Sérgio Lifschitz, Ângelo Brayner, Automated Selection of Materialized Views

[26] B.Ashadevi, Dr.R.Balasubramanian, Optimized Cost Effective Approach for Selection of Materialized Views in Data Warehousing

[27] Jiratta Phuboon-ob, and Raweewan Auepanwiriyakul, Selecting Materialized Views Using Two-Phase Optimization with Multiple View Processing Plan