Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Magistrale in Scienze e Tecnologie Informatiche

Shrink: un nuovo operatore OLAP per la presentazione di risultati approssimati

Tesi di Laurea in Data Mining

Relatore:
Chiar.mo Prof.
Matteo Golfarelli

Presentata da:
Simone Graziani

Sessione I
Anno Accademico 2012/2013

Indice

Introduzione **1**

1 Rappresentazione approssimata di analisi OLAP **5**

1.1 Modello multidimensionale 5

1.2 Analisi OLAP 8

1.3 Rappresentazione approssimata tramite shrink 12

1.4 Approcci in letteratura 16

2 Operatore shrink **23**

2.1 Background 23

2.1.1 Schema multidimensionale 24

2.1.2 Shrinking framework 25

2.2 Misurazione dell'errore di approssimazione 28

2.3 Operatore fine shrink 30

2.3.1 Algoritmo branch-and-bound per fine shrink 32

2.3.2 Algoritmo greedy per fine shrink 36

2.4 Operatore coarse shrink 39

2.4.1 Algoritmo branch-and-bound per coarse shrink 40
2.4.2 Algoritmo greedy per coarse shrink 42

3 Implementazione degli algoritmi di shrinking 47

3.1 Ambiente di sviluppo e architettura 47
3.2 Input e output 49
3.2.1 Mondrian schema file 51
3.2.2 Output 53
3.3 Strutture dati 54
3.3.1 Ipercubo e slice 54
3.3.2 Matrici delle distanze nelle versioni greedy 57
3.3.3 Ipercubo con slice condivisi 58
3.3.4 Albero delle aggregazioni e partizioni coarse 60

4 Valutazione delle performance degli operatori di shrinking 63

4.1 Setup dei test 63
4.2 Errore di approssimazione 66
4.2.1 Fine e coarse a confronto 68
4.2.2 Errore come coefficiente di variazione 69
4.2.3 Ottimo VS Greedy 70
4.3 Performance 73
4.3.1 Tempi di esecuzione 73
4.3.2 Utilizzo delle risorse computazionali 75
4.4 Valutazioni conclusive 77

5 Conclusioni e sviluppi futuri 79

Bibliografia 83

Introduzione

In ambito *business intelligence OLAP—On-Line Analytical Processing*—è il paradigma principale per interrogare basi di dati multidimensionali e analizzare grandi quantità di dati. In una tipica sessione OLAP l'utente richiede un insieme di valori di misure corrispondenti ad una certa prospettiva di analisi e tramite una serie di *operazioni* trasforma la query iniziale fino ad arrivare ad un risultato per lui più interessante. Spesso però si presenta il problema di trovare un giusto compromesso tra quantità di dati visualizzati e precisione degli stessi: più dati danno più informazioni ma possono nascondere trend generali e richiedono uno sforzo di analisi maggiore all'utente, d'altra parte meno dati rendono difficile individuare trend su piccola scala.

Il problema descritto è particolarmente evidente quando vengono utilizzate rappresentazioni testuali come le *pivot table* per visualizzare i dati, non sempre infatti è conveniente esprimere i risultati tramite grafici o tecniche di presentazione più complesse. Inoltre negli ultimi anni hanno preso piede sempre più i cosiddetti *smart device*, come smartphone e tablet, che sono caratterizzati da una maggiore maneggiabilità rispetto a PC desktop e portatili ma allo stesso tempo presentano limitazioni dal punto di vista della presentazione a video e della connettività.

In questa tesi affrontiamo il problema di ridurre la dimensione dei risultati derivati da una generica query OLAP cercando di mantenere la perdita di precisione a livelli accettabili. La soluzione descritta può essere vista come una forma di *OLAM—On-Line Analytical Mining*—basata su clustering ge-

rarchico agglomerativo. Proponiamo quindi un nuovo operatore chiamato *shrink* da applicare ad una dimensione di un generico ipercubo OLAP per ridurne la dimensione. Il cubo è visto come un insieme di slice corrispondenti ai valori della dimensione a cui l'operatore è applicato. Le slice simili sono "fuse" assieme, rispettando la semantica dettata dalla gerarchia, sostituendole con una nuova slice rappresentativa calcolata come la media dei dati di partenza. Il processo di riduzione è vincolato da un parametro che definisce l'errore di approssimazione massimo accettabile oppure la dimensione massima del risultato.

Sono presentate due differenti versioni di shrink: *fine shrink* e *coarse shrink*. I due operatori svolgono sostanzialmente la stessa funzione ma mentre il primo riesce ad ottenere risultati caratterizzati da un minor errore, il secondo ha il vantaggio di presentare una riduzione più chiara e leggibile.

Il seguito della tesi è così organizzato:

- Il primo capitolo ha l'obiettivo di fornire le informazioni di background necessarie a comprendere i capitoli successivi. In particolare viene introdotta la problematica della rappresentazione approssimata in ambito OLAP e delle relative soluzioni presenti in letteratura.

- Nel secondo capitolo viene data una formalizzazione del framework di shrinking e vengono descritti in dettaglio gli operatori fine shrink e coarse shrink proponendo anche gli algoritmi ottimali e euristici per la loro implementazione.

- Il terzo capitolo è dedicato all'implementazione degli algoritmi descritti nel capitolo precedente. Oltre all'architettura, che dà una panoramica dei componenti principali, sono descritte le strutture dati più interessanti e importanti.

- Il quarto capitolo presenta i test effettuati per valutare la qualità dei risultati ottenuti e le performance degli algoritmi implementati.

- Infine nel capitolo conclusivo si riassume quanto già detto precedentemente e si delineano i possibili sviluppi futuri di questo studio.

Capitolo 1

Rappresentazione approssimata di analisi OLAP

In questo capitolo viene introdotta la problematica della rappresentazione approssimata durante analisi OLAP e dei relativi studi presenti in letteratura. Inizialmente introdurremo i concetti basilari del modello multidimensionale e degli operatori OLAP (sezioni 1.1 e 1.2) utilizzati per condurre analisi dinamiche di grandi quantità di dati. In seguito procederemo ad illustrare la problematica affrontata in questa tesi, l'intuizione alla base della soluzione proposta e i relativi studi già presenti in letteratura (sezioni 1.3 e 1.4).

1.1 Modello multidimensionale

Il modello multidimensionale [GR06] viene adottato come paradigma di rappresentazione dei dati nei data warehouse (DW) grazie alla sua semplicità e alla sua intuitività anche ai non esperti di informatica. Questo modello si basa sulla consapevolezza che gli oggetti che influenzano il processo decisionale sono *fatti* del mondo aziendale—come vendite, spedizioni etc.—e le occorrenze di tali fatti corrispondono ad *eventi* accaduti. Ogni evento è inol-

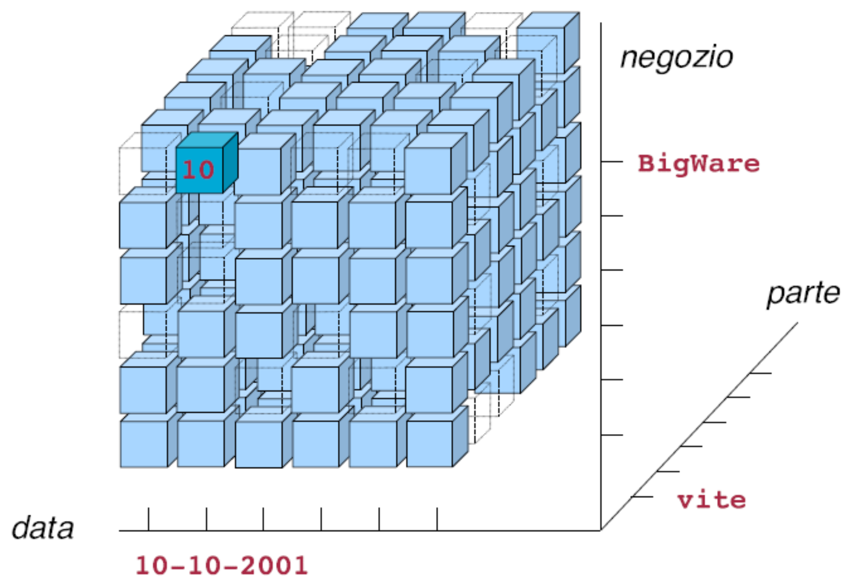


Figura 1.1: Rappresentazione di un cubo OLAP.

tre descritto quantitativamente da un insieme di *misure*: numero di vendite, costo di un acquisto.

Per poter meglio gestire la grande mole di eventi che accadono in azienda, questi vengono collocati in uno spazio n-dimensionale in cui le dimensioni definiscono le prospettive per la loro identificazione. Ad esempio gli eventi spedizione possono essere collocati in uno spazio tridimensionale in cui le dimensioni sono: data di spedizione, luogo di destinazione e modalità.

Per rappresentare questa tipologia di dati è molto diffusa la metafora del *cubo* (o *ipercubo*) in cui gli spigoli rappresentano le dimensioni di analisi e le celle contengono un valore per ciascuna misura. È importante notare come il cubo in questione sia *sparso*, infatti non a tutte le coordinate corrisponde un evento: non tutti i prodotti vengono spediti ad ogni data e con ogni modalità.

Generalmente ad ogni dimensione viene associata una gerarchia di livelli di aggregazione, questi livelli vengono anche chiamati attributi dimensionali e

		Year		
		2010	2011	2012
City	Miami	47	45	50
	Orlando	44	43	52
	Tampa	39	50	41
	Washington	47	45	51
	Richmond	43	46	49
	Arlington	—	47	52

Figura 1.2: Esempio di una pivot table.

rappresentano un determinato modo di aggregare i valori della dimensione. Ad esempio i prodotti possono essere raggruppati in tipi che a loro volta sono raggruppati in categorie.

Prima di concludere questa parte riguardante il modello multidimensionale è utile far notare che esistono terminologie alternative a quelle utilizzate in questa sede, tali terminologie sono importanti in quanto utilizzate anche da prodotti commerciali ampiamente utilizzati. I termini *fatto* e *cubo* sono a volte usati in modo intercambiabile. Le misure sono anche dette *variabili*, *metriche*, *proprietà*, *attributi*, *indicatori*. Infine con il termine *dimensione* si indica a volte l'intera gerarchia di aggregazione anziché il livello più fine di tale gerarchia.

1.2 Analisi OLAP

L'acronimo *OLAP* sta per *On-Line Analytical Processing* e identifica un insieme di tecniche utilizzate per la fruizione di grandi quantità di dati, memorizzate ad esempio in un DW.

Assieme alla reportistica l'analisi OLAP è l'approccio più diffuso per accedere alle informazioni contenute in un DW ed è caratterizzato dalla possibilità di esplorare interattivamente i dati in sessioni complesse in cui ciascun passo è conseguenza dei risultati ottenuti ai passi precedenti.

Una sessione OLAP è quindi descrivibile come un *percorso di navigazione* costituito da una sequenza di interrogazioni al DW o per differenza dei risultati dei passi precedenti. Ad ogni passo di una sessione si applica un *operatore OLAP* per trasformare il risultato del passo precedente in un'altra interrogazione. Gli operatori OLAP più diffusi sono: *roll-up*, *drill-down*, *slice-and-dice*, *pivoting*, *drill-across*, *drill-through*.

- *Roll-up* significa letteralmente arrotolare o alzare, e induce un aumento nell'aggregazione dei dati eliminando un livello di dettaglio da una gerarchia. Applicando l'operatore roll-up è possibile eliminare alcune informazioni in quanto si riduce il livello di granularità.

- *Drill-down* è l'operatore opposto a roll-up, infatti diminuisce il livello di aggregazione dei dati aumentando quindi il livello di granularità. In questo caso, a differenza di roll-up, è possibile aggiungere informazioni a quelle già presenti.

- *Slice-and-dice* è un termine ambiguo utilizzato a volte per indicare le operazioni di selezione e proiezione, altre volte invece per indicare l'intero processo di navigazione OLAP.

- L'operazione di *Pivoting* non comporta un cambiamento dei dati ma semplicemente un cambiamento di prospettiva. Tornando alla metafora

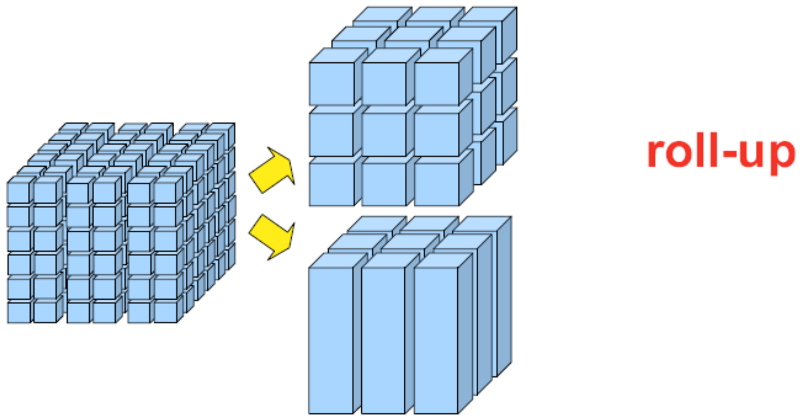


Figura 1.3: Operatore roll-up.

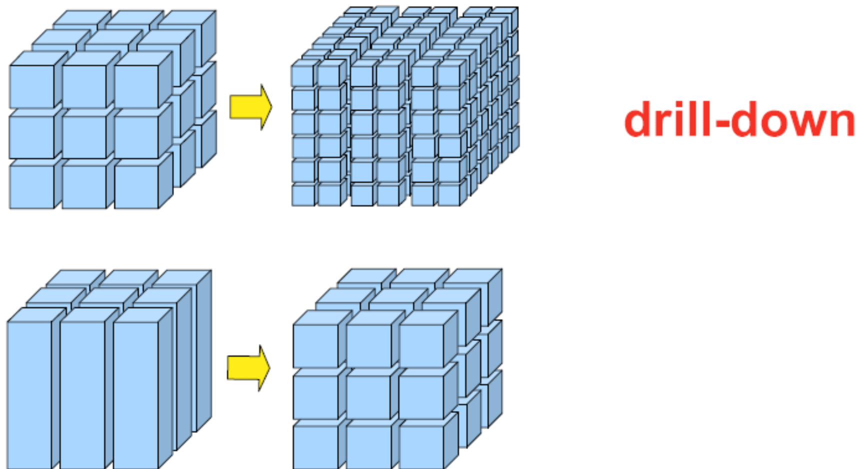


Figura 1.4: Operatore drill-down.

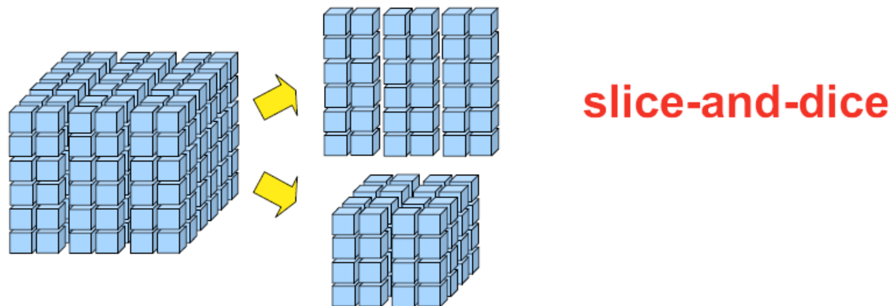


Figura 1.5: Operatore slice-and-dice.

dell'ipercubo è possibile interpretare questa operazione come la rotazione dell'ipercubo per portare in primo piano una diversa dimensione.

- Con il *Drill-across* si stabilisce un collegamento tra le celle di due o più cubi diversi per poterne comparare i valori. Il collegamento può essere anche l'applicazione di una funzione alle misure delle diverse celle.
- Tramite il *Drill-through* è possibile passare dai dati memorizzati nel DW ai dati operazionali presenti nelle sorgenti o nel livello riconciliato.

Data la difficoltà nel rappresentare efficacemente dati in più di tre dimensioni tramite grafici, spesso si ricorre a presentazioni in forma tabellare. In particolare uno strumento molto utilizzato anche in una sessione OLAP è la *pivot table*, ovvero una tabella contenente dati su cui è possibile effettuare diverse operazioni matematiche e statistiche, come: media, somma, massimo, minimo, deviazione etc.

La popolarità della pivot table deriva in gran parte alla semplicità e flessibilità di cui questo strumento gode, inoltre diversi prodotti commerciali ne

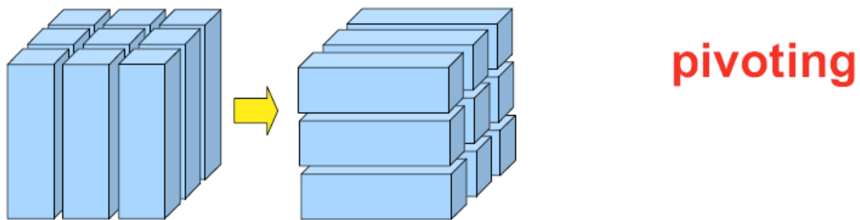


Figura 1.6: Operatore pivot.

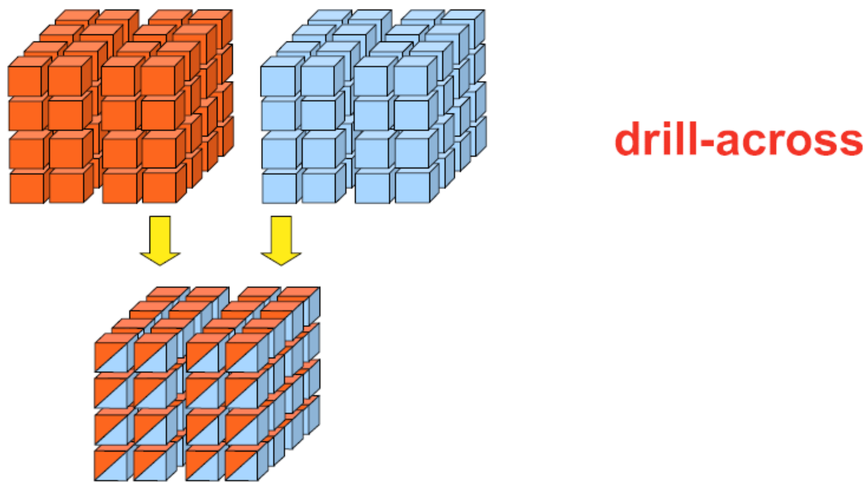


Figura 1.7: Operatore drill-across.

propongono un'implementazione, su tutti si cita il popolare Microsoft Excel (2013) che è ormai a tutti gli effetti utilizzato anche come front-end per analisi OLAP.

1.3 Rappresentazione approssimata tramite shrink

Una delle problematiche riscontrabili durante una sessione OLAP è quella di non riuscire ad ottenere la giusta quantità di informazioni che permetta di identificare pattern interessanti—generali e dettagliati—mantenendo comunque al minimo lo sforzo di analisi richiesto all'utente.

Il problema appena descritto è particolarmente evidente quando per la presentazione dei dati vengono utilizzate pivot table (che come già detto sono lo strumento più utilizzato per la rappresentazione testuale) infatti in questo caso è di importanza critica riuscire ad ottenere un compromesso soddisfacente tra dimensione del risultato presentato e precisione dei dati. Con un livello dettagliato dei dati si hanno a disposizione più informazioni ma ciò rende complesso il lavoro di analisi dell'utente e nasconde trend generali, d'altra parte concentrarsi su dati meno dettagliati impedisce l'individuazione di fenomeni su piccola scala.

Alcune delle soluzioni adottate per alleviare questo problema sono:

- *Semi-static reporting*: ovvero vengono posti dei limiti all'esplorazione dell'ipercubo per evitare ad esempio quei percorsi che portano ad un livello di dettaglio troppo alto.

- *Query personalization*: gli utenti sono in grado di esprimere delle preferenze sulle informazioni contenute nel risultato di una query OLAP [GRB11].

- *Intensional query answering*: i dati risultanti di una query vengono riassunti tramite una descrizione delle proprietà in comune [MMR12].
- *Approximate query answering*: il cui scopo è quello di migliorare i tempi di query restituendo però informazioni imprecise [VW99].
- *OLAM—On-Line Analytical Mining*: ovvero l'utilizzo di tecniche di data mining durante una sessione OLAP per estrarre pattern interessanti al costo però di una maggiore complessità computazionale [Han97].

La soluzione descritta in questa tesi rientra nella categoria OLAM ed è basata sul clustering gerarchico agglomerativo. Si propone un nuovo operatore OLAP chiamato *shrink* la cui idea di base è quella di cercare di ridurre la dimensione del risultato di una query OLAP (presentato idealmente tramite pivot table) cercando di minimizzare la perdita di precisione dei dati. La dimensione del risultato dell'operazione è limitata da un apposito parametro dato in input.

L'operatore shrink partendo da un generico ipercubo cerca di costruire cluster di slice simili rispettando i vincoli imposti dalle gerarchie. Ogni cluster così creato viene quindi rappresentato tramite un singolo slice calcolato come la media dei dati originali. Lo slice risultante può essere quindi visto come il centroide del cluster. Questa descrizione, seppur breve ed informale, introduce alcuni concetti fondamentali dell'operatore shrink che necessitano una spiegazione più approfondita per essere compresi.

La scelta di limitare la costruzione dei cluster tramite i vincoli gerarchici è dovuta in primo luogo alla volontà di presentare un risultato chiaro e facilmente leggibile. Inoltre l'imposizione dei vincoli gerarchici abbassa notevolmente il costo computazionale necessario alla ricerca e creazione dei cluster, per maggiori dettagli si rimanda al capitolo 2. Infine potrebbe essere sensato affermare che, ad esempio, due città italiane siano più simili tra loro che non una italiana con una francese, questo a prescindere dai valori presentati poiché in questo modo si rispetta la semantica delle informazioni.

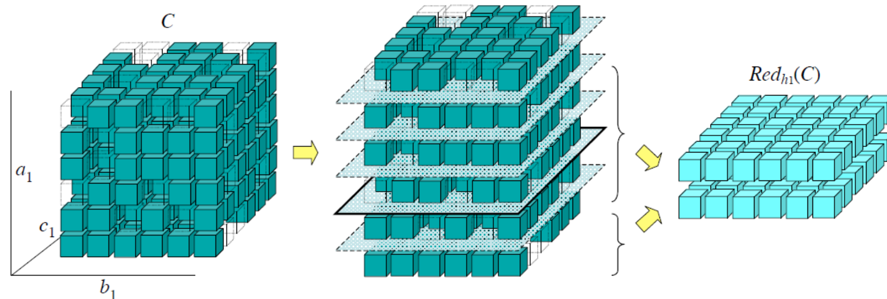


Figura 1.8: L'operatore shrink.

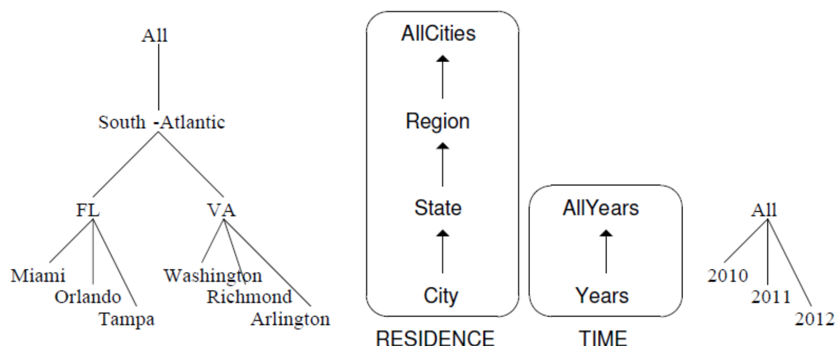


Figura 1.9: Esempio di una gerarchia e dei relativi roll-up.

Per spiegare meglio cosa si intende con vincoli gerarchici prendiamo in considerazione la gerarchia *RESIDENCE* mostrata in Figura 1.9. Per poter aggregare *Miami* con *Richmond* è necessario prima aggregare sia *Miami*, *Orlando*, *Tampa*—*FL*—che *Washington*, *Richmond*, *Arlington*—*VA*. È possibile invece aggregare liberamente tra loro Miami, Orlando e Tampa.

Un altro punto fondamentale è quello della computazione del cluster e della definizione dei loro rappresentanti—o centroidi. Nella descrizione presentata sopra si usano i termini *cluster di slice*, infatti l'ipercubo viene prima affettato lungo una determinata gerarchia e gli slice risultanti diventeranno gli elementi costituenti dei vari cluster. I motivi per cui si è scelto di utilizzare

questo approccio sono sostanzialmente due: il primo è quello di riuscire ad avere così un risultato che si presta bene alla visualizzazione tramite pivot table, il secondo è che considerando slice di questo tipo il costo computazionale rimane relativamente contenuto. La scelta di rappresentare un cluster tramite la media dei loro elementi è sembrata quella più naturale e intuitiva, si presta inoltre bene al calcolo misura di distanza/dissimilarità scelta, ovvero SSE—Sum of Squared Errors.

Se si utilizza un criterio di costruzione dei cluster come descritto fino a questo momento l'operatore shrink è più precisamente chiamato *fine shrink*, se invece si impone un ulteriore vincolo di aggregazione che obbliga ad aggregare tutti gli slice relativi ad un determinato attributo della gerarchia—ad esempio tutte le città della regione Emilia-Romagna—l'operatore viene chiamato *coarse shrink*.

Le motivazioni che hanno spinto verso lo sviluppo della versione coarse, oltre che a quella fine, sono due:

- Semplicità di visualizzazione: l'operatore fine shrink presenta slice più concisi ma non sempre riesce a fare lo stesso con le etichette corrispondenti, infatti aggregando n slice è possibile avere fino a n etichette.

- Efficienza: aggiungendo un ulteriore vincolo all'aggregazione degli slice si riduce sensibilmente lo spazio delle soluzioni ammissibili e questo di conseguenza impatta sulla complessità computazionale degli algoritmi.

L'approccio coarse offre quindi una diversa *granularità* nella riduzione dell'ipercubo che da un lato porta evidenti vantaggi, ovvero una computazione e una presentazione più snella, dall'altro perde però in efficacia, infatti la granularità più grossolana impedisce di avere quella precisione che invece ha il fine e di conseguenza i risultati dell'operatore coarse saranno caratterizzati da un maggiore errore di approssimazione. Per maggiori dettagli sulle prestazioni delle due versioni di shrink si rimanda al capitolo 4.

1.4 Approcci in letteratura

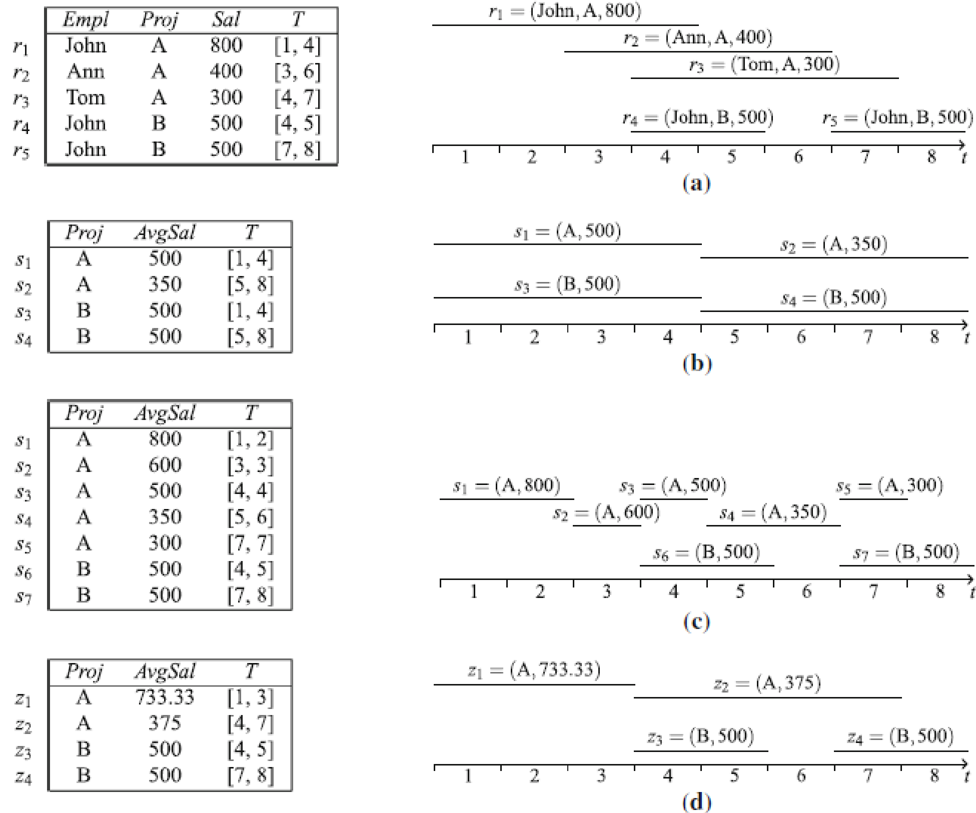
Una problematica simile a quella descritta sopra si presenta in molti ambiti, generalmente in tutte quelle situazioni in cui è necessario ottenere informazioni—anche approssimate—da una mole di dati talmente grande tale da renderne difficile una lettura completa. In questa sezione si effettua una panoramica su due scenari in particolare: quello dello studio di dati temporali e quello OLAP.

I dati temporali sono molto comuni in molti settori, infatti spesso si affianca ai dati registrati una validità temporale, che sia essa un istante di tempo o un intervallo. Proprio questa abbondanza di dati temporali ha suscitato un forte interesse per lo sviluppo di tecniche di *temporal aggregation*—aggregazione temporale. Le due forme di aggregazione temporale più studiate sono *ITA—Instant Temporal Aggregation*—e *STA—Span Temporal Aggregation*.

Nell'approccio ITA [BGJ06], [KS95], [MFVLI03] si calcola un dato aggregato in funzione di tutti i valori che hanno validità temporale in un determinato istante, in seguito i dati così ottenuti vengono fusi se appartenenti ad istanti consecutivi e con valori identici. Il risultato finale è una serie di valori accompagnati da un range temporale in cui valgono. Dato che i dati in input possono avere validità temporale sovrapposte, il risultato finale può essere di grandezza doppia rispetto ai dati originali, ovviamente questo è in contrasto con l'idea di aggregazione intesa come sommarizzazione.

Con STA [BGJ06], [KS95] è possibile invece specificare gli intervalli temporali in cui aggregare i dati, ad esempio per quadrimestre, annuo, biennio etc. In questo modo la dimensione del risultato può essere ridotta anche considerevolmente e in modo prevedibile, però è difficile trovare buone approssimazioni poiché i dati vengono aggregati senza tenere conto dei loro valori.

Una soluzione interessante che cerca di superare gli svantaggi dei due metodi appena descritti è quella di *PTA* [GGB12]—*Parsimonious Temporal Aggre-*



	Empl	*Proj*	*Sal*	*T*
r_1	John	A	800	[1, 4]
r_2	Ann	A	400	[3, 6]
r_3	Tom	A	300	[4, 7]
r_4	John	B	500	[4, 5]
r_5	John	B	500	[7, 8]

	Proj	*AvgSal*	*T*
s_1	A	500	[1, 4]
s_2	A	350	[5, 8]
s_3	B	500	[1, 4]
s_4	B	500	[5, 8]

	Proj	*AvgSal*	*T*
s_1	A	800	[1, 2]
s_2	A	600	[3, 3]
s_3	A	500	[4, 4]
s_4	A	350	[5, 6]
s_5	A	300	[7, 7]
s_6	B	500	[4, 5]
s_7	B	500	[7, 8]

	Proj	*AvgSal*	*T*
z_1	A	733.33	[1, 3]
z_2	A	375	[4, 7]
z_3	B	500	[4, 5]
z_4	B	500	[7, 8]

Figura 1.10: Due esempi di aggregazione PTA.

gation. Il procedimento consiste nel calcolare un'aggregazione ITA che verrà poi ridotta fondendo tuple simili e adiacenti. In questo caso con adiacente non si intende solo temporalmente ma che appartengono anche ad uno stesso *gruppo di aggregazione*. Le aggregazioni in PTA sono calcolate come media pesata in base all'intervallo di tempo.

Per chiarire meglio il risultato di un'aggregazione mediante PTA, in Figura 1.10 possiamo trovare due esempi tratti direttamente da [GGB12]. In figura si considera una relazione con attributi *impiegato*, *progetto*, *salario*, *intervallo temporale*. (a) e (c) sono i grafici relativi alle situazioni iniziali dei due esempi mentre (b) e (d) rappresentano il risultato aggregato.

In ambito OLAP sono presenti numerose tecniche per il problema approxi-

mate query answering già citato nel capitolo precedente che si ricorda avere lo scopo di ottenere una risposta in tempi brevi anche al costo di ottenere informazioni imprecise. Brevemente alcuni approcci sono descritti di seguito.

Una rappresentazione concisa di una data serie di dati può essere genericamente creata tramite appositi metodi di campionamento. Come ci si potrebbe aspettare l'accuratezza di questi metodi aumenta con l'aumentare della dimensione del campione. Un esempio di metodo di questo tipo è dato da *aqua system*[AGPR99] che dopo aver campionato i dati calcola delle misure aggregate che verranno poi utilizzate per i risultati di query approssimate. Una caratteristica interessante di aqua system è che riesce a calcolare le informazioni di sintesi incrementalmente.

Esistono anche sistemi in grado di effettuare il campionamento *online*, come [HHW97] che anziché appoggiarsi a sintesi precalcolate campiona a tempo di query fino a quando non riesce a dare una risposta con un determinato grado di confidenza.

Le tecniche basate su *wavelet*—es. [VW99]—decompongono l'input per ottenere una sintesi dei dati che include una piccola collezione di coefficienti wavelet che richiedono poco spazio di memorizzazione. La sintesi così creata può essere sfruttata direttamente per rispondere alle interrogazioni.

Sono stati studiati anche approcci che sfruttano gli *istogrammi*, come ad esempio [PG99]. Un istogramma è creato partizionando la distribuzione delle frequenze dei valori in *bucket*. Per ognuno di questi bucket vengono memorizzate informazioni come la somma delle frequenze contenute, ottenendo così una rappresentazione sintetica dei dati sfruttabile per ottenere risultati approssimati.

In [FW02] viene proposto invece un metodo basato su clustering per comprimere un ipercubo che può poi essere utilizzato direttamente per ricavare il risultato di una query. Il procedimento di clustering è vincolato da una soglia di errore che ne diventa quindi l'upper-bound. L'idea di base è la seguente:

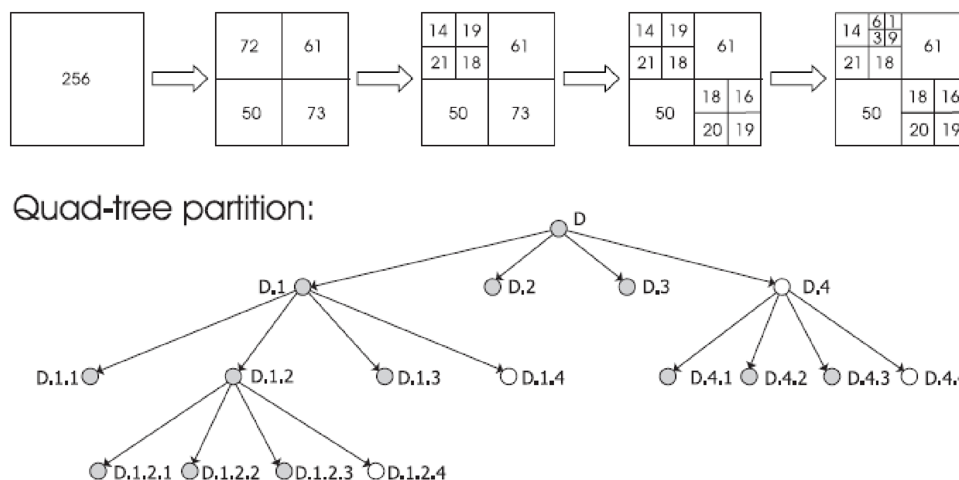


Figura 1.11: Esempio di partizione basata su quad-tree (immagine tratta da [BFS03]).

si suddivide l'ipercubo in tanti blocchi di dimensione omogenea che possono essere rappresentati come vettori di misure, in seguito si effettua il clustering di questi vettori. I cluster sono ora composti da vettori simili tra loro che possono essere rappresentati—sostituiti—dal centroide del cluster ottenendo così una riduzione di dell'ipercubo.

Gli autori di [BFS03] scelgono di concentrarsi su dati bi-dimensionali utilizzando una tecnica basata su quad-tree. La collezione bi-dimensionale di dati in input viene partizionata gerarchicamente costruendo un quad-tree le cui foglie rappresentano blocchi di dati originali in forma concisa. La scelta di utilizzare un partizionamento di questo tipo viene giustificata affermando che in questo modo si riesce a raggiungere una buona approssimazione mantenendo partizioni facili da descrivere e quindi piccole in termini di occupazione di memoria. La bontà delle partizioni è valutata utilizzando la metrica SSE. In Figura 1.11 viene mostrato un esempio di partizione basata su quad-tree.

In [CL12] viene presentata una possibile evoluzione (o variante) del metodo basato su quad-tree descritto sopra. Gli autori affermano che in [BFS03] non

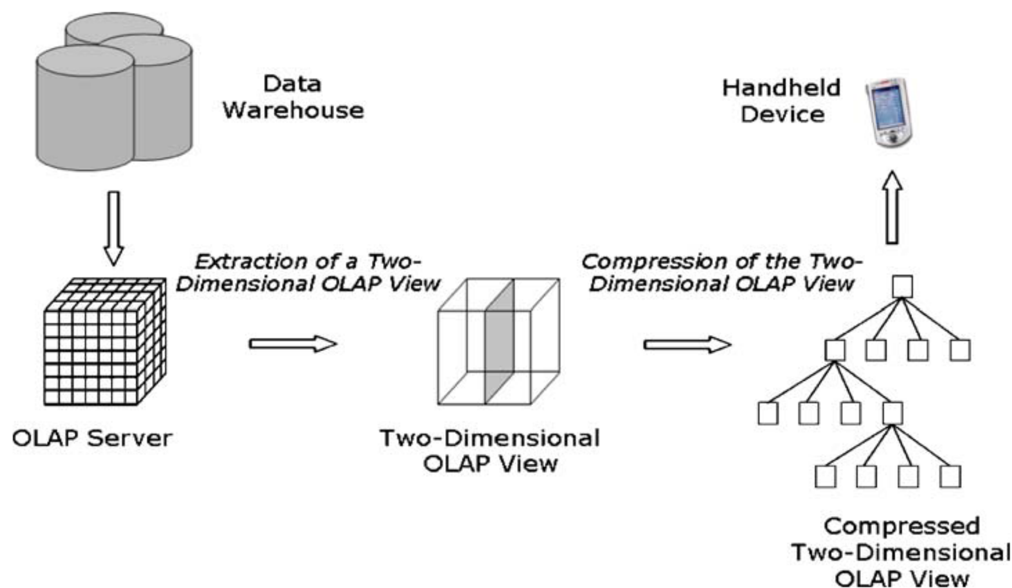


Figura 1.12: Procedimento di estrazione e compressione di una vista OLAP nel sistema Hand-OLAP (immagine tratta da [CFS09]).

vengono considerate ne le relazioni semantiche dei dati e gli interessi degli utenti finali. Il nuovo approccio è basato su R-Tree e ha la peculiarità di sfruttare un partizionamento *user-driven* in cui l'utente, dopo aver selezionato una *vista* bi-dimensionale tramite query OLAP, definisce due gerarchie in base ai propri interessi di analisi. In questo modo viene quindi definito un nuovo *cubo virtuale* utilizzato per creare le informazioni di sintesi.

Le tecniche di rappresentazione approssimata dei dati diventano di notevole importanza anche in ambito OLAP mobile poiché lo spazio di memorizzazione, le capacità di computazione e visualizzazione di informazioni sono presenti in forma molto più limitata rispetto ad architetture più tradizionali. A titolo di esempio si citano [LMBC12], [CFS09].

In particolare [CFS09] presenta *Hand-OLAP*, ovvero un sistema che permette di comprimere una vista OLAP 2D e di salvarla in dispositivi mobile per poter essere in seguito utilizzata per varie interrogazioni in locale (vedi Figura 1.12).

In questo modo si decentralizza il carico di lavoro spostando il calcolo delle query sui dispositivi mobile.

Capitolo 2

Operatore shrink

In questo capitolo viene data una formulazione dei problemi di shrinking e delle soluzioni proposte. Inizialmente—con la sezione 2.1—viene data una modellazione degli oggetti dello schema multidimensionale e del framework di shrinking. In 2.2 viene descritta la misura di errore utilizzata negli algoritmi sviluppati. Infine in 2.3 e 2.4 sono presenti le soluzioni ottimali e euristiche per risolvere i problemi fine shrinking e coarse shrinking.

2.1 Background

In questa sezione viene introdotta una formalizzazione basilare del modello multidimensionale e delle relative operazioni di manipolazione. Per semplicità vengono considerate solamente gerarchie senza ramificazioni e schemi con una sola misura. Vengono inoltre introdotte alcune definizioni specificamente correlate agli operatori di shrink.

2.1.1 Schema multidimensionale

Definizione 1 (Schema multidimensionale) *Uno* schema multidimensionale *(abbreviato* schema*)* $\mathcal{M}$ *è una coppia di*

- *un insieme finito di gerarchie disgiunte,* $\{h_1, \ldots, h_n\}$*, ognuna caratterizzata da un insieme* A_i *di attributi e un ordine di* roll-up $\prec_{h_i}$ *di* A_i*. Ogni attributo gerarchico* a *è definito su un dominio categorico* $Dom(a)$*.*

- *una famiglia di funzioni che, per ogni coppia di attributi* $a_k, a_j \in A_i$ *tali che* $a_k \prec_{h_i} a_j$*, roll-up ogni valore in* $Dom(a_k)$ *ad un valore in* $Dom(a_j)$*.*

Per semplificare la notazione, verrà utilizzata la lettera a per gli attributi h_1, lettera b per gli attributi di h_2, e così via; inoltre, verranno ordinati gli indici degli attributi in ogni gerarchia in base al loro ordine di roll-up: $a_1 \prec_{h_1} a_2 \prec_{h_1} \ldots$.

Un group-by include un livello per ogni gerarchia e definisce un possibile modo di aggregare i dati.

Definizione 2 (Group-by) *Un* group-by *di uno schema* $\mathcal{M}$ *è un elemento* $G \in A_1 \times \ldots \times A_n$*. Una* coordinata *di* $G = \langle a, b, \ldots \rangle$ *è un elemento* $g \in Dom(a) \times Dom(b) \times \ldots$*.*

Esempio 1 IPUMS è un database pubblico contenente informazioni utili per ricerce economiche e sociali [Min08]. Prendiamo come esempio una forma semplificata dello schema multidimensionale CENSUS basato su due gerarchie: $\mathsf{RESIDENCE}$ e TIME. In $\mathsf{RESIDENCE}$ abbiamo che $\mathsf{City} \prec_{\mathsf{RESIDENCE}} \mathsf{State}$, $RollUp^{\mathsf{State}}_{\mathsf{City}}(\text{Miami}) = \text{FL}$, e $Drill(\mathsf{City}, \{\text{FL}\}) = \{\text{Miami}, \text{Orlando}, \text{Tampa}\}$ (Figura 1.9).

Alcuni esempi di group-by sono $G_1 = \langle \mathsf{City}, \mathsf{Year} \rangle$ and $G_2 = \langle \mathsf{State}, \mathsf{AllYears} \rangle$. Una coordinata di G_1 è $\langle \text{Miami}, 2012 \rangle$.

Un'istanza di uno schema è un insieme di fatti; ogni fatto è caratterizzato da un group-by G che definisce il suo livello di aggregazione, da una coordinata di G e da un valore numerico m.

L'operazione di shrink verrà applicata ad un cubo—un sottoinsieme di fatti risultante da una query OLAP. Questo può essere formalizzato come segue:

Definizione 3 (Cube) *Un* cubo *al group-by G è una funzione parziale C che mappa una coordinata di G ad un valore numerico (*misura*). Ogni coppia $\langle g, m \rangle$ tale che $C(g) = m$ è chiamata un* fatto *di C.*

La ragione per cui la funzione C è parziale è che i cubi sono generalmente *sparsi*—ovvero alcuni fatti hanno misura NULL. Un esempio di fatti mancanti è quello della città di Arlington nell'anno 2010 in Figura **??**.

Esempio 2 Due esempi di fatti di CENSUS sono $\langle\langle \text{Miami}, 2012\rangle, 50\rangle$ and $\langle\langle \text{Orlando, 2011}\rangle, 43\rangle$. La misura in questo caso quantifica la media dei salari dei cittadini. Un possibile cubo ad G_1 è rappresentato in Figura **??**.

2.1.2 Shrinking framework

Assumiamo per semplicità che l'operatore shrink venga applicato lungo gerarchia h_1 di un cubo C al suo livello di group-by più *fine* per h_1, $G = \langle a_1, b, c, \ldots \rangle$.

Introduciamo innanzitutto il concetto di cubo espresso come insieme di *slice*. Dato il valore di attributo $v \in Dom(a_1)$, il cubo C può essere riscritto come:

$$C = \{C^v, v \in Dom(a_1)\}$$

dove C^v è lo slice di C corrispondente a $a_1 = v$.

Applicando l'operazione di shrinking al cubo C gli slice di C vengono partizionati—in modo disgiunto e completo—in un insieme di cluster. Gli slice di un determinato cluster vengono fusi in un unico slice approssimato chiamato

f-slice calcolato come la media delle misure originali. Un f-slice risultante può quindi riferirsi ad un set di valori del dominio a_1.

Per formalizzare la riduzione di un cubo verranno utilizzate coordinate incomplete di G nella forma $\overline{g} \in Dom(b) \times Dom(c) \times \ldots$.

Definizione 4 (Slice) *Dato il cubo C al group-by $G = \langle a_1, b, c, \ldots \rangle$ e il valore di attributo $v \in Dom(a_1)$, lo* slice *di C corrispondente a $a_1 = v$ è una funzione che mappa ogni $\overline{g} \in Dom(b) \times Dom(c) \times \ldots$ o ad un valore numerico oppure a $NULL$, definita come:*

$$C^v(\overline{g}) = C(g),\ \forall g = \langle v, \overline{g} \rangle$$

Quindi uno slice è un cubo in cui la prima componente è fissata ad un singolo valore v.

Definizione 5 (F-Slice) *Dati due slice $C^{v'}$ è $C^{v''}$ di C, l'*f-slice *risultante dalla loro fusione è una funzione $F = C^{v'} \oplus C^{v''}$ che mappa ogni $\overline{g} \in Dom(b) \times Dom(c) \times \ldots$ o ad un valore numerico oppure a $NULL$, definita come:*

$$F(\overline{g}) = \begin{cases} NULL & \text{if } C^{v'}(\overline{g}) = C^{v''}(\overline{g}) = NULL \\ avg\{C^{v'}(\overline{g}), C^{v''}(\overline{g})\} & \text{otherwise} \end{cases}$$

assumendo per convenzione che $avg\{m, NULL\} = m$ per ogni $m \neq NULL$.

Questa definizione può essere estesa per funzionare anche con insiemi di slice. Il valore della misura di ogni fatto in F^V è calcolato come la media dei valori non nulli dei fatti corrispondenti agli slice appartenenti a $Drill_1(V)$.

Sia a_j, $j \geq 1$, un attributo di h_1, e sia $V \subseteq Dom(a_j)$ un insieme di valori di a_j. Definiamo con $Drill_1(V)$ the l'insieme di tutti i valori di a_1 che tramite roll-up risultano in un valore in V (per convenzione, se $j = 1$ sarà $Drill_1(V) = V$ per $V \subseteq Dom(a_1)$). L'operazione di shrinking prende in input un cubo C e restituisce una riduzione $Red_{h_1}(C)$ di C:

$$Red_{h_1}(C) = \{F^V, V \subseteq Dom(a_j), a_j \in A_1\}$$

tale che

$$F^V = \bigoplus_{v \in Drill_1(V)} C^v \tag{2.1}$$

$$\bigcup_{F^V \in Red_{h_1}(C)} Drill_1(V) = Dom(a_1) \tag{2.2}$$

$$Drill_1(V') \cap Drill_1(V'') = \emptyset, \forall F^{V'}, F^{V''} \in Red_{h_1}(C) \tag{2.3}$$

Se $V \subseteq Dom(a_j)$, l'f-slice F^V si dice *avere livello* j; f-slice con differenti livelli possono essere aggregati in una riduzione. La dimensione—*size*—di una riduzione è il numero di f-slice che include: $size(Red_{h_1}(C)) = |Red_{h_1}(C)|$.

Da notare che un f-slice può avere qualunque valore maggiore o uguale a 1 ma i vincoli (2.2) e (2.3) assicurano che la riduzione sia completa e disgiunta. Per preservare la semantica dettata dalla gerarchia, sono necessari ulteriori vincoli:

1. Due slice corrispondenti ai valori v e v' di a_1 possono essere fusi in un unico f-slice con livello j solo se entrambi v e v' aggregano—roll-up—allo stesso valore di a_{j+1}.

2. Se gli slice corrispondenti a tutti i valori di a_1 il cui roll-up corrisponde ad un singolo valore v^* di a_j ($j > 1$) sono *tutti* fusi assieme, allora il corrispondente f-slice ha livello j ed è associato nella riduzione con un insieme V tale che $v^* \in V$.

Una riduzione che soddisfa tali vincoli è detta *hierarchy compliant*—ovvero che rispetta la gerarchia.

Esempio 3 In Figura 2.1 sono rappresentate due possibili riduzioni del cubo in Figura 1.2. In (a) *Miami* e *Orlando* vengono fusi in un unico slice di livello 1 e *Washington*, *Richmond* e *Arlngton* vengono fusi in un f-slice *Virginia* di livello 2. Nel secondo esempio invece tutti e sei gli slice di partenza vengono fusi nell'f-slice *South-Atlantic* di livello 3.

City	2010	2011	2012
Miami, Orlando	45.5	44	51
Tampa	39	50	41
Virginia	45	46	50.6

(a)

	2010	2011	2012
South-Atlantic	44	46	49.2

(b)

Figura 2.1: Due esempi di riduzione di uno stesso cubo.

L'approccio descritto può essere generalizzato considerando come input un qualsiasi cubo $G = \langle a_k, \ldots \rangle$ risultante da una query OLAP, ovviamente gli f-slice risultanti avranno tutti livello maggiore o uguale a k.

2.2 Misurazione dell'errore di approssimazione

La fusione di alcuni slice in un unico f-slice rappresentante la riduzione comporta un'approssimazione. Come nel caso di [GGB12] viene utilizzato *SSE—Sum Squared Error*.

Dato un cubo C ad un group-by $G = \langle a_k, b, \ldots \rangle$, sia $V \subseteq Dom(a_j)$, $k \leq j$. Denotiamo con $Desc_k(V)$ l'insieme di tutti i valori di a_k che tramite roll-up risultano in un valore in V, e con F^V l'f-slice corrispondente. Sia $\overline{g} \in Dom(b) \times Dom(c) \times \ldots$ una coordinata incompleta di G (non è specificato nessun valore per l'attributo a_k). L'SSE associato ad F^V è

$$SSE(F^V) = \sum_{\overline{g} \in Dom(b) \times Dom(c) \times \ldots} \sum_{v \in Desc_k(V)} (C^v(\overline{g}) - F^V(\overline{g}))^2 \tag{2.4}$$

(convenzionalmente, $C^v(\overline{g}) - F^V(\overline{g}) = 0$ se $C^v(\overline{g})$ è nullo). Data la riduzione

$Red_{h_1}(C)$, l'SSE associato a $Red_{h_1}(C)$ è

$$SSE(Red_{h_1}(C)) = \sum_{\langle V, F^V \rangle \in Red_{h_1}(C)} SSE(F^V) \tag{2.5}$$

Esempio 4 L'SSE associato ai tre f-slice in Figura 2.1.a sono $(2.25+2.25)+(1+1)+(1+1) = 8.5$, 0, e 14.68 rispettivamente; l'SSE totale associato alla riduzione è 23.2. L'SSE associato alla riduzione in Figura **??**.b è 158.8.

Da notare come è possibile calcolare SSE in modo incrementale. Ad esempio l'SSE di un f-slice $F^{V' \cup V''}$ può essere calcolato a partire dall'SSE degli slice V' e V''.

Teorema 1 *Sia $Red_{h_1}(C)$ una riduzione di C, e siano $F^{V'}$ e $F^{V''}$ due f-slice di $Red_{h_1}(C)$, con*

$$F^{V'} = \bigoplus_{v \in Drill_k(V')} C^v$$
$$F^{V''} = \bigoplus_{v \in Drill_k(V'')} C^v$$

Sia

$$F^{V' \cup V''} = \bigoplus_{v \in Drill_k(V') \cup Drill_k(V'')} C^v$$

l'aggregazione di tutti gli slice in $F^{V'}$ e $F^{V''}$. Allora

$$SSE(F^{V' \cup V''}) = SSE(F^{V'}) + SSE(F^{V''}) + \times \sum_{\overline{g} \in Dom(b) \times Dom(c) \times \ldots} \frac{|H'_{\overline{g}}| \cdot |H''_{\overline{g}}|}{|H'_{\overline{g}}| + |H''_{\overline{g}}|} \cdot (F^{V'}(\overline{g}) - F^{V''}(\overline{g}))^2 \tag{2.6}$$

dove $H'_{\overline{g}} = \{v \in Drill_k(V')$ s.t. $C^v(\overline{g})$ is not $NULL\}$ (in modo simile a $H''_{\overline{g}}$).

Dimostrazione: Sia $H_{\overline{g}} = H'_{\overline{g}} \cup H''_{\overline{g}}$. Osserviamo che

$$F^V(\overline{g}) = \frac{\sum_{v \in H_{\overline{g}}} C^v(\overline{g})}{|H_{\overline{g}}|}$$

per definizione 5 (similarmente per V' e V''), allora

$$\begin{aligned}SSE(F^{V'\cup V''}) &= \sum_{\overline{g}} \sum_{v\in H_{\overline{g}}} (C^v(\overline{g}) - F^V(\overline{g}))^2 \\ &= \sum_{\overline{g}} \left[\sum_{v\in H_{\overline{g}}} (C^v(\overline{g}))^2 + |H_{\overline{g}}| \cdot F^V(\overline{g})^2 - 2F^V(\overline{g}) \sum_{v\in H_{\overline{g}}} C^v(\overline{g}) \right] \\ &= \sum_{\overline{g}} \left[\sum_{v\in H_{\overline{g}}} (C^v(\overline{g}))^2 - \frac{\left(\sum_{v\in H_{\overline{g}}} C^v(\overline{g})\right)^2}{|H_{\overline{g}}|} \right]\end{aligned}$$

Similarmente

$$SSE(F^{V'}) = \sum_{\overline{g}} \left[\sum_{v\in H'_{\overline{g}}} (C^v(\overline{g}))^2 - \frac{\left(\sum_{v\in H'_{\overline{g}}} C^v(\overline{g})\right)^2}{|H'_{\overline{g}}|} \right]$$

(lo stesso per V''). E infine

$$\begin{aligned}SSE(F^{V'\cup V''}) &= \sum_{\overline{g}} \left[\sum_{v\in H'_{\overline{g}}} (C^v(\overline{g}))^2 + \sum_{v\in H''_{\overline{g}}} (C^v(\overline{g}))^2 - \frac{\left(\sum_{v\in H_{\overline{g}}} C^v(\overline{g})\right)^2}{|H'_{\overline{g}} + H''_{\overline{g}}|} \right] \\ &= SSE(F^{V'}) + SSE(F^{V''}) \\ &\quad + \sum_{\overline{g}} \left[\frac{\left(\sum_{v\in H'_{\overline{g}}} C^v(\overline{g})\right)^2}{|H'_{\overline{g}}|} + \frac{\left(\sum_{v\in H''_{\overline{g}}} C^v(\overline{g})\right)^2}{|H''_{\overline{g}}|} - \frac{\left(\sum_{v\in H_{\overline{g}}} C^v(\overline{g})\right)^2}{|H'_{\overline{g}} + H''_{\overline{g}}|} \right] \\ &= SSE(F^{V'}) + SSE(F^{V''}) + \sum_{\overline{g}} \frac{|H'_{\overline{g}}| \cdot |H''_{\overline{g}}|}{|H'_{\overline{g}}| + |H''_{\overline{g}}|} \left(F^{V'}(\overline{g}) - F^{V''}(\overline{g})\right)^2\end{aligned}$$

□

2.3 Operatore fine shrink

Dato un cubo C, esistono svariate riduzioni applicabili, una per ogni modo di partizionare gli slice preservando i vincoli imposti dalle gerarchie. Ovviamente più si porta avanti il processo di riduzione, più la soluzione sarà caratterizzata da un minor numero di slice e da un maggior errore di approssimazione. È quindi necessario un parametro α che stabilisca un certo trade-off tra dimensione e precisione dei risultati.

Nel corso di questo studio vengono presi in considerazione due utilizzi del parametro α: (i) stabilire la dimensione massima che il risultato deve avere, (ii) definire il massimo errore tollerabile. Questi due utilizzi del parametro α definiscono due possibili problemi formulati di seguito.

Problema di riduzione 1 Trova una riduzione che minimizzi la dimensione tra quelle il cui SSE non è maggiore di α_{SSE}.

Problema di riduzione 2 Trova una riduzione che minimizzi l'SSE tra quelle la cui dimensione non è maggiore di α_{size}.

D'ora in avanti ci riferiremo a questi due problemi anche come *SSE-bound* e *size-bound* rispettivamente.

Entrambi i problemi descritti possono essere visti come problemi di clustering con vincoli. Gli oggetti su cui applicare il clustering sono gli slice del cubo e la dimensione del risultato è data dal numero di cluster creati. I vincoli da rispettare in questo caso sono quelli gerarchici (o semantici) e, nei limiti della nostra conoscenza, al momento non esistono tecniche di clustering per questo tipo di restrizioni; il vincolo più simile al nostro caso è quello *cannot-link* e stabilisce che due oggetti non possono essere messi insieme nello stesso cluster. Questo tipo di vincolo è però definito *staticamente* mentre nel nostro caso alcuni vincoli si *rilassano* procedendo con il processo di riduzione. Ad esempio una città italiana non può essere aggregata con una tedesca a meno che tutte le città italiane e quelle tedesche rispettivamente non siano già aggregate.

Prima di descrivere gli algoritmi studiati, passiamo a dare una stima dello spazio di ricerca. Per semplicità viene preso in considerazione solo il caso peggiore in cui nessun vincolo gerarchico è imposto. Il numero di partizionamenti possibili in questo caso è definito dal numero di Bell, che è calcolato ricorsivamente come

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

dove $B_0 = B_1 = 1$. È chiaro quindi che per poter dare risposte in tempi brevi un approccio ottimale sia inadeguato, per questo nelle prossime sezioni vengono descritti sia un algoritmo ottimale che uno euristico che riduce drasticamente le risorse computazionali richieste.

Da notare come i due problemi di riduzione possano essere visti entrambi come problemi di clustering con vincoli. La differenza principale tra i due è che il Problema 2 fissa la dimensione massima risultante—numero di slice—e può essere visto quindi come un un problema di clustering *k-means*. Questo non è vero per il Problema 1. Per semplicità risolviamo il Problema 1 e adottiamo un approccio basato su clustering agglomerativo che può essere adattato ad entrambi i problemi con facilità.

2.3.1 Algoritmo branch-and-bound per fine shrink

La soluzione ottimale per il problema SSE-bound può essere ottenuta tramite un'enumerazione sistematica di tutte le partizioni degli slice di partenza che soddisfano i vincoli gerarchici. L'enumerazione descritta si basa su quella proposta in [LL08] che però non considera i vincoli gerarchici.

Come mostrato nell'Algoritmo 1 l'enumerazione è agglomerativa: data una gerarchia h, si inizia dalla partizione radice $P_0 = \{S_0, \ldots, S_n\}$ in cui ogni valore di a_1 definisce un diverso elemento S_i.

In Figura 2.2 è mostrato un esempio di albero delle partizioni creato tramite la procedura *FineChild*.

Nell'Algoritmo 1 due cluster—f-slice—S_j e S_i possono essere fusi se e solo se tutte e tre le seguenti condizioni sono soddisfatte. Ad ogni condizione segue una breve descrizione dell'intuizione dietro di essa.

1. S_i contiene un singolo valore v. Supponiamo che $S_i = \{v_1, v_2\}$, allora la partizione figlia con S_j e S_i aggregati insieme può essere generata altrove nell'albero unendo S_j prima con $\{v_1\}$ e in seguito con $\{v_2\}$.

Algoritmo 1 L'enumerazione FineChildren

```
Require: Un partizionamento P = {S_1, ..., S_t} su Dom(a_1)
Require: L'indice k dell'ultimo insieme aggregato in P
Ensure: L'insieme T delle partizioni figlie di P
 1: T ← ∅                                                   ▷ Inizializza il risultato
 2: for all i = k + 1 to t do
 3:     if S_i contains one element v then                  ▷ Condizione 1
 4:         for all j = i − 1 to 1 do
 5:             if v > any of the elements in S_j then      ▷ Condizione 3
 6:                 P_New = {S_1, S_j ⋃ S_i, , ..., S_t}
 7:                 if IsCompliant(S_j ⋃ S_i) then          ▷ Verifica i vincoli gerarchici
 8:                     T = T ⋃ P_New
 9:                 else
10:                     T = T ⋃ Recover(P_New)              ▷ Forza i vincoli gerarchici
11:             else
12:                 break;
13:             if |S_j| > 1 then                           ▷ Condizione 3
14:                 break;
15: return T
```

2. Ogni cluster tra S_j e S_i—ovvero $[S_{j+1}, \ldots, S_{i-1}]$—contiene un singolo elemento. Supponiamo che esista un k tale che $j+1 \leq k \leq i-1$ e S_k contenga più di un elemento, allora altrove nell'albero, abbiamo una partizione in cui S_k è sostituito da diversi insiemi, ognuno dei quali contiene un elemento. La partizione con S_j e S_i uniti sarà generata li.

3. Ogni elemento di S_j è minore di v. Supponiamo che $S_j = \{v_1, v_2\}$ con $v_1 < v < v_2$, allora la partizione con S_j e S_i è generabile altrove con $\{v_1\}$ prima unito a e e in seguito con v_2. Notare che S_j deve contenere un elemento che è minore di v poiché S_j viene prima di S_i.

Una dimostrazione di correttezza e sistematicità dell'enumerazione senza i vincoli gerarchici è disponibile in [LL08].

La funzione IsCompliant (vedi riga 7 dell'Algoritmo 1) verifica che il cluster rispetti i vincoli gerarchici. Dato che P soddisfa tali vincoli, la violazione deriva dall'ultima unione $S_j \bigcup S_i$. Un controllo di ammissibilità può essere implementato come segue. Sia $v_1 \in S_j$ e $v_2 \in S_i$, affinché la fusione sia valida deve essere rispettata la seguente condizione:

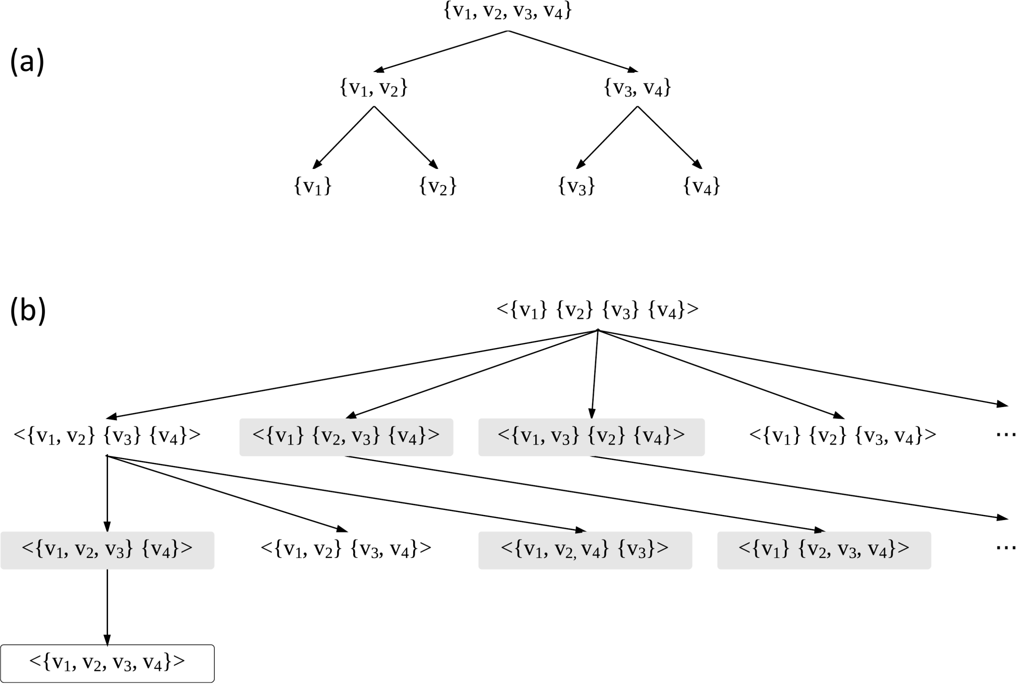


Figura 2.2: In (b) abbiamo alcune delle partizioni ottenute dai valori gerarchici mostrati in (a). Le partizioni con sfondo grigio non soddisfano i vincoli gerarchici, mentre le partizioni bordate sono ammissibili ma ottenute da partizioni non ammissibili.

$$Desc((HUA(v_1, v_2)) \bigcup Desc((HUA(v_2, v_1)) = S_i \bigcup S_j \tag{2.7}$$

dove $Desc(x)$ restituisce tutti valori foglia nel sotto-albero x, $HUA(v_1, v2)$ restituisce il più alto antenato non comune di v_1 rispetto a v_2.

Esempio 5 In Figura 2.2 il cluster $\{v_2, v_3\}$ non soddisfa i vincoli gerarchici poiché gli HUAs di v_2 e v_3 sono $\{v_1, v_2\}$ e $\{v_3, v_4\}$ rispettivamente e i loro discendenti sono $\{v_1\} \bigcup \{v_2\} \bigcup \{v_3\} \bigcup \{v_4\} \neq \{v_2, v_3\}$.

Anche se P non soddisfa i vincoli, uno dei suoi discendenti—nell'albero delle

enumerazioni—potrebbe essere una partizione valida (vedi Figura ??). Tali partizioni devono essere generate, altrimenti l'enumerazione non sarebbe più completa. A partire da una partizione P non ammissibile è possibile *saltare* direttamente alla partizione P' più vicina—nell'albero di enumerazione—e ammissibile. La partizione P' che soddisfa l'equazione 2.7 può essere ottenuta nel seguente modo: siano S_j e S_i gli ultimi cluster fusi con $v_1 \in S_j$ e $v_2 \in S_i$,

$$\begin{aligned} P' = \\ & P \cup (Descendants((HUA(v_1, v_2)) \cup Descendants((HUA(v_2, v_1)))- \\ & \bigcup S \in P \; s.t. \\ & S \cap (Descendants((HUA(v_1, v_2)) \cup Descendants((HUA(v_2, v_1))) \\ & \neq \emptyset \end{aligned} \tag{2.8}$$

Prima di restituire P', Recover deve verificare che sarà generata nel sotto-albero di P secondo le condizioni di enumerazione descritte precedentemente. P' può essere generata se e solo se $\forall v_k \in P' - P$ valgono le seguenti condizioni:

1. v_k è un *singleton* in P
2. $v_k > v_i, \forall v_i \in S_j$ in P

Per ridurre il numero di partizioni visitate nell'albero delle enumerazioni viene adottato un approccio branch-and-bound. I sotto-alberi corrispondenti alle partizioni il cui SSE è superiore alla miglior soluzione attuale vengono potati poiché produrrebbero solo soluzioni peggiori di quella attuale. Il criterio di ricerca adottato è un best-first search—BSF—in cui si sceglie di volta in volta il nodo con il maggior numero di riduzioni e in caso vi siano più nodi con lo stesso numero di riduzioni, tra questi si sceglie quello con minor SSE. In pratica questo criterio assomiglia molto ad una depth-first search e permette di contenere le risorse di memoria richieste. Sono stati testati anche

Algoritmo 2 Algoritmo ottimale Fine Shrink—SSE-bound

```
Require: P_0
Require: α_SSE
Ensure: P_Opt
 1: Stack.Push(P_0)                          ▷ Inizializza lo stack
 2: P_Opt ← ∅                                ▷ Inizializza la soluzione
 3: while P ← StackPop() do
 4:     T = FineChildren(P)
 5:     for all P_Child ∈ T do
 6:         if SSE(P_Child) > α_SSE then
 7:             break;                       ▷ P_Child e i suoi sotto-alberi vengono potati
 8:         else
 9:             if |P_Child| < |P_Opt| then
10:                 P_Opt ← P_Child          ▷ Nuova soluzione
11:             R∪ = P_Child
12: Stack.Push(BestOrder(R))   ▷ Applica localmente un criterio best-first e aggiunge le partizioni allo
    stack
13: return P_Opt
```

altri criteri che però risultati troppo costosi in termini di memoria data la dimensione dello spazio di ricerca.

L'Algoritmo 2 descrive l'approccio ottimale per il problema fine shrink.

2.3.2 Algoritmo greedy per fine shrink

In questa sezione viene mostrato come sia possibile risolvere i due problemi di riduzione già presentati tramite un algoritmo di clustering agglomerativo con vincoli gerarchici.

Il *clustering gerarchico* ha lo scopo di costruire una gerarchia di cluster [TS06]; questo può essere fatto seguendo un approccio top-down oppure un approccio bottom-up. In quest'ultimo caso vengono chiamati *agglomerativi*: ogni elemento—ogni slice del cubo C nel nostro caso—inizialmente costituisce un proprio cluster, in seguito coppie di cluster vengono uniti man a mano che si procede. Il criterio utilizzato per decidere quale coppia di cluster unire è tipicamente greedy. Nel nostro caso unire due cluster significa unire due f-slice, ovvero fondere tutti gli slice corrispondenti ai due f-slice in un unico f-

Algoritmo 3 Algoritmo greedy Fine Shrink—size-bound

Require: C al group-by $G = \langle a_k, b, \ldots \rangle$, $size_{max}$
Ensure: $Red_{h_1}(C)$
1: $Red_{h_1}(C) \leftarrow \emptyset$ ▷ Inizializza la riduzione...
2: **for all** $v \in Dom(a_k)$ **do** $Red_{h_1}(C) \leftarrow Red_{h_1}(C) \cup \{C^{\{v\}}\}$ ▷ ...uno slice per ogni f-slice
3: **while** $size(Red_{h_1}(C)) > size_{max}$ **do** ▷ Controlla il vincolo di dimensione massima
4: find $F^{V'}, F^{V''} \in Red_{h_1}(C)$ s.t. $F^{V'}$ and $F^{V''}$ have the same level j
5: **and** $\Delta SSE = SSE(F^{V' \cup V''}) - SSE(F^{V'}) - SSE(F^{V''})$ is minimal
6: **and** all values in $V' \cup V''$ roll-up to the same value of a_{j+1} ▷ Vincoli gerarchici
7: $Red_{h_1}(C) \leftarrow Red_{h_1}(C) \setminus \{F^{V'}, F^{V''}\}$ ▷ Fonde in un unico f-slice...
8: **if** $\exists \overline{v} \in Dom(a_{j+1})$ s.t. $Desc_j(\overline{v}) = V' \cup V''$ **then**
9: $Red_{h_1}(C) \leftarrow Red_{h_1}(C) \cup \{F^{\{\overline{v}\}}\}$ ▷ ...o al livello $j+1$
10: **else**
11: $Red_{h_1}(C) \leftarrow Red_{h_1}(C) \cup \{F^{V' \cup V''}\}$ ▷ ...oppure al livello j
12: **return** $Red_{h_1}(C)$

slice. Il criterio che adottiamo è il *Ward's minimum variance method* [TS06]: ad ogni passo uniamo la coppia di f-slice che porta ad un minor incremento di ΔSSE, nell'SSE totale della corrispondente riduzione. Inoltre i due cluster vengono uniti solo se il risultato rispetta i vincoli gerarchici. Il processo di clustering viene interrotto in base ai vincoli espressi da SSE_{max} o $size_{max}$. Il procedimento è descritto nell'Algoritmo 3.

Esempio 6 Consideriamo il cubo descritto in Figura 1.2. Di seguito viene descritto il processo di riduzione step-by-step con limite $size_{max} = 3$ (Figura 2.3).

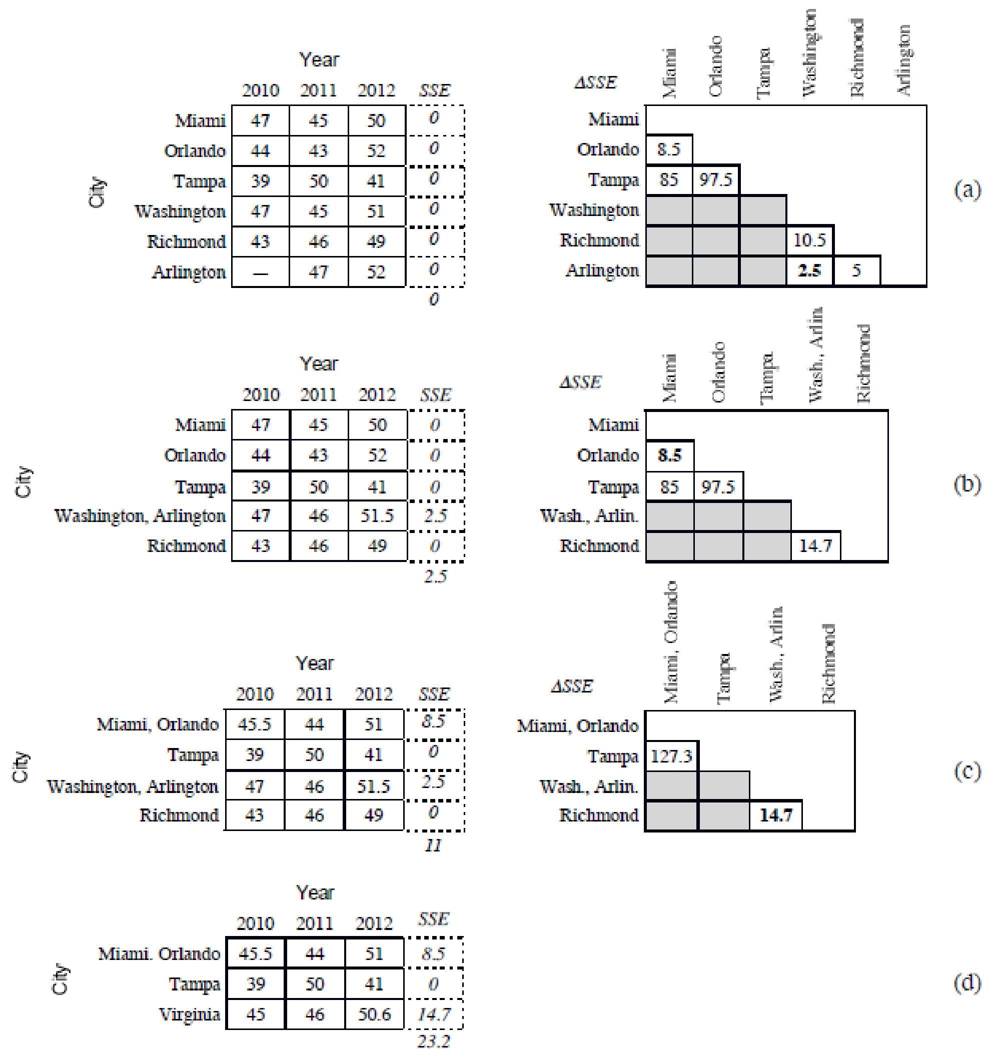
1. Nel passo di inizializzazione, (riga 2), vengono creati sei f-slice *singleton*, uno per ogni slice in C. Dato che questa prima riduzione ha dimensione 6 viola il vincolo $size_{max}$ e quindi si entra nel ciclo *while*.

2. La fusione più promettente è quella tra Arlington e Washington, che incrementa l'SSE totale di $\Delta SSE = 2.5$ (Figura 2.3.a, destra).

3. Nella riduzione risultante (Figura 2.3.b, sinistra), che viola ancora il vincolo di dimensione, l'unione più promettente è quella tra Miami e Orlando. (Figura 2.3.b, destra).

4. A questo punto, Richmond è unito con Washington-Arlington (Figura 2.3.c, destra). Dato che l'f-slice risultante copre tutta la Virginia, il suo livello diventa 2. (Figura 2.3.d).

5. La riduzione risultante soddisfa i vincoli, quindi l'algoritmo termina.



(a)

City	2010	2011	2012	*SSE*
Miami	47	45	50	*0*
Orlando	44	43	52	*0*
Tampa	39	50	41	*0*
Washington	47	45	51	*0*
Richmond	43	46	49	*0*
Arlington	—	47	52	*0*
				0

ΔSSE	Miami	Orlando	Tampa	Washington	Richmond	Arlington
Miami						
Orlando	8.5					
Tampa	85	97.5				
Washington						
Richmond				10.5		
Arlington				**2.5**	5	

(b)

City	2010	2011	2012	*SSE*
Miami	47	45	50	*0*
Orlando	44	43	52	*0*
Tampa	39	50	41	*0*
Washington, Arlington	47	46	51.5	*2.5*
Richmond	43	46	49	*0*
				2.5

ΔSSE	Miami	Orlando	Tampa	Wash., Arlin.	Richmond
Miami					
Orlando	**8.5**				
Tampa	85	97.5			
Wash., Arlin.					
Richmond				14.7	

(c)

City	2010	2011	2012	*SSE*
Miami, Orlando	45.5	44	51	*8.5*
Tampa	39	50	41	*0*
Washington, Arlington	47	46	51.5	*2.5*
Richmond	43	46	49	*0*
				11

ΔSSE	Miami, Orlando	Tampa	Wash., Arlin.	Richmond
Miami, Orlando				
Tampa	127.3			
Wash., Arlin.				
Richmond			**14.7**	

(d)

City	2010	2011	2012	*SSE*
Miami. Orlando	45.5	44	51	*8.5*
Tampa	39	50	41	*0*
Virginia	45	46	50.6	*14.7*
				23.2

Figura 2.3: Esempio di applicazione dell'algoritmo greedy fine shrink.

Le istruzioni a riga 5 nell'Algoritmo 3 hanno lo scopo di minimizzare l'SSE. Da un punto di vista implementativo questo può essere realizzato visitando

una matrice simmetrica che memorizza i ΔSSE per ogni coppia di f-slice. Dopo ogni fusione, la matrice viene aggiornata ricalcolando i ΔSSE per tutte le celle coinvolte nella fusione. Per via dei vincoli gerarchici non tutte le celle della matrice sono valorizzate in quanto non tutte le coppie di f-slice possono essere unite rispettando tali vincoli. Per migliorare le performance nella nostra implementazione vengono visitate solamente le celle valorizzate. Per semplicità tali dettagli sono nascosti nell'Algoritmo 3.

La complessità computazionale nel caso peggiore corrisponde al caso in cui non vi sono vincoli gerarchici e il vincolo di dimensione è $size_{max} = 2$. In questo caso la complessità è $O(\delta^3 \cdot |C^v|)$ dove $\delta = |Dom(a_k)|$ e $|C^v| = |Dom(b)| \cdot |Dom(c)| \cdot \ldots$ è la dimensione di ogni slice. Più precisamente, se $size_{max} = 2$ il ciclo while (riga 3 dell'Algoritmo 3) è eseguito $\delta - 2$ volte. Alla i-esima iterazione, la matrice dei ΔSSE viene visitata per trovare il minimo (riga 5) e in seguito aggiornata dopo che la fusione più promettente è stata individuata. La prima operazione richiede $\frac{(\delta-i+1)(\delta-i)}{2}$ confronti, la seconda richiede $|C^v|(\delta - i)$ operazioni.

2.4 Operatore coarse shrink

Se sulla riduzione $Red_{h_1}(C)$ risultante dall'operazione di shrink si impone l'ulteriore vincolo per cui gli f-slice corrispondenti a tutti i valori di a_j—j-esimo attributo dimensionale della gerarchia h_1—il cui roll-up corrisponde ad un singolo valore v^* di a_{j+1} devono essere tutti aggregati assieme oppure tutti separati, allora parliamo di *coarse shrinking*. Di qui in avanti le operazioni di aggregazione che rispettano questo vincolo verranno chiamate anche *aggregazioni coarse*.

L'operazione coarse shrink può essere vista come una variante di fine shrink in cui ogni f-slice può essere descritto come l'unione di tutti gli f-slice corrispondenti ai valori $v_1 \in Dom(a_1)$ che aggregano—roll-up—a un singolo valore $v_k \in Dom(a_k)$ con $k \geq 1$. Alcuni esempi di *riduzioni coarse* sono

Algoritmo 4 L'enumerazione CoarseChildren

Require: Una partizione $P = \{S_1, \ldots, S_t\}$
Require: L'indice k dell'ultimo insieme aggregato in P
Ensure: L'insieme T delle partizioni figlie di P

```
 1: T ← ∅                                                    ▷ Inizializza il risultato
 2: SParent ← Parent(S_k)
 3: SChildren ← {SChild : SChild ∈ Child(SParent)}
 4: if SChildren ⊆ P then                                    ▷ Controlla i vincoli gerarchici
 5:     P_new ← (P \ SChildren) ∪ SParent
 6:     T ← T ∪ P_new
 7: S_j ← LastChild(SParent)
 8: for i ← j + 1 to |P| do
 9:     SParent ← Parent(S_i)
10:     SChildren ← {SChild : SChild ∈ Child(SParent)}
11:     P_new ← (P \ SChildren) ∪ SParent
12:     T ← T ∪ P_new
13:     i ← i + |SChildren|          ▷ Salta al gruppo successivo di insiemi da unire
14: return T
```

mostrate a sinistra in Figura 2.6.

Le definizioni dei problemi di riduzione 1 e 2 presentate precedentemente sono valide anche in questo caso, con la differza che la riduzione risultante deve rispettare l'ulteriore vincolo presentato sopra. Come fatto per fine shrink, presenteremo di seguito due approcci per la soluzione di tali problemi: uno ottimale e uno euristico.

2.4.1 Algoritmo branch-and-bound per coarse shrink

In modo simile al caso precedente abbiamo utilizzato un approccio branch-and-bound con un'enumerazione agglomerativa. L'enumerazione è descritta nell'Algoritmo 4.

In Figura 2.4 è rappresentato un esempio di enumerazione prodotta tramite la procedura CoarseChildren.

Si inizia da una partizione radice $P_0 = \{S_0, \ldots, S_n\}$ in cui ogni valore di a_1 della gerarchia da ridurre h_1 definisce un diverso elemento S_i. Ogni partizione generata con questa enumerazione è hierarchy compliant e soddisfa il vincolo

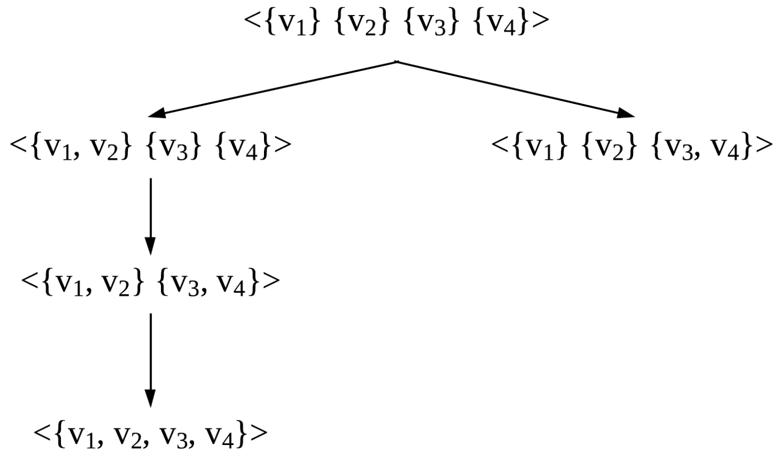


Figura 2.4: Esempio di albero delle enumerazioni coarse generato dalla gerarchia in Figura 2.2.a.

aggiuntivo che caratterizza le riduzioni coarse.

Come nel caso di fine shrink i nodi dell'albero delle enumerazioni vengono visitati seguendo un criterio best-first in cui la partizione successiva è scelta come quella con minor SSE tra quelle con numero numero di slice minimo. Nonostante lo spazio di ricerca sia più ristretto rispetto a fine shrink, è comunque abbastanza grande da rendere proibitivo criteri di ricerca più aggressivi.

L'Algoritmo 5 descrive l'approccio ottimale per il problema SSE-bound.

Algoritmo 5 Algoritmo ottimale Coarse Shrink—SSE-bound

Require: P_0
Require: α_{SSE}
Ensure: P_{Opt}
1: $Stack.Push(P_0)$ ▷ Inizializza lo stack
2: $P_{Opt} \leftarrow \emptyset$ ▷ Inizializza la soluzione
3: **while** $P \leftarrow StackPop()$ **do**
4: $T = CoarseChildren(P)$
5: **for all** $P_{Child} \in T$ **do**
6: **if** $SSE(P_{Child}) > \alpha_{SSE}$ **then**
7: break; ▷ P_{Child} and its sub-tree are pruned
8: **else**
9: **if** $|P_{Child}| < |P_{Opt}|$ **then**
10: $P_{Opt} \leftarrow P_{Child}$ ▷ Nuova soluzione
11: $R \cup = P_{Child}$
12: $Stack.Push(BestOrder(R))$ ▷ Applica localmente un criterio best-first e aggiunge le partizioni allo stack
13: **return** P_{Opt}

2.4.2 Algoritmo greedy per coarse shrink

Algoritmo 6 Algoritmo greedy Coarse Shrink—size-bound

Require: C al group-by $G = \langle a_k, b, \ldots \rangle$, $size_{max}$
Ensure: $Red_{h_1}(C)$
1: $Red_{h_1}(C) \leftarrow \emptyset$ ▷ Inizializza la riduzione...
2: **for all** $v \in Dom(a_k)$ **do**
3: $Red_{h_1}(C) \leftarrow Red_{h_1}(C) \cup \{C^{\{v\}}\}$ ▷ ...uno slice per ogni f-slice
4: **while** $size(Red_{h_1}(C)) > size_{max}$ **do** ▷ Controlla il vincolo di dimensione massima
5: find $F^{V_i}, \ldots, F^{V_k} \in Red_{h_1}(C)$ s.t. $\{F^{V_i}, \ldots, F^{V_k}\} = Desc_j(\overline{v})$ with $\overline{v} \in Dom(a_{j+1}), i \leq k$
6: **and** $\frac{\Delta SSE}{|Desc_j(\overline{v})|}$ with $\Delta SSE = SSE(F^{\overline{v}}) - \sum_{j=i}^{k} SSE(F^{V_j})$ is minimal
7: $Red_{h_1}(C) \leftarrow Red_{h_1}(C) \setminus \{F^{V_i}, \ldots, F^{V_k}\}$ ▷ Aggrega in un f-slice
8: $Red_{h_1}(C) \leftarrow Red_{h_1}(C) \cup \{F^{\{\overline{v}\}}\}$
9: **return** $Red_{h_1}(C)$

L'Algoritmo 6 descrive la procedura greedy agglomerativa per coarse shrink. Il procedimento è molto simile al caso fine con la differenza che in questo caso vengono calcolate solamente le aggregazioni coarse che sono in numero notevolmente ridotto rispetto alle aggregazioni in fine srhink. In questo caso applicando l'algoritmo su una gerarchia h_1 con $\alpha_{size} = 1$ vengono eseguite esattamente n iterazioni e calcolate esattamente n f-slice—e relativi ΔSSE—con $n = NumeroNodi(h_1) - NumeroFoglie(h_1)$.

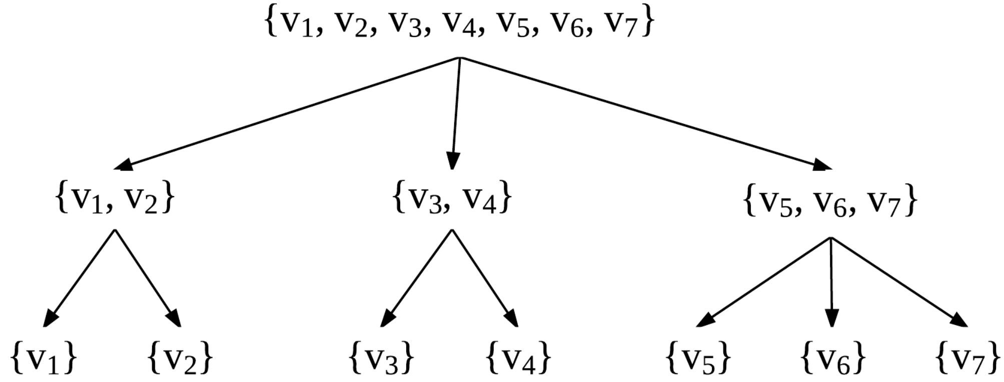


Figura 2.5: Esempio di gerarchia.

Nell'algoritmo greedy fine shrink il criterio di scelta della coppia di f-slice da unire era dettata solamente dal loro ΔSSE. Come si può notare a riga 6 dell'Algoritmo 6 il criterio qui consiste nel selezionare l'aggregazione con il minor rapporto tra ΔSSE e numero di f-slice aggregati. Mentre in fine shrink ogni aggregazione fonde esattamente due f-slice, in coarse shrink un'aggregazione può fonderne un numero diverso. Per questo è importante pesare l'incremento di SSE in relazione alla riduzione nel numero di f-slice.

Esempio 7 Per mostrare l'importanza di quanto detto riguardo al criterio di scelta della migliore aggregazione coarse prendiamo in considerazione la gerarchia mostrata in Figura 2.5. Fissiamo il parametro $\alpha_{size} = 5$ e i seguenti ΔSSE: $\Delta SSE(\{v_1, v_2\}) = \Delta SSE(\{v_3, v_4\}) = 15$, $\Delta SSE(\{v_5, v_6, v_7\}) = 20$.

Seguendo il criterio di scelta della prossima aggregazione migliore definito nell'Algoritmo 3—ovvero $MIN(\Delta SSE)$—l'algoritmo esegue due iterazioni. Nella prima iterazione viene aggregato $\{v_1, v_2\}$ con $\Delta SSE = 15$, nella seconda $\{v_3, v_4\}$ anch'esso con $\Delta SSE = 15$. Abbiamo soddisfatto il vincolo di dimensione e l'SSE totale risultante è pari a $15 + 15 = 30$.

Seguendo invece il criterio definito per l'Algoritmo 6 l'aggregazione più pro-

mettente alla prima iterazione è $\{v_5, v_6, v_7\}$ con un rapporto $\frac{\Delta SSE}{\#slice\ aggregati} = \frac{20}{3} = 6.7$. Dopo la prima iterazione il vincolo $\alpha_{size} = 5$ è soddisfatto e l'SSE totale è pari a 20.

Esempio 8 Per un esempio passo a passo dell'algoritmo consideriamo il cubo descritto in Figura 1.2 e il limite $size_{max} = 1$ (Figura 2.6). Il limite in questo caso viene fissato al numero minimo di slice raggiungibile allo scopo di mostrare più iterazioni dell'algoritmo possibile.

1. Inizialmente abbiamo sei f-slice e il vincolo $size_{max}$ è quindi violato. Sono possibili solo due aggregazioni coarse, Florida e Virginia. Entrambe aggregano tre slice e la più promettente è VA.

2. Abbiamo ora quattro slice e l'unica aggregazione possibile è quella relativa alla Florida con ΔSSE pari a 135.8.

3. A questo punto un vincolo gerarchico si rilassa ed è possibile aggregare le città della Florida assieme a quelle della Virginia per creare l'f-slice South-Atlantic. Il vincolo $size_{max}$ è soddisfatto e l'algoritmo termina.

	2010	2011	2012	*SSE*
Miami	47	45	50	0
Orlando	44	43	52	0
Tampa	39	50	41	0
Washington	47	45	51	0
Richmond	43	46	49	0
Arlington		47	52	0

	Δ SSE
FL	135.8
VA	14.7

	2010	2011	2012	*SSE*
Miami	47	45	50	0
Orlando	44	43	52	0
Tampa	39	50	41	0
VA	45	46	50.6	14.7

	Δ SSE
FL	135.8

	2010	2011	2012	*SSE*
FL	43.3	46	47.6	135.8
VA	45	46	50.6	14.7

	Δ SSE
South-Atlantic	22.3

	2010	2011	2012	*SSE*
South-Atlantic	44	46	49.2	172.8

Figura 2.6: Esempio di applicazione dell'algoritmo greedy coarse shrink.

Capitolo 3

Implementazione degli algoritmi di shrinking

Questo capitolo presenta l'applicazione realizzata per testare e valutare gli operatori di shrinking. Inizialmente, in 3.1 viene data una panoramica dell'architettura dei suoi componenti principali. Successivamente in 3.3 vengono descritte quelle che secondo noi sono le strutture dati più interessanti implementate. Quest'ultima parte in particolare fornisce informazioni utili per meglio comprendere le valutazioni sulle performance fatte nel capitolo 4.

3.1 Ambiente di sviluppo e architettura

L'applicazione è stata sviluppata interamente in *C++* utilizzando *Visual Studio 2010* su sistema operativo *Windows 7 (64 bit)*. Il motivo per cui si è scelto C++ e non un linguaggio di livello più alto—e quindi generalmente più produttivo—è per poter avere migliori prestazioni sia a livello di tempo di esecuzione che di utilizzo di memoria.

Per quanto riguarda il DBMS si è scelto invece *MySQL*. In questo caso non era necessario avere prestazioni eccellenti e MySQL rappresenta un buon

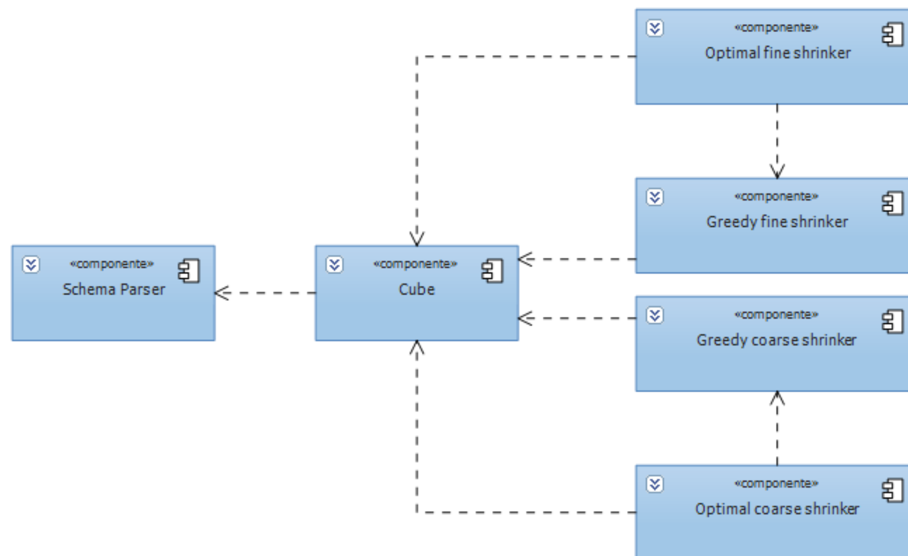


Figura 3.1: Diagramma dei componenti dell'applicazione (ad alto livello di astrazione).

compromesso tra semplicità di utilizzo ed efficienza. La versione utilizzata è la 5.1. Va sottolineato che l'accesso al DBMS è effettuato tramite ODBC quindi seppur per i test e per lo sviluppo si sia scelto di utilizzare MySQL, è possibile connettersi anche altri DBMS.

Per arrivare a scrivere le operazioni di shrinking—che ricordo essere il focus del progetto—sono necessarie altre strutture dati e routine non banali come ad esempio una struttura per modellare un ipercubo e le routine per popolare tale struttura con dati provenienti da fonti esterne. Il primo passo è stato quindi quello di cercare un framework con le feature appena descritte, l'unico degno di nota è *Pentaho Mondrian* [mon]. Purtroppo però le API di Mondrian non permettono di gestire la struttura del cubo con la libertà a noi necessaria. Tra modificare Mondrian per adattarlo alle nostre esigenze e scrivere da zero gli elementi necessari al progetto si è scelta la seconda opzione.

Come si può notare da Figura 3.1, l'applicazione può essere scomposta nei

seguenti componenti principali:

- *Schema parser*: il modello logico dei dati viene rappresentato tramite un file schema Mondrian—vedi 3.2.1. Il parser ha lo scopo di leggere ed estrarre le informazioni necessarie all'applicazione per poter effettuare interrogazioni "sensate" al database. I costrutti di uno schema sono molto numerosi, per cui il parser scritto ne comprende solo un sottoinsieme.

- *Cube*: in questo componente rientrano tutte le strutture dati necessarie a modellare un ipercubo e le routine per popolarlo e gestirlo.

- *Greedy fine shrinker* e *optimal fine shrinker*: implementazione dell'algoritmo di fine shrinking, versione greedy e ottimale.

- *Greedy coarse shrinker* e *optimal coarse shrinker*: implementazione dell'algoritmo di coarse shrinking, versione greedy e ottimale.

Per maggiori dettagli sulle strutture dati nei vari componenti appena descritti fare riferimento alla sezione 3.3.

3.2 Input e output

L'applicazione prima di poter eseguire un'operazione di shrinking ha bisogno dei seguenti input:

- Una *connessione ODBC* al database contenente i dati. Grazie ad ODBC è possibile interfacciare l'applicazione con diversi DBMS senza dover modificare l'implementazione.

- Un file *schema* che definisca la modellazione logica.

- Una *selezione dei dati*, come ad esempio "le vendite per regione dell'anno 2012".

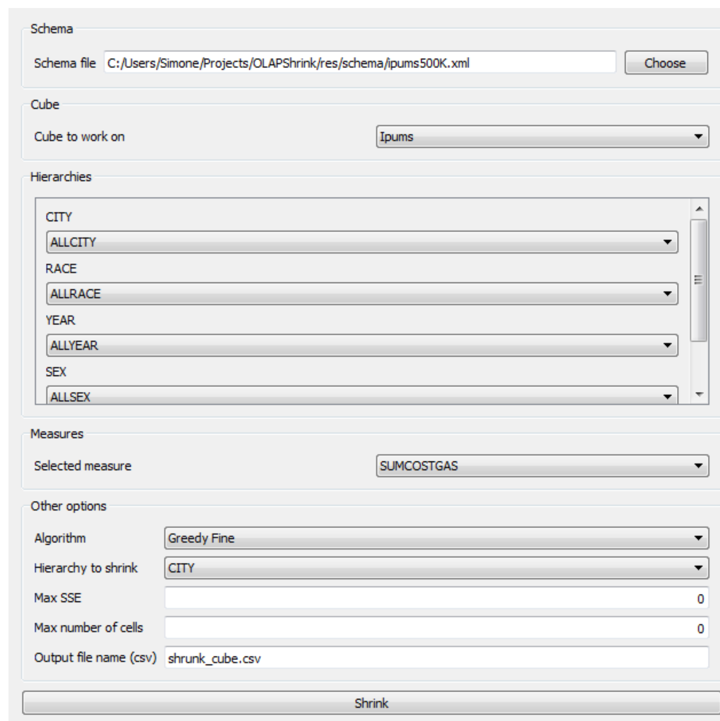


Figura 3.2: GUI dell'applicazione.

- *Parametri di shrinking* per stabilire i limiti di riduzione.

Come si può vedere in Figura 3.2 la GUI dell'applicazione è molto semplice, infatti lo scopo non è quello di fornire uno strumento software da *produzione* ma semplicemente per poter meglio testare e valutare l'efficacia delle idee studiate.

I dati nel database sono memorizzati con il classico schema a stella (o snowflake), per poterli estrarre e modellare in un ipercubo è quindi necessario avere a disposizione altre informazioni—meta-dati—che definiscano il modello multidimensionale. A questo scopo si è deciso di utilizzare *Mondrian Schema* che viene descritto brevemente di seguito.

3.2.1 Mondrian schema file

Uno schema definisce un modello multidimensionale in relazione ai dati contenuti in un database. Il modello logico, costituito da cubi, gerarchie e misure, viene mappato su un modello fisico. Quest'ultimo è quindi la sorgente dei dati a cui si accede tramite il modello logico definito dallo schema.

La descrizione di uno schema è realizzata con un file XML e in questo file vengono definiti i vari componenti del modello. Lo schema di Mondrian è molto flessibile e supporta diversi costrutti, per semplicità però si riporta di seguito solamente una piccola panoramica degli elementi essenziali.

Codice 3.1: Esempio di un semplice file schema Mondrian.

```
1 <?xml version="1.0"?>
2
3 <Schema name="Ipums">
4   <Cube name="Ipums">
5     <Table name="FACT500K"/>
6
7     <Dimension name="CITY" foreignKey="CITY">
8       <Hierarchy hasAll="true" primaryKey="IDCITY" allLevelName=
            "ALLCITY"
9                   allMemberName="All">
10        <Table name="CITY"/>
11        <Level name="REGION" column="REGION" type="String"
             uniqueMembers="true"/>
12        <Level name="STATE" column="STATE" type="String"
             uniqueMembers="true"/>
13        <Level name="CITY" column="CITY" type="String"
             uniqueMembers="false"/>
14      </Hierarchy>
15    </Dimension>
16
17    <Dimension name="YEAR" foreignKey="YEAR">
18      <Hierarchy hasAll="true" primaryKey="IDYEAR" allLevelName=
            "ALLYEAR"
19                  allMemberName="All">
```

```
20          <Table name="YEAR"/>
21          <Level name="YEAR" column="YEAR" type="String"
                uniqueMembers="true"/>
22        </Hierarchy>
23      </Dimension>
24
25      <Measure name="SUMCOSTGAS" column="COSTGAS" aggregator="
            sum" formatString="#,###.##"/>
26      <Measure name="EVENTCOUNT" aggregator="count" formatString=
            "#,###">
27          <MeasureExpression>
28              <SQL dialect="generic"> * </SQL>
29          </MeasureExpression>
30      </Measure>
31    </Cube>
32  </Schema>
```

- *Table*: è l'elemento principale per effettuare il collegamento tra modello logico e fisico, tramite questo elemento è ad esempio possibile definire quale tabella fisica corrisponde alla *fact table* di un cubo.

- *Cube*: definisce un ipercubo fornendo un contenitore in cui inserire i vari elementi costituenti, ovvero dimensioni, gerarchie e misure etc.

- L'elemento *dimension* indica invece una collezione di gerarchie che al livello fisico fanno riferimento alla stessa chiave esterna nella fact table.

- *Hierarchy*: è un insieme di membri organizzati in una struttura di analisi. Nel modello fisico fa riferimento ad una determinata dimension table.

- *Level*: rappresenta un particolare attributo gerarchico e sul modello fisico fa riferimento ad una colonna di un *dimension table*.

- Tramite l'elemento *measure* vengono definiti i valori delle celle dell'ipercubo. È molto flessibile ed possibile creare misure calcolate e for-

	A	B	C	D	E	F	G
1		2000	2001	2002	2003	2004	2005
2	MI, IL	397107	463919,5	474397	728036,5	737738	585209,5
3	OH	524296	636494	632853	1004648	1015670	781069
4	WI, IN	243223,5	275609,5	274450,5	440983	448715,5	349784,5
5	East south central	96472,25	116011,75	114804	185429,5	185847,75	155167,5
6	NJ	372730	456315	439130	724795	724248	602181
7	NY	799602	980132	995186	1482771	1466965	1117641
8	PA	599611	702538	714220	1141140	1136414	912899
9	CO, AZ	82448	101280,5	111079,5	163534	173591,5	133858
10	WY, NM, MT, ID, UT, NV	26280,666667	30802,5	31387,5	48737,833333	50355,333333	37859,333333
11	ND	8201	8277	6025	14592	13851	13735
12	CT	223439	269314	258769	419313	445668	334977
13	MA	585734	696353	692266	1103012	1123874	881999
14	RI, NH, ME	67961,666667	77412	74217,666667	134349,66667	132446,33333	111777,33333
15	VT, RI (1850)	6774	7334,5	6248	11554,5	12681	11876
16	CA	1233331	1483303	1510555	2293541	2267946	1813940
17	OR, HI, AK	32420	42975,333333	42427,333333	68580	63972,333333	50282
18	WA	190173	212547	210934	360900	340617	271405
19	NC, GA, MD	122521,66667	158571	156540	241919,33333	244373,33333	191818
20	VA, FL	222952	270958,5	287483,5	414268	431826,5	322638
21	WV, PR, DE, SC, DC	51249,5	57928,25	59130	92501,75	90156,5	72465,5
22	MN, IA	132986,5	151362	156433	248737	249307,5	199743
23	MO	229477	271548	271993	416981	440905	319840
24	SD, NE, KS	42444,666667	48019	48113,333333	78128,333333	76071,333333	70948,666667
25	OK, LA, AR	101396,66667	125324,33333	124304	189576,66667	187981	151361
26	TX	481865	581837	606535	921395	925242	686615

Figura 3.3: Risultato di un'operazione di fine shrinking su una relazione *state*, *year* di 53 slice (visualizzato tramite OpenOffice).

mattate in un certo schema. Nel modello fisico fa riferimento ad una o più colonne della fact table.

Per un esempio di file XML fare riferimento al codice 3.1.

3.2.2 Output

Sempre seguendo una filosofia di praticità, l'output dell'applicazione è un file .CSV—*Comma-Separated Values*—che può essere aperto e visualizzato ad esempio con Microsoft Excel.

Il file .CSV prodotto è organizzato come una tabella in cui le righe rappresentano gli slice aggregati. L'intestazione delle righe saranno le label (liste di label in caso di fine shrinking) della gerarchia su cui è stata applicata l'operazione di shrinking, mentre l'intestazione delle colonne sarà costituita dalle

	A	B	C	D	E	F	G
1		2000	2001	2002	2003	2004	2005
2	East north central	360991,4	423110,4	426109,6	668537,4	677715,4	530211,4
3	East south central	96472,25	116011,75	114804	185429,5	185847,75	155167,5
4	NJ	372730	456315	439130	724795	724248	602181
5	NY	799602	980132	995186	1482771	1466965	1117641
6	PA	599611	702538	714220	1141140	1136414	912899
7	Mountain	40322,5	48422	51310,5	77436,875	81164,375	61859
8	ND	8201	8277	6025	14592	13851	13735
9	CT	223439	269314	258769	419313	445668	334977
10	MA	585734	696353	692266	1103012	1123874	881999
11	ME	52909	55856	60765	105888	95390	88029
12	NH	53452	61176	51917	90190	101602	83653
13	RI	97524	115204	109971	206971	200347	163650
14	RI (1850)	1553	375	1878	1955	786	1959
15	VT	11995	14294	10618	21154	24576	21793
16	AK	18727	32337	26883	43296	41309	33241
17	CA	1233331	1483303	1510555	2293541	2267946	1813940
18	HI	18644	25928	23506	38270	38106	27261
19	OR	59889	70661	76893	124174	112502	90344
20	WA	190173	212547	210934	360900	340617	271405
21	South Atlantic	113163	138815,88889	142345,22222	213811,22222	217488,77778	167843,55556
22	West north central	103797,33333	119721,5	121533,16667	191473,33333	194622,33333	155362
23	AR	42093	51076	55830	79907	88544	72018
24	LA	135367	171914	165058	266059	255526	194408
25	OK	126730	152983	152024	222764	219873	187657
26	TX	481865	581837	606535	921395	925242	686615

Figura 3.4: Risultato di un'operazione di coarse shrinking su una relazione *state*, *year* di 53 slice (visualizzato tramite OpenOffice).

label delle altre gerarchie. Per qualche esempio di output fare riferimento alle Figure 3.3, 3.4.

3.3 Strutture dati

In questa sezione vengono descritte le strutture dati principali dell'applicazione. Da notare che per chiarezza i nomi usati in questa sede non corrispondono necessariamente ai nomi usati nel codice in quanto questi ultimi risulterebbero poco leggibili.

3.3.1 Ipercubo e slice

La prima struttura dati descritta è l'ipercubo che rappresenta i dati selezionati dall'utente ed estratti dal database e, dopo varie trasformazioni, l'output

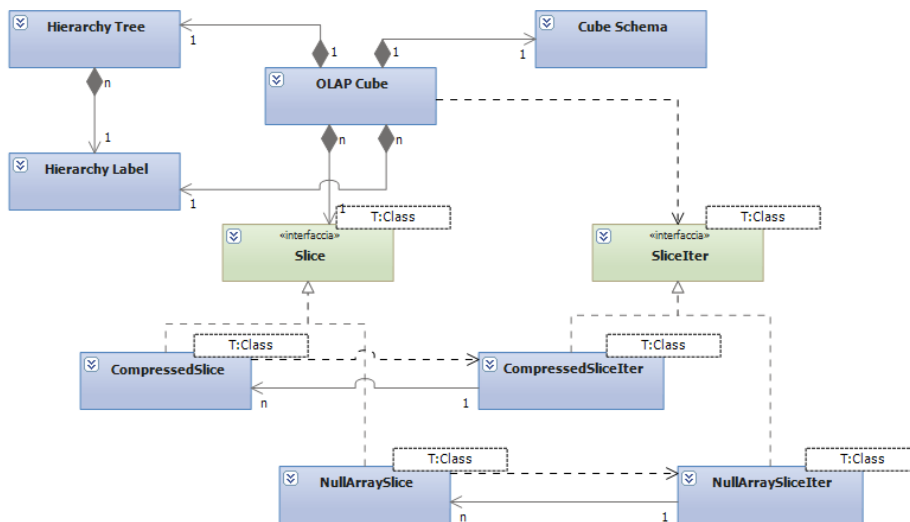


Figura 3.5: Diagramma delle classi per modellare un ipercubo.

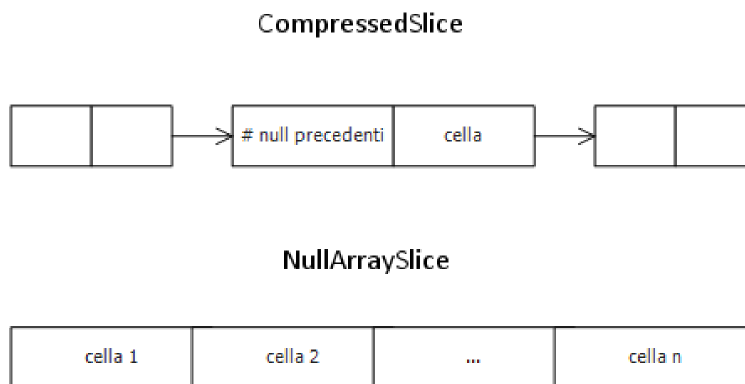


Figura 3.6: Rappresentazione di uno slice come lista e come array.

finale.

Nel nostro studio le operazioni di shrinking vengono applicate ad una sola gerarchia, per semplicità quindi la struttura del cubo è modellata come

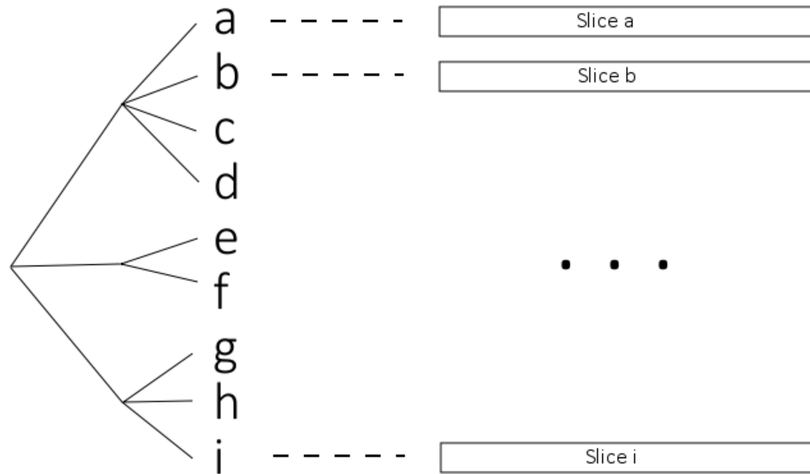


Figura 3.7: Visualizzazione della struttura di un cubo.

una gerarchia alle cui foglie sono associati degli slice. Sono inoltre memorizzate altre informazioni come la descrizione dello schema multidimensionale e le label delle altre gerarchie. Per una rappresentazione più formale fare riferimento al diagramma in Figura 3.5.

Il popolamento di un cubo avviene in più fasi. Per prima viene popolata la gerarchia su cui applicare l'operazione di shrinking. In seguito vengono estratte solo le label—non la struttura completa—delle gerarchie secondarie. Per ultimo si popolano gli slice con le misure contenute nella fact table.

Come è facilmente intuibile uno degli elementi critici per ottenere buone prestazioni è la struttura utilizzata per modellare uno slice. Uno dei fattori da considerare è quello della *sparsità* dei dati. I valori delle celle del cubo possono essere assenti—ovvero *NULL*—anche in gran numero. Purtroppo non è sempre possibile sapere a priori le proporzioni di questo fenomeno, in generale quindi uno slice è modellato come una lista in cui vengono memorizzati

solamente gli elementi non nulli. Nel caso in cui il cubo è molto denso può essere più performante, sia a livello di memoria che di velocità, una struttura basata su array in cui vengono memorizzati sia elementi nulli che non.

3.3.2 Matrici delle distanze nelle versioni greedy

Inizialmente, in quella che chiameremo versione *naive* dell'algoritmo fine greedy, per memorizzare le distanze dei vari slice veniva utilizzata una mezza matrice di dimensione $n^2/2$, con n numero di slice. Ad ogni passo dell'algoritmo, viene scelta la coppia di slice con minor distanza—ΔSSE. Per trovare tale minimo nella versione naive si scorre *tutta* la mezza matrice. Grazie ai vincoli gerarchici però non tutte le coppie di slice sono aggregabili e di conseguenza solo una parte della matrice sarà valorizzata. È evidente quindi che l'approccio naive presenta notevoli sprechi nella fase di ricerca della distanza minima.

Nella successiva—e attuale—versione per evitare di leggere $n^2/2$ distanze si creano tante piccole mezze matrici *piene*. La struttura utilizzata per implementarle è rappresentata in Figura 3.9 ed è sostanzialmente un reticolato di liste dal quale si possono togliere o aggiungere nuovi livelli per adattarlo al numero di slice da comparare. In questo modo si leggono solamente le distanze relativi a coppie di slice che è possibile aggregare senza violare i vincoli gerarchici.

Una possibile alternativa all'utilizzo delle strutture appena descritte è quella di usare strutture ordinate in base alla distanza in modo da ridurre il tempo di ricerca del minimo. Ad ogni aggregazione è però necessario aggiornare distanze precedentemente calcolate e ciò richiederebbe di ricercare tali distanze all'interno della struttura oppure di mantenere informazioni aggiuntive per un accesso rapido. Si è ritenuto quindi che non valesse la pena utilizzare strutture più complicate poiché i vantaggi che introdurrebbero sarebbero controbilanciati dagli svantaggi.

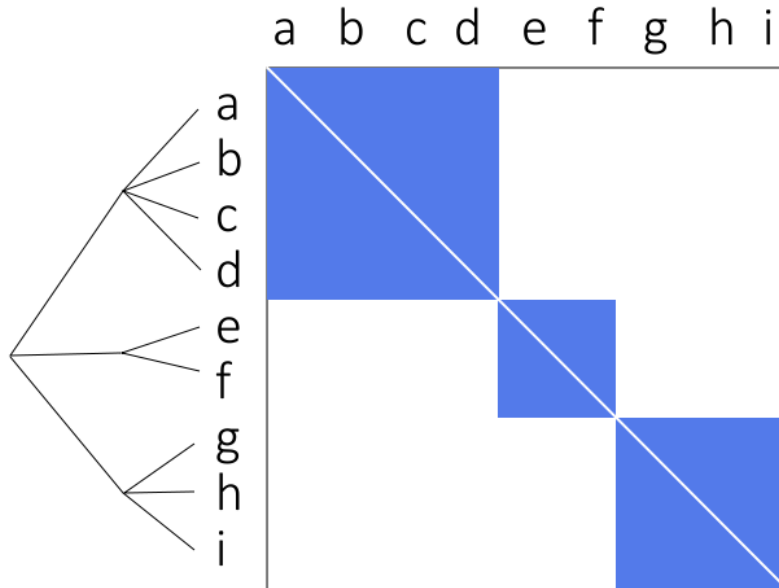


Figura 3.8: Matrice completa usata nella versione naive dell'algoritmo greedy per il problema fine shrink. Le parti blu rappresentano le distanze valorizzate, ovvero relative alle aggregazioni possibili.

Nella versione coarse dell'operatore shrink la problematica appena descritta cambia in modo significativo in quanto dopo un'aggregazione di un gruppo di slice non vi è mai bisogno di aggiornare distanze già calcolate ma solamente di inserirne di nuove. Grazie a questa peculiarità si è scelto di modellare la matrice delle distanza con una coda a priorità.

3.3.3 Ipercubo con slice condivisi

Nell'algoritmo ottimale per il problema fine shrink viene utilizzato un approccio branch-and-bound in cui ogni nodo dell'albero di branching rappresenta un possibile partizionamento degli slice. Ogni partizionamento è di fatto un

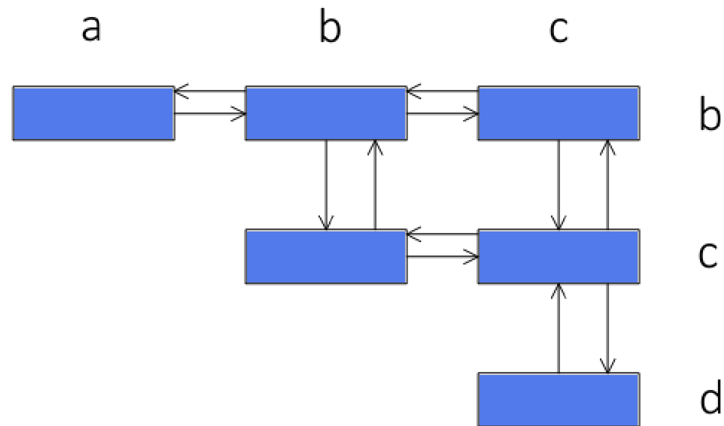


Figura 3.9: Struttura di una sub-matrice nell'algoritmo greedy per il problema fine shrink.

ipercubo di uno slice più piccolo rispetto all'ipercubo della partizione padre. Data la natura combinatoria del problema vengono generati numerosi partizionamenti—e quindi cubi—che devono essere memorizzati e calcolati.

È evidente che è necessario un modo furbo per gestire la memorizzazione dei vari partizionamento. Partiamo dalle seguenti considerazioni:

1. La parte più "costosa" di un cubo sono i suoi slice.

2. Tra una partizione padre e una partizione figlia, solo pochi slice cambiano—generalmente ne vengono fusi solo due alla volta.

3. Memorizzare solo la struttura della partizione senza gli slice, per poi ricalcolare da capo questi ultimi ogni volta, risulta in tempi di computazione troppo elevati.

In base a tali considerazioni la struttura descritta precedentemente per modellare un ipercubo viene modificata per permettere la condivisione degli slice con un sistema che ricorda la gestione della memoria nei *garbage collector* di linguaggi come *C#* e *Java*. L'idea è quella di sostituire i puntatori agli slice nelle foglie della gerarchia con puntatori più furbi che rilascino automaticamente la memoria dello slice solo quando nessun altro puntatore vi punta. Il risultato è che dato un cubo padre è possibile generarne i figli copiando solamente l'albero della gerarchia e calcolando il nuovo slice aggregato.

3.3.4 Albero delle aggregazioni e partizioni coarse

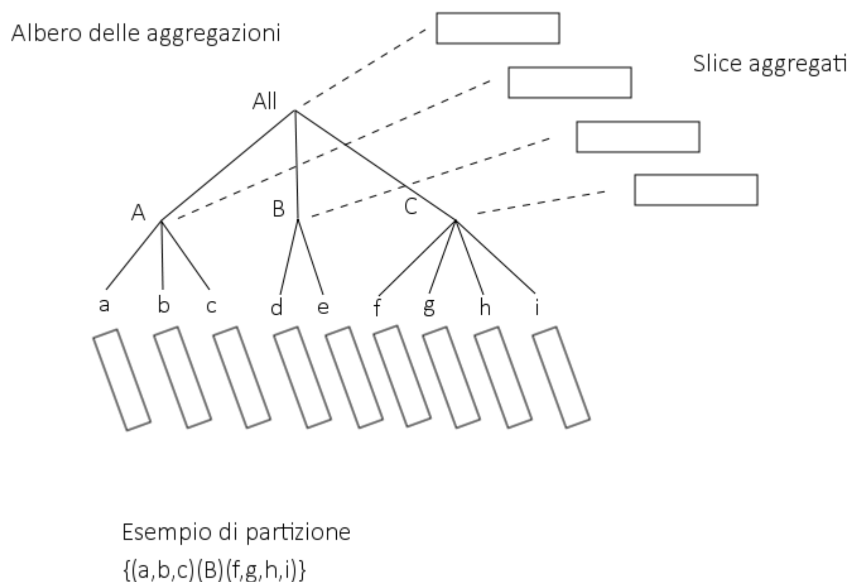


Figura 3.10: Rappresentazione dell'albero delle aggregazioni e delle partizioni coarse.

Una problematica analoga a quella appena descritta si presenta nel caso

dell'algoritmo ottimale per il problema coarse shrink. In questo caso valgono le assunzioni fatte per il caso fine più le seguenti:

1. I vincoli imposti alle aggregazioni sono ancora più ristretti rispetto al caso fine: gli slice corrispondenti ad un certo nodo della gerarchia devono essere aggregati *tutti* assieme oppure *nessuno*. Da questo è possibile notare che il numero di slice differenti che è possibile creare è pari al numero di nodi della gerarchia da comprimere.

2. La natura del problema è ancora combinatoria ma in questo caso ad ogni sottoinsieme di slice aggregati in una partizione corrisponde un nodo nella gerarchia.

In base a (1) è possibile pre-calcolare tutti gli slice con i relativi incrementi di SSE eliminando la necessità di calcolarli e memorizzarli per ogni nuova partizione. Grazie a (2) per descrivere una partizione basta una semplice lista di riferimenti ai nodi della gerarchia. Tale lista è lunga al massimo quanto il numero di foglie della gerarchia.

Capitolo 4

Valutazione delle performance degli operatori di shrinking

In questo capitolo vengono valutate la qualità e le performance dell'implementazione degli operatori di shrinking presentati in questo studio. Nella prima parte (4.1) viene presentata l'impostazione dei test e i relativi dataset utilizzati. In seguito, in 4.2 e 4.3.1, viene valutata la qualità delle approssimazioni e le performance delle due diverse versioni di shrink—fine e coarse. Per concludere viene data una sintesi delle varie valutazioni precedenti.

4.1 Setup dei test

Per i test sono stati utilizzati dati tratti da due dataset differenti: *foodmart* [mon] e *census* da *Integrated Public Use Microdata Series* [Min08]. I rispettivi estratti sono descritti nei DFM in Figura 4.2 e 4.1. *Sales* contiene circa 250.000 fatti non nulli mentre *census* ne contiene 500.000.

I cubi selezionati sono quattro—due per ogni dataset—e sono descritti in Tabella 4.1. Per i dati relativi a dimensioni e densità fare riferimento alla Tabella 4.2. Le gerarchie selezionate per essere ridotte sono *CITY* per census

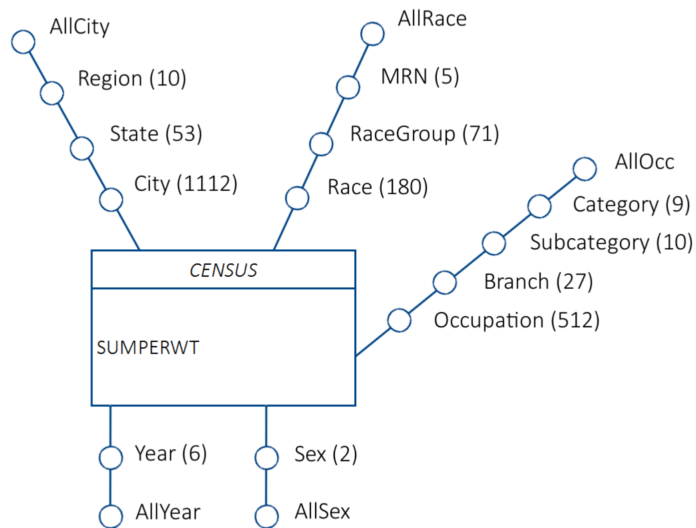


Figura 4.1: DFM del dataset *IPUMS census*.

e *PRODUCT* per sales.

Nome	Descrizione
Census C1	CENSUS[City, Race, Branch, Year].SUMPERWT
Census C2	CENSUS[City, MRN, Category].SUMPERWT
Sales C1	SALES[P. Name, S. Name, Quarter, C. City].UnitSales
Sales C2	SALES[P. Name, S. Country, Year, C. Country].UnitSales

Tabella 4.1: Cubi selezionati per i test.

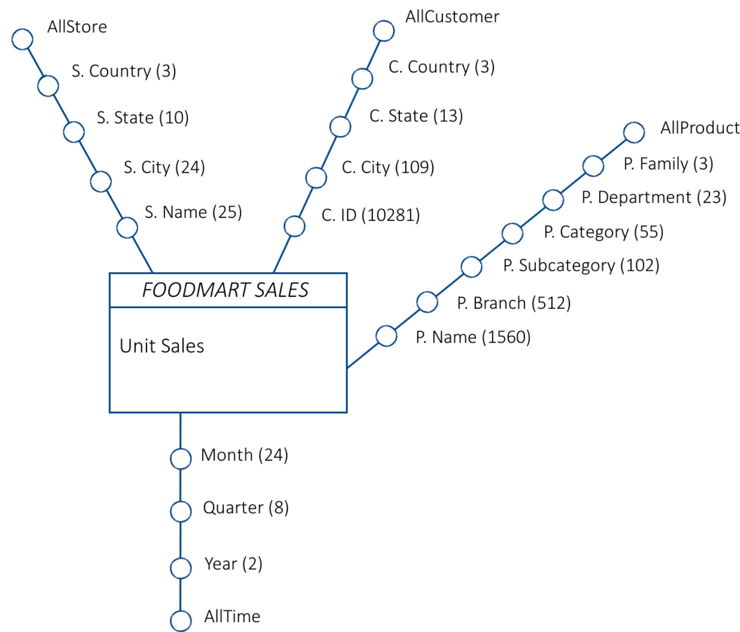


Figura 4.2: DFM del dataset *foodmart sales*.

Cubo	Numero fatti	Numero fatti non nulli	Densità (%)
Census C1	32425920	245387	0,75
Census C2	50040	12097	24,17
Sales C1	34008000	198980	0,58
Sales C2	28080	6236	22,20

Tabella 4.2: Dimensioni e densità dei cubi.

La macchina su cui sono stati effettuati i test è un PC desktop con sistema operativo Windows 7 (64 bit), processore Intel© Core™i5 2.67GHz (quad-core), 4 GB di RAM.

Il resto del capitolo è organizzato come segue. In 4.2 viene analizzata la qualità dei risultati degli algoritmi di shrinking valutando l'errore di approssimazione commesso. In 4.3.1 si valutano invece le performance, ovvero tempi e risorse computazionali. In 4.4 si traggono le conclusioni sulle valutazioni

fatte precedentemente.

4.2 Errore di approssimazione

Come descritto precedentemente la misura di errore utilizzata è l'SSE che è però dipendente dai dati. Ad esempio cubi con valori di misure più elevati tenderanno ad avere un SSE più alto di cubi con valori di misure bassi. Per chiarezza si è scelto di visualizzare l'SSE normalizzato (equazione 4.1) in un range da 0 a SSE_{max} dove quest'ultimo rappresenta l'errore che si commette riducendo un cubo ad un unico slice—massimo grado di riduzione considerando che applichiamo shrink ad una sola gerarchia. Analogamente il grado di riduzione del cubo viene rappresentato tramite dimensione percentuale.

L'SSE% è calcolato come

$$SSE\% = \frac{SSE(Red_{h_1}(C))}{SSE_{max}(C)} \tag{4.1}$$

dove $Red_{h_1}(C)$ rappresenta una riduzione del cubo C e $SSE_{max}(C)$ rappresenta l'SSE della riduzione del cubo C costituita da un singolo f-slice.

In Figure 4.3, 4.4, 4.5 e 4.6 viene mostrato l'andamento dell'SSE all'aumentare del grado di riduzione—o diminuire della dimensione della riduzione. Nei grafici sono visualizzati sia la versione fine che quella coarse, al momento però ci concentreremo solo sulla prima.

Per prima cosa notiamo che i risultati sono nettamente migliori nei cubi census rispetto a quelli sales. È possibile giustificare questa differenza notando che la gerarchia di sales su cui sono applicati gli operatori di shrink impone vincoli gerarchici molto più *stretti* rispetto a quella di census. Infatti, facendo riferimento alle Figure 4.1 e 4.2, gli attributi di CITY hanno le seguenti cardinalità: 1112, 53, 10. Gli attributi di PRODUCT hanno cardinalità: 1560, 512, 102, 55, 23, 3. Genericamente possiamo vedere il rapporto tra le cardinalità dei suoi attributi come indice di quanto una gerarchia sia vincolante.

Figura 4.3: SSE degli algoritmi greedy fine e coarse per il cubo C1 del dataset census.

In Figura 4.7 possiamo vedere le variazioni di densità di tutti i cubi al variare del grado di riduzione. Inizialmente i cubi—considerati a gruppi di due omogenei per dimensione—hanno densità simile ma poi proseguendo con la riduzione i cubi census tendono ad "addensarsi" considerevolmente; i cubi sales invece restano molto sparsi. Dato che nel calcolo dell'SSE aggregare valori nulli non porta errore, è naturale che gli operatori di shrinking tendano ad aggregarli per primi diminuendo così la sparsità. Se durante l'operazione di riduzione la densità non varia—come nel caso di sales—allora significa che non si sta traendo vantaggio dalla presenza dei valori nulli e l'SSE potrebbe risentirne.

Altro fattore che incide sulla qualità dei risultati è la dimensione degli slice. Nel nostro caso si può notare come i cubi C2—caratterizzati da slice più piccoli—ottengano risultati migliori rispetto ai cubi C1. Questo comportamento può essere ricondotto al fenomeno della *curse of dimensionality*.

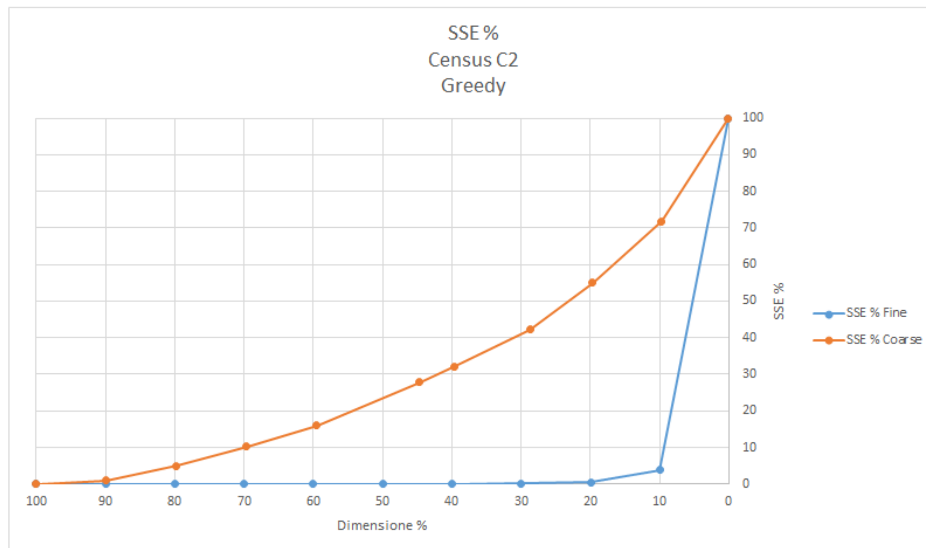


Figura 4.4: SSE degli algoritmi greedy fine e coarse per il cubo C2 del dataset census.

4.2.1 Fine e coarse a confronto

Passiamo ora a valutare le differenze di errore tra la versione fine e quella coarse (Figure 4.3, 4.4, 4.5, 4.6). Come ci si potrebbe aspettare, in tutti i casi fine shrink ottiene risultati migliori di coarse. In particolare la differenza è molto maggiore nei cubi census (Figure 4.3, 4.4) rispetto a quelli sales (Figure 4.5, 4.6) per le differenze nelle gerarchie di cui si è parlato sopra, infatti con gerarchie più vincolanti il divario tra i due approcci tende ad essere minore.

Si ricorda che con coarse shrink si perde precisione nei risultati ma si guadagna in chiarezza di presentazione e risorse computazionali richieste, quindi la scelta di uno o l'altro dipende in gran parte dal caso d'uso.

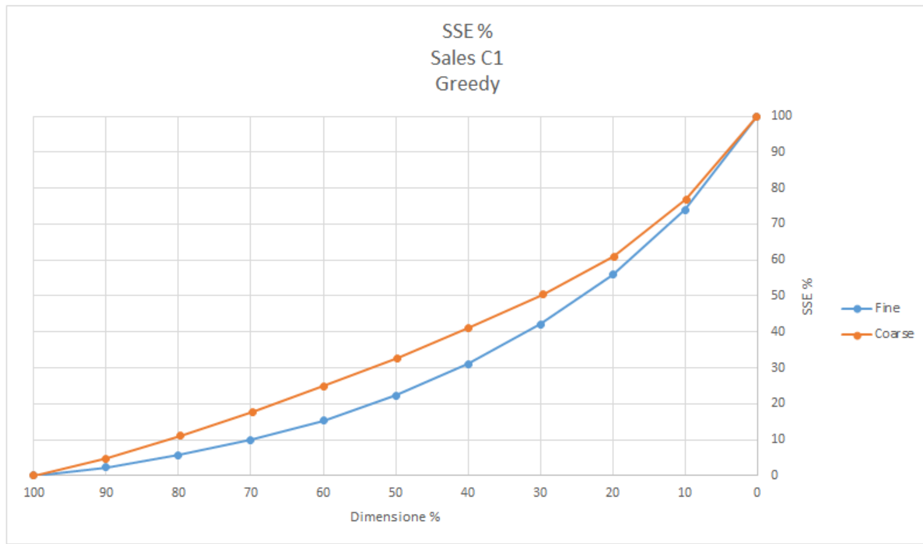


Figura 4.5: SSE degli algoritmi greedy fine e coarse per il cubo C1 del dataset sales.

4.2.2 Errore come coefficiente di variazione

Fino a questo momento abbiamo utilizzato l'andamento di SSE come termine di confronto tra i vari cubi e algoritmi. Per avere un'idea del livello di approssimazione indipendente dai dati, nelle Figure 4.8 e 4.9 è rappresentato l'andamento dell'errore misurato come *coefficiente di variazione* (equazione 4.2). Nel nostro caso il coefficiente di variazione viene calcolato come la media dei coefficienti delle celle appartenenti al cubo ridotto. Va notato che l'algoritmo è comunque guidato dall'SSE durante la riduzione e l'uso della deviazione standard relativa ha solo lo scopo di quantificare in modo più chiaro la distanza dei valori finali da quella dei valori originali.

Il coefficiente di variazione (o *deviazione standard relativa*) può essere calcolato come

$$\sigma^* = \frac{\sigma}{|\mu|} \tag{4.2}$$

dove σ rappresenta la deviazione standard e μ la media.

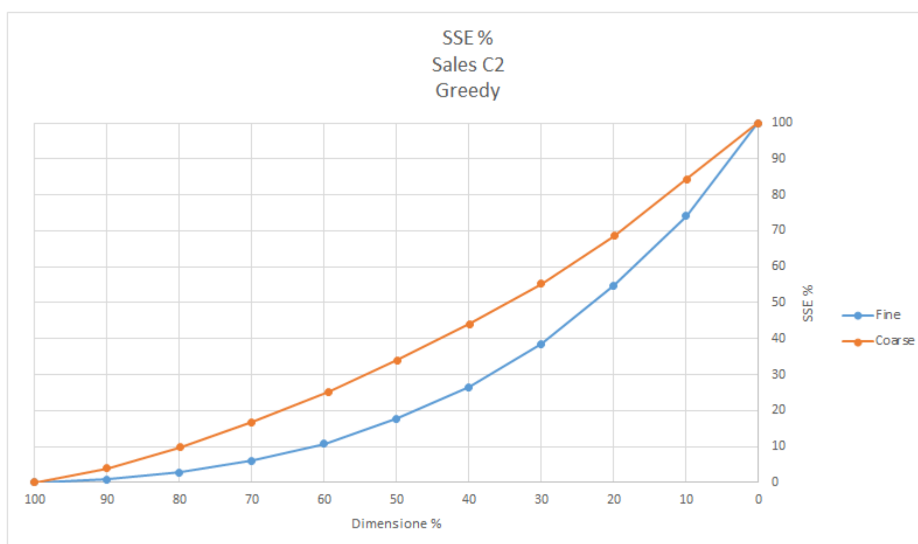


Figura 4.6: SSE degli algoritmi greedy fine e coarse per il cubo C2 del dataset sales.

4.2.3 Ottimo VS Greedy

In Tabella 4.3 sono confrontati i risultati ottimi di fine shrink assieme a quelli greedy. Data la complessità del problema è stato possibile risolvere solamente problemi piccoli. I test elencati hanno un tempo di esecuzione che vanno da qualche secondo—23 e 24 slice—a tre minuti circa—27 slice. Abbiamo provato anche provato con un cubo di 53 slice ma neanche dopo sei ore di esecuzione l'algoritmo è terminato. Come si può notare dalla colonna dei delta, i risultati sono molto vicini o identici. Test su problemi più complessi sono però necessari per fare valutazioni significative.

# slice	# celle iniziali	# celle finali	Ottimo	Greedy	Delta (%)
23	184	90	3,26099E+07	3,55681E+07	8,31%
24	192	90	4,15599E+05	4,15599E+05	0%
27	135	60	5,25283E+12	5,25283E+12	0%

Tabella 4.3: Confronto tra ottimo e greedy per fine shrink.

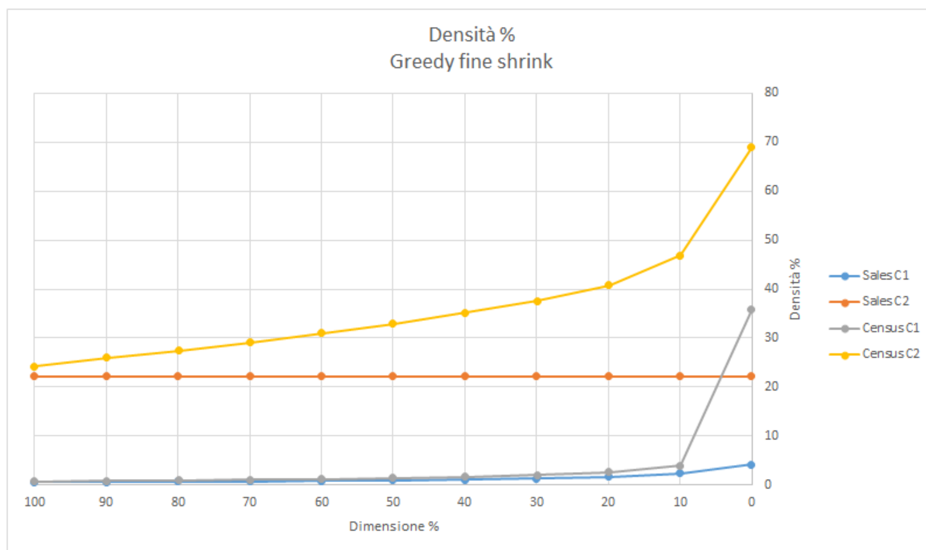


Figura 4.7: Variazione della densità al variare della dimensione di riduzione.

In Tabella 4.4 sono confrontati i risultati ottimi di coarse shrink assieme a quelli greedy. In questo caso è stato possibile ottenere risultati anche con input di dimensione non banale. A parte nel caso con 27 slice, la soluzione ottima corrisponde a quella greedy. La soluzione al problema con 512 slice è stata ottenuta in circa 1 ora e 15 minuti di esecuzione mentre gli altri problemi sono stati risolti in meno di 1 secondo.

# slice	# celle iniziali	# celle finali	Ottimo	Greedy	Delta (%)
27	135	55	3,34681E+13	3,63888E+13	8,72%
53	265	115	1,38197E+13	1,38197E+13	0%
512	2560	1190	2,76879E+12	2,76879E+12	0%

Tabella 4.4: Confronto tra ottimo e greedy per coarse shrink.

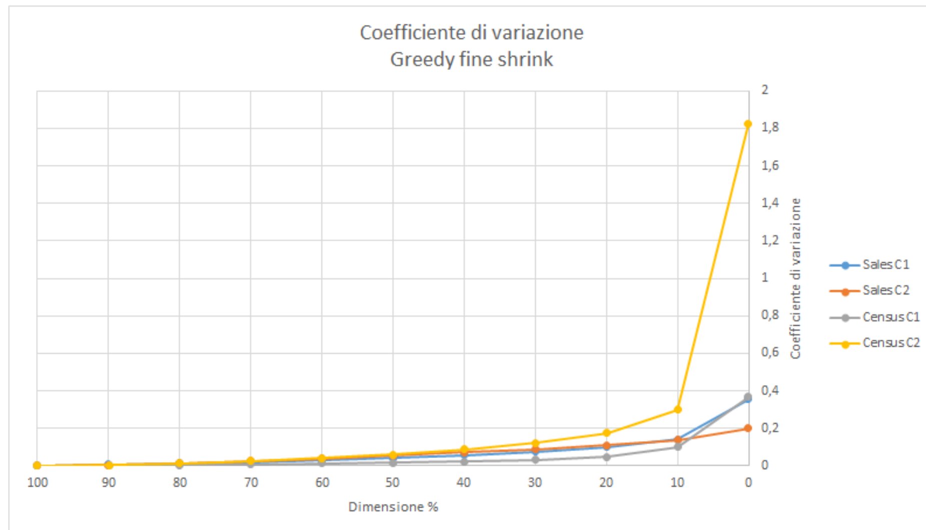


Figura 4.8: Coefficiente di variazione per l'algoritmo greedy fine shrink.

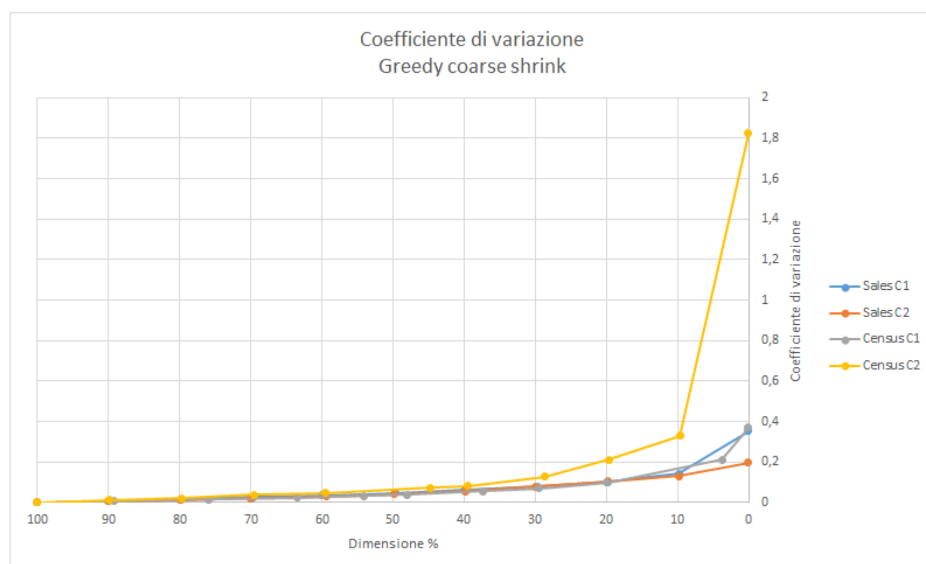


Figura 4.9: Coefficiente di variazione per l'algoritmo greedy coarse shrink.

4.3 Performance

4.3.1 Tempi di esecuzione

Nei seguenti test sono stati considerati i tempi riguardanti le sole operazioni di shrink (greedy), escludendo quindi tempi di query e popolamento del cubo.

Nei grafici 4.10 e 4.11 sono mostrati i tempi di esecuzione dei 4 cubi utilizzati anche per gli altri test. La prima cosa che cattura l'attenzione è la differenza tra la versione coarse e quella fine. Il primo è nettamente più veloce e ciò è naturale dato che il problema risolto è più semplice. Va fatto notare che la versione coarse può migliorare ancora di un buon margine in quanto il calcolo degli slice è stato implementato nella stessa modalità della versione fine—fondendo gli slice due a due.

Essendo i vincoli gerarchici più laschi, census C1 e C2 si dimostrano essere più impegnativi per fine shrink rispetto ai cubi sales. Census C1 per arrivare alla riduzione minima impiega quasi 2.5 secondi. Gli altri tempi di esecuzione rimangono invece ben inferiori ai 0.5 secondi.

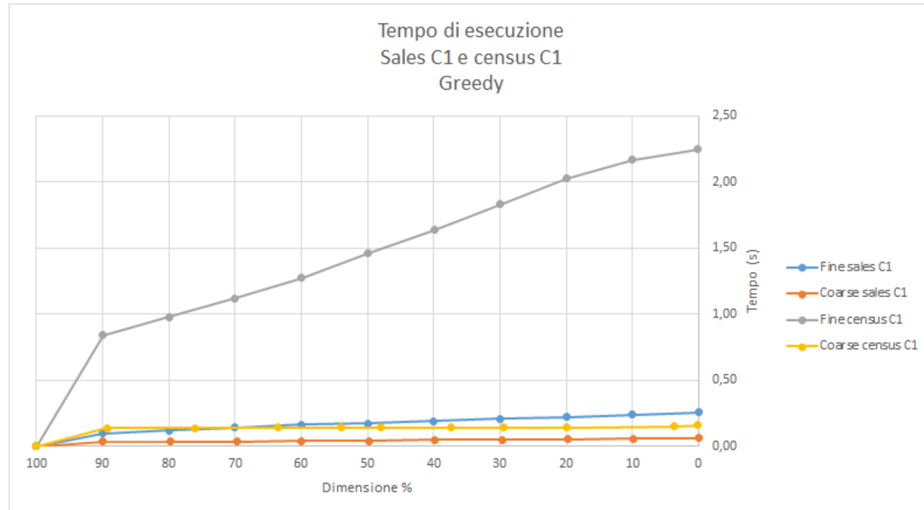


Figura 4.10: Tempi di esecuzione per i cubi sales C1 e census C1.

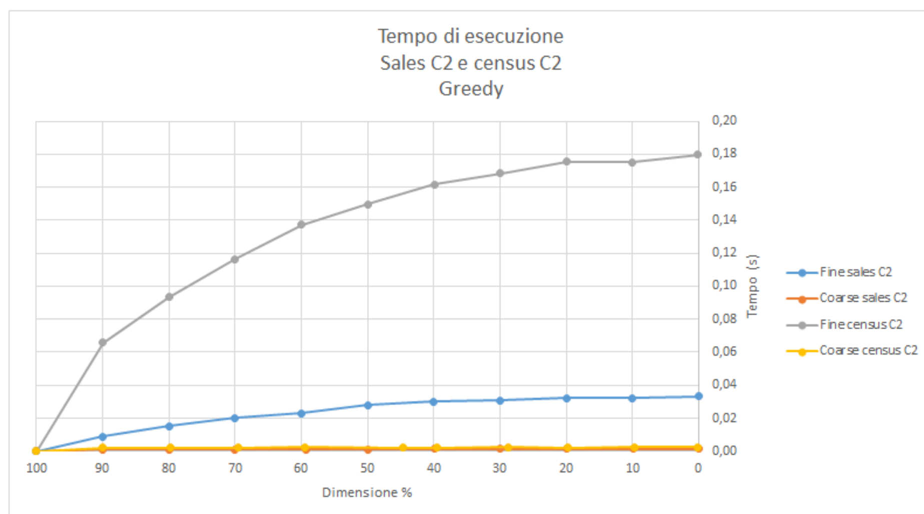


Figura 4.11: Tempi di esecuzione per i cubi sales C2 e census C2.

4.3.2 Utilizzo delle risorse computazionali

Passiamo ora a valutare l'utilizzo delle risorse computazionali per gli algoritmi greedy fine shrink e greedy coarse shrink. Per fare ciò abbiamo effettuato il *tracing* di due esecuzioni—una per ogni algoritmo—sul cubo Sales C1 con limite di dimensione massima fissato ad un singolo slice, ovvero al grado di riduzione massima possibile.

Per il tracing delle esecuzione è stato utilizzato il tool *Xperf* del pacchetto *Windows Performance Toolkit (WPT)* [wpt]. WPT è un pacchetto sviluppato da Microsoft per l'analisi delle performance di applicazioni software. In questo caso abbiamo deciso di analizzare i dati riguardanti la dimensione dell'heap in memoria centrale (Figure 4.12 e 4.13).

Le Figure 4.12 e 4.13 mostrano rispettivamente l'andamento della dimensione dell'heap per greedy fine shrink e per greedy coarse shrink. In entrambe possiamo notare una parte iniziale in cui l'heap cresce lentamente e in modo costante. Questa è la parte relativa al caricamento della struttura del cubo con i dati estratti dal database. La parte successiva si riferisce all'operazione di shrinking vera e propria. Qui fine shrink ha un andamento più costante rispetto a coarse shrink per via della diversa granularità delle aggregazioni. Il dato più interessante però è la dimensione massima raggiunta dall'heap: 40MB circa per fine shrink e 20MB circa per coarse shrink. Se contiamo che 10MB sono occupati dalla struttura del cubo in input, abbiamo che fine shrink occupa tre volte più memoria di coarse shrink. Similarmente alla valutazione dei tempi di esecuzione e come ci si potrebbe aspettare, abbiamo che fine shrink è nettamente più dispensioso di coarse shrink.

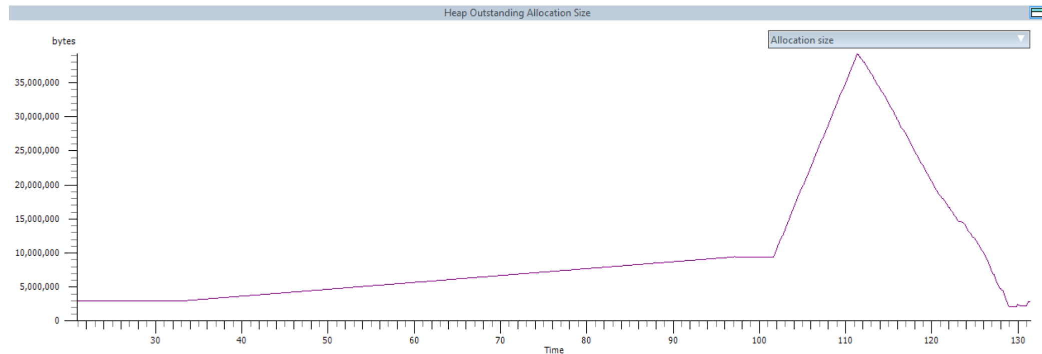


Figura 4.12: Andamento della dimensione dell'heap in memoria centrale per l'algoritmo greedy fine shrink. Il cubo in input è Sales C1 fissando il parametro di dimensione massima ad uno slice—riduzione massima.

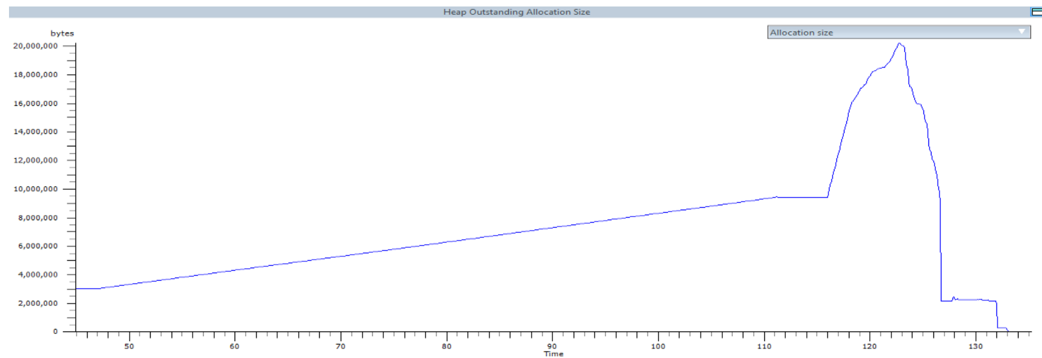


Figura 4.13: Andamento della dimensione dell'heap in memoria centrale per l'algoritmo greedy coarse shrink. Il cubo in input è Sales C1 fissando il parametro di dimensione massima ad uno slice—riduzione massima.

4.4 Valutazioni conclusive

Abbiamo valutato la qualità dei risultati e le performance degli operatori fine shrink e coarse shrink—versione greedy. Possiamo concludere che l'operatore fine shrink, rispetto a coarse shrink, ottiene risultati caratterizzati da un minor errore di approssimazione, al costo però di tempi di esecuzione più elevati e di una presentazione più complicata a causa delle liste di label. Non vi è quindi un'operazione migliore dell'altra ma la loro applicazione dipende dalle esigenze dell'utente.

Gli elementi che incidono sulla qualità dei risultati degli operatori sono:

- Struttura della gerarchia da comprimere: gerarchie con vincoli più stringenti consentiranno migliori risultati con coarse shrink ma avranno l'effetto opposto su fine shrink.
- Dimensione degli slice: più è grande lo slice e più sarà difficile avere buone aggregazioni.
- Densità dei dati: con dati più sparsi si otterrà generalmente un errore più basso grazie all'aggregazione di valori nulli.

Similmente i fattori che incidono sui tempi di esecuzione sono:

- Struttura della gerarchia da comprimere: gerarchie con vincoli più laschi appesantiscono la computazione per fine shrink ma la alleggeriscono per coarse shrink.
- Densità dei dati: a parità di dimensione di partenza, un cubo più denso avrà più fatti non nulli e quindi più dati da elaborare.

Capitolo 5

Conclusioni e sviluppi futuri

In questa tesi abbiamo presentato due nuove operazioni OLAP, chiamate fine shrink e coarse shrink, che permettono all'utente di gestire il risultato di una query OLAP per trovare un giusto compromesso tra dimensione e precisione. I due operatori sviluppati hanno lo stesso scopo di base ma differiscono nel risultato finale e nelle prestazioni in modo tale da essere entrambi utili in situazioni differenti. Per ognuno degli operatori abbiamo presentato algoritmi branch-and-bound per trovare la soluzione ottima e algoritmi euristici per ottenere risultati real-time al costo però di un maggior errore di approssimazione.

Tramite opportuni test abbiamo constatato che i risultati ottenuti sono nel complesso soddisfacenti. In particolare le versioni euristiche presentano una perdita di precisione accettabile a fronte della riduzione di dimensione apportata e ai bassi tempi di calcolo. Abbiamo inoltre confrontato tra loro i risultati dei due operatori fine e coarse shrink. Il primo presenta riduzioni caratterizzate da un minor errore di approssimazione rispetto al secondo, al costo però di maggiori risorse computazionali richieste e di una visualizzazione più complessa. In base alle esigenze sarà quindi più opportuno l'utilizzo di uno piuttosto che dell'altro.

Gli algoritmi euristici si sono rivelati molto efficaci anche al confronto con le versioni ottimali, in particolare coarse shrink presenta soluzioni molto vicine o addirittura identiche anche per problemi di dimensioni non banali. Nel caso di fine shrink, data la complessità del problema, non è stato possibile ottenere soluzioni di dimensioni importanti, i dati raccolti però mostrano uno scostamento ridotto anche in questo caso.

I passi successivi nello studio avranno principalmente l'intento di migliorare i risultati ottenuti, sia in termini di qualità che di performance. Le versioni ottimali in particolare presentano un buon margine di miglioramento che se sfruttato potrebbe rendere possibile la soluzione anche di problemi di dimensioni notevoli.

Nelle implementazioni correnti gli operatori di shrinking vengono applicati su una sola gerarchia, sarebbe quindi interessante vedere come cambierebbero i risultati se venissero applicati su più gerarchie contemporaneamente. Probabilmente in questo caso la precisione aumenterebbe a scapito di un maggior costo computazionale che potrebbe rendere necessario lo sviluppo di euristiche più efficienti.

Bibliografia

[AGPR99] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. *ACM SIGMOD Record*, 28(2):574–576, 1999.

[BFS03] Francesco Buccafurri, Filippo Furfaro, and C Sirangelo. A quad-tree based multiresolution approach for two-dimensional summary data. In *Scientific and Statistical Database Management, 2003. 15th International Conference on*, pages 127–137. IEEE, 2003.

[BGJ06] Michael Böhlen, Johann Gamper, and Christian S Jensen. Multi-dimensional aggregation for temporal data. In *Advances in Database Technology-EDBT 2006*, pages 257–275. Springer, 2006.

[CFS09] Alfredo Cuzzocrea, Filippo Furfaro, and Domenico Saccà. Enabling olap in mobile environments via intelligent data cube compression techniques. *Journal of Intelligent Information Systems*, 33(2):95–143, 2009.

[CL12] Alfredo Cuzzocrea and Carson K Leung. Efficiently compressing olap data cubes via r-tree based recursive partitions. In *Foundations of Intelligent Systems*, pages 455–465. Springer, 2012.

[FW02] Yu Feng and Shan Wang. Compressed data cube for approximate olap query processing. *Journal of Computer Science and Technology*, 17(5):625–635, 2002.

[GGB12] Juozas Gordevicius, Johann Gamper, and Michael H. Böhlen. Parsimonious temporal aggregation. *VLDB J.*, 21(3):309–332, 2012.

[GR06] M. Golfarelli and S. Rizzi. *Data warehouse: teoria e pratica della progettazione*. McGraw-Hill Companies, 2006.

[GRB11] Matteo Golfarelli, Stefano Rizzi, and Paolo Biondi. myOLAP: An approach to express and evaluate OLAP preferences. *IEEE Trans. Knowl. Data Eng.*, 23(7):1050–1064, 2011.

[Han97] Jiawei Han. OLAP mining: Integration of OLAP with data mining. In *Proc. Working Conf. on Database Semantics*, pages 3–20, Leysin, Switzerland, 1997.

[HHW97] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. Online aggregation. In *ACM SIGMOD Record*, volume 26, pages 171–182. ACM, 1997.

[KS95] Nick Kline and Richard Thomas Snodgrass. Computing temporal aggregates. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 222–231. IEEE, 1995.

[LL08] Tiancheng Li and Ninghui Li. Towards optimal¡ i¿ k¡/i¿-anonymization. *Data & Knowledge Engineering*, 65(1):22–39, 2008.

[LMBC12] Hsun Ming Lee, Francis A M? ndez Mediavilla, Enrique P Becerra, and James R Cook. Approximation queries for building energy–aware data warehouses on mobile ad hoc networks.

International Journal of Information and Decision Sciences, 4(1):1–18, 2012.

[MFVLI03] Bongki Moon, Ines Fernando Vega Lopez, and Vijaykumar Immanuel. Efficient algorithms for large-scale temporal aggregation. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):744–759, 2003.

[Min08] Minnesota Population Center. Integrated public use microdata series. http://www.ipums.org, 2008.

[MMR12] Patrick Marcel, Rokia Missaoui, and Stefano Rizzi. Towards intensional answers to OLAP queries for analytical sessions. In *Proc. DOLAP*, pages 49–56, Maui, USA, 2012.

[mon] Pentaho mondrian. http://mondrian.pentaho.com/.

[PG99] Viswanath Poosala and Venkatesh Ganti. Fast approximate answers to aggregate queries on a data cube. In *Scientific and Statistical Database Management, 1999. Eleventh International Conference on*, pages 24–33. IEEE, 1999.

[TS06] Pang-Ning Tan and Michael Steinbach. Vipin kumar, introduction to data mining, 2006.

[VW99] Jeffrey Scott Vitter and Min Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proc. SIGMOD*, pages 193–204, Philadelphia, USA, 1999.

[wpt] Windows performance toolkit (wpt). http://msdn.microsoft.com/en-us/performance/cc825801.aspx.