Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Progettazione di un sistema proattivo per l'identificazione di aggregati

Relatore:
Chiar.mo Prof.
Matteo Golfarelli

Presentata da:
Matteo Corfiati

Correlatore:
Dott.ssa
Simona Santoro

Sessione III
Anno Accademico 2013/2014

Ai miei quattro nonni...

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Uno dei problemi principali dei *data warehouse* è rappresentato dalle *performance*. Se inizialmente veniva utilizzato un approccio *"indipendent data mart"*, oggi si tende sempre più a una visione *enterprise* che si rifà maggiormente all'idea di Kimball. Questo passaggio porta sicuramente tanti benefici, il principale è l'integrità dei dati (in una visione a data mart indipendenti la tabella PRODOTTO potrebbe essere replicata in più aree di business e aggiornata partendo da fonti differenti, quindi essere inconstistente tra i vari settori aziendali); tuttavia avere un data warehouse centralizzato o delle dimensioni conformi che legano i data mart porta anche a un aumento dei dati gestiti e un conseguente aumento dei tempi di risposta a fronte di un'analisi. Gli analisti e i ricercatori di Gartner Group[1] identificano il miglioramento delle performance e l'ottimizzazione come una delle nove tendenze chiave che dovrebbero essere rispettate dai vendor di soluzioni di data warehousing [GP11].
Se in ambienti OLTP l'obiettivo del *tuning* è la massimizzazione del *throughput*, ovvero l'aumento del numero di transazioni per unità di tempo, in sistemi OLAP occorre minimizzare il tempo di risposta delle analisi.
Diverse sono le tecniche per controllare e migliorare le *performance* di un *data warehouse*, e si possono differenziare tra loro se vengono effettuate a livello logico o a livello fisico. Il *monitoring* fornisce un aiuto passivo al DBA o al consulente che si occupa della gestione del *data warehouse*; può utilizzare

[1]Gartner, Inc è l'azienda di ricerca strategica nel campo dell'Information Technology leader al mondo. Obiettivi principali sono la ricerca, il benchmarking, la consulenza in informatica e l'analisi dei nuovi trend a livello globale.

la raccolta delle statistiche gestita dal DBMS, rifacendosi alle *query* fisiche, oppure quelle memorizzate dal sistema di BI basandosi sulle *query* logiche. Diversi sono gli studi fatti e i prodotti sviluppati nel corso degli anni che hanno come obiettivo quello di migliorare le *performance* del *data warehouse*. Ogni *vendor* propone delle *best practice* sul *tuning* (all'interno delle guide online) o dei moduli aggiuntivi ai loro sistemi, come Database Engine Tuning Advisor di Microsoft, SQL Tuning Advisor di Oracle, Pentaho Reporting e tanti altri. Ognuno di questi prodotti è ovviamente legato al proprio *vendor* e non esiste un software indipendente dalla piattaforma.
Molte sono anche le ricerche effettuate sia a livello accademico che aziendale. Data la vastità di articoli e *paper* sull'argomento, ci si concentrerà sulle tecniche che lavorano sul *physical layer*, dove il miglioramento delle performance può essere più significativo. Oltre all'utilizzo di memorie *cache* o all'aumento del livello di parallelismo delle singole *query*, due sono i metodi più studiati e solitamente adottati in un normale processo aziendale da un DBA: il *design* degli indici e delle viste materializzate.
Obiettivo di questa tesi è analizzare le tecniche e gli algoritmi noti utilizzati in processi di *tuning* del *data warehouse* e progettare, e in seguito implementare, uno strumento indipendente dalla tecnologia che fornisca informazioni utili per la risoluzione del problema.

Capitolo 1

Data warehousing e *business intelligence*

In questo capitolo viene mostrata l'evoluzione che le basi di dati hanno avuto all'interno del mondo aziendale e la crescita sempre maggiore del'importanza delle informazioni in campo decisionale. Viene quindi analizzata la contrapposizione tra sistemi transazionali e sistemi analitici, e di questi ultimi vengono dettagliati quelli facenti parte di una specifica tipologia, i *data warehouse*.

1.1 L'evoluzione delle basi di dati

Le basi di dati han sempre avuto un ruolo centrale all'interno dei sistemi informativi aziendali. Le principali funzioni sono due: avere un'applicazione funzionale a livello operazionale (o transazionale) ed essere utilizzate per processi di analisi.
Storicamente i dati hanno assunto valore quando si è iniziato ad utilizzarli per la gestione dei processi di *business* a livello di transazioni, quindi il loro principale scopo era quello di valorizzare e tenere traccia degli eventi aziendali. Col passare del tempo si è notato che il possedere informazioni utili alle analisi rappresentava un'evoluzione radicale nel concepire l'intero sistema in-

formativo. Il cambiamento sostanziale che è stato fatto è quello di dividere concettualmente e fisicamente le due macro-categorie (quella transazionale e quella analitica) in due sistemi separati basandosi sulle motivazioni proposte da [Inm92], secondo cui la diversità di tipologie di utenti, di tecnologie, di elaborazioni e di rappresentazione dei dati mostrano una separazione sostanziale tra i due sistemi. E' così che nascono, in contrapposizione ai sistemi OLTP (*On-Line Transactional Process*), i DSS (*Decision Support System*).
L'insieme di processi aziendali, tecnologie di supporto e informazioni finali ottenute compongono la *Business Intelligence*, la "scienza" che ha come obiettivo quello di rendere disponibile al settore direzionale di un'azienda informazioni su cui basare le diverse strategie di mercato. In questo ambito i DSS fan parte del macro-settore che è la BI.
Il termine appare per la prima volta nel 1865 in uno scritto di Richard Millar Devens, nel quale l'autore spiega che il successo di un banchiere dell'epoca era stato dovuto alla conoscenza e comprensione dell'ambiente di interesse prima dei concorrenti. Nel 1958, l'informatico dell'IBM Hans Peter Luhn pubblica l'articolo "*A Business Intelligence System*" nel quale rappresenta la BI come un modo semplice per raccogliere e comprendere grandi quantità di informazioni in modo da poter prendere le decisioni migliori. Nonostante Luhn ne sia considerato il padre, la *business intelligence* come è conosciuta oggi nasce alla fine degli anni ottanta e inizia con la definizione di un processo specifico per la gestione e l'analisi dei dati (con la nascita dei DSS e l'evoluzione delle tecnologie).

1.2 Il *data warehousing*

Probabilmente tra i sistemi di supporto alle decisioni, i sistemi di *data warehousing* sono quelli di maggior interesse o, almeno, quelli più studiati nel mondo della ricerca e più utilizzati nel mondo aziendale.
Secondo [GR06], il *data warehousing* è una collezione di metodi, tecnologie e strumenti di ausilio al *knowledge worker* per l'analisi dei dati finalizzata ai processi decisionali e al miglioramento del patrimonio informativo. Il componente principale del processo è il *data warehouse* che, secondo [Inm92], è una collezione di dati di supporto per il processo decisionale con le seguenti caratteristiche:

- è orientata ai soggetti di interesse
- è integrata e consistente
- è rappresentativa dell'evoluzione temporale e non volatile

1.2.1 OLTP vs OLAP

La tabella 1.1 propone le principali differenze tra i due sistemi, quello OLTP e quello analitico, nel caso specifico il *data warehouse*. La prima tipologia è specifica per la gestione e l'esecuzione dei processi di *business* e si contrappone alla seconda, che analizza l'andamento degli stessi processi.
Per *subject-oriented* si intende incentrata sui concetti di interesse dell'azienda, quindi i clienti, le vendite, gli ordini, etc. Le prime sono caratterizzate da interazioni e aggiornamenti frequenti e continui, che si traducono in letture e scritture costanti di dati che rappresentano le singole transazioni da memorizzare e interrogare sul sistema informativo. Questo tipo di interrogazioni sono racchiuse nella logica applicativa del sistema informatico e questo non permette di gestire nessuna componente di impredicibilità all'interno delle stesse, contrariamente ai *data warehouse* dove la logica delle analisi viene decisa dal singolo utente che può modellare l'analisi secondo le proprie scelte,

rendendo caratteristica essenziale dei DSS l'interattività analitica. Ogni transazione dei sistemi operazionali è "atomica", quindi si riferisce alla lettura o alla scrittura del dato al massimo dettaglio disponibile (es. uno scontrino, un nuovo cliente, una fattura, etc), mentre i sistemi analitici si basano su informazioni aggregate. Possedere i dati non equivale però a disporre di informazioni necessarie al processo decisionale, i dati operazionali vengono quindi selezionati e aggregati per ottenere informazioni di sintesi.
Un'ulteriore differenza tra i due sistemi sta nel significato dato alla componente temporale. I primi rappresentano il presente: molti articoli o libri sull'argomento li paragonano a delle *snapshot* dell'azienda in uno specifico momento. Nei DSS si ha un approccio che valorizza la serie temporale rispetto alla fotografia statica e sono rivolti spesso all'analisi dello stato presente paragonato allo storico. Il tempo è quindi una dimensione logica d'analisi fissa per tutti i *data warehouse*.
Infine, a livello di progettazione si tende a denormalizzare i dati fusi per questioni di efficienza e prestazioni, mentre la normalizzazione rappresenta una caratteristica essenziale dei sistemi transazionali.
Riprendendo la definizione di Inmon, il *data warehouse* è caratterizzato dall'essere integrato e consistente. L'idea che sta alla base di questa affermazione è che il sistema si appoggia sui vari *database* operazionali utilizzandone i dati contenuti, potenzialmente molto eterogenei tra loro che, tramite procedure *ad hoc*, dette di ETL (*Extraction, Trasformation and Loading*), fondono i dati "puliti" e adattati ottenendo informazioni che ne danno una visione unificata.

1.2.2 Modello multidimensionale

Il modello di rappresentazione dei dati utilizzato nel *data warehousing* viene detto multidimensionale e il contenitore di dati è rappresentato idealmente come un ipercubo. Le *query* di analisi su questo modello risulterebbero complesse se tradotte in linguaggi come SQL, senza contare i tempi di risposta elevati. Al contrario, questa modalità modella come un processo di

Sistema operazionale	**Sistema analitico**
Esecuzione dei processi di business	Analisi dei processi di business
Letture/Scritture	Letture
Transazioni singole	Transazioni aggregate
Interrogazioni previste dalla logica	Interrogazioni inpredicibili
Presente	Presente e passato
Normalizzazione	Denormalizzazione

Tabella 1.1: La tabella mostra le principali caratteristiche dei sistemi OLTP e dei sistemi DSS, in modo da evidenziare le differenze maggiori.

business è misurato in relazione a uno specifico contesto. La misurazione è l'equivalente di un fatto del mondo aziendale (es. vendita, ordine, pagamento, etc.), rappresentato solitamente da uno o più valori numerici di interesse (detti misure) che assumono diversi significati se analizzati a diversi livelli di dettaglio. Il contesto, usato per filtrare i vari eventi accaduti, rappresenta il livello di dettaglio al quale viene effettuata l'analisi, quindi è l'insieme delle dimensioni dell'ipercubo fissate per ottenere le misure aggregate. Le operazioni di base che possono essere fatte sul cubo multidimensionale sono cinque: il *pivoting*, lo *slicing*, il *dicing*, il *roll-up* e il *drill-down*.

- L'operazione di *pivoting* permette di ruotare gli assi del cubo e, di conseguenza, il metodo di visualizzazione dei dati; considerando un cubo con n dimensioni, il numero di possibili viste ottenibili dal *pivoting* è $n!$.

- Lo *slicing* è l'istruzione che permette di fissare una o più dimensioni (es. "data = 10-10-2011"), mentre il *dicing* fissa un intervallo (es. "data tra 10-10-2011 e 15-10-2011").

- Il *roll-up* e il *drill-down* sono operazioni di aggregazione che permettono di navigare il cubo sulla gerarchia delle dimensioni, dando agli utenti la scelta del livello di aggregazione per la visualizzazione del dato (es. pas-

sando da "prodotto" a "categoria" si effettua un *roll-up* e si visualizza un livello di dettaglio minore con dati maggiormente aggregati).

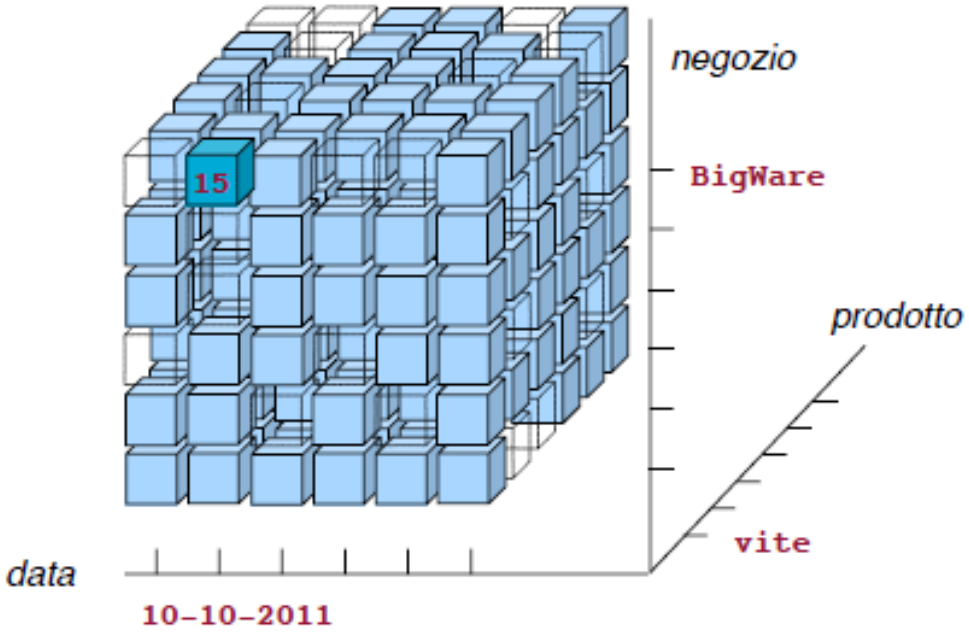


Figura 1.1: Rappresentazione grafica di un cubo multidimensionale. Fissando le tre dimensioni "data", "prodotto" e "negozio", si ottiene il valore non aggregato della misura, che può indicare ad esempio la quantità venduta del prodotto "vite" il giorno "10-10-2011" nel negozio "BigWare". Figura tratta da [GR06].

Le dimensioni al massimo livello di dettaglio sono dette primarie, le altre, in gerarchia rispetto alle prime, sono chiamate secondarie.
Non esiste uno standard comune per la modellazione di un *data warehouse* in fase di progettazione concettuale. Un possibile formalismo, a cui si farà riferimento durante questa tesi, è il DFM (*Dimensional Fact Model*), ovvero un modello concettuale grafico per *data mart*. Il DFM consiste in un insieme di schemi di fatto, che si compongono degli elementi di base del modello multidimensionale, ovvero fatti, misure, dimensioni e gerarchie.

Un *fatto* è un concetto di interesse per il processo decisionale, quindi un evento che accade periodicamente e abbia un aspetto dinamico.
Una *misura* è una proprietà numerica di un fatto che ne descrive quantitativamente un aspetto.
Una *dimensione* è una proprietà con dominio finito che rappresenta una coordinata di analisi del fatto. Fissando un valore per ciascuna dimensione si ottiene un evento primario, a cui è associato un valore per ogni misura.
Una *gerarchia* è un albero direzionato i cui nodi sono attributi dimensionali, quindi dimensioni primarie e secondarie, e i cui archi modellano associazioni molti-a-uno tra coppie di nodi.
Questi sono gli elementi di base del modello multidimensionale che vengono poi arricchiti con i costrutti avanzati del DFM, quali attributi descrittivi, archi multipli, gerarchie condivise, convergenze, attributi cross-dimensionali, archi opzionali, gerarchie incomplete e ricorsive.

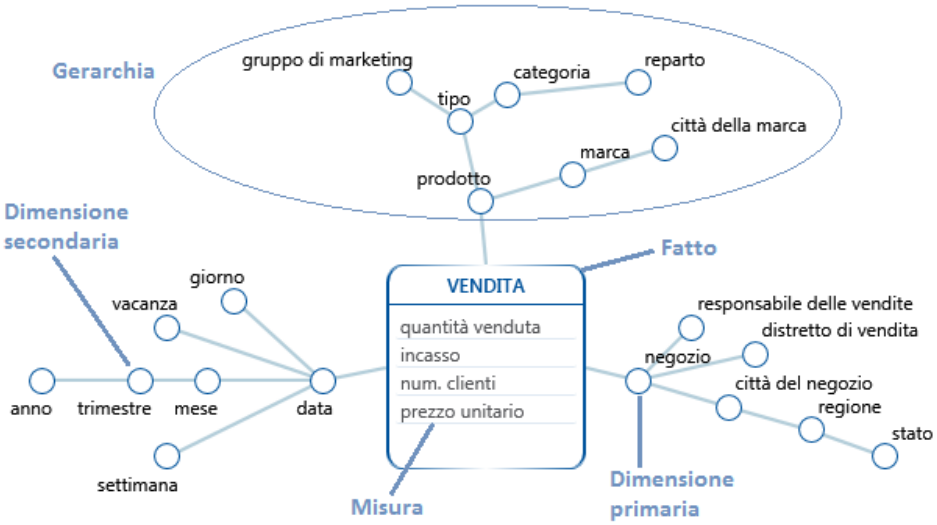


Figura 1.2: Esempio di DFM di un fatto delle vendite: all'interno sono riportate le misure, mentre collegate al fatto ci sono le dimensioni primarie, che rappresentano le radici delle gerarchie, i cui nodi interni sono le dimensioni secondarie. Figura liberamente tratta da [GR06].

1.2.3 L'architettura di un *data warehouse*

L'architettura di un *data warehouse* può essere di varie tipologie, ma la suddivisione più semplice e utilizzata è quella a livelli. L'obiettivo principale su cui viene poi scelta l'architettura in fase progettuale è quello di garantire alcuni requisiti essenziali al sistema, tra cui quello di scalabilità, estendibilità, sicurezza (intesa come controllo degli accessi) e di separazione tra elaborazioni analitiche e transazionali.
Secondo la distinzione classica le tipologie sono tre:

- le architetture a un livello separano le sorgenti dati dagli strumenti analitici di reportistica attraverso un *layer* intermedio che ricopre la funzione di *middleware*. Questa soluzione serve a evitare spese aggiuntive di installazione e manutenzione dell'*hardware* necessario alla storicizzazione dei dati aggregati, ma non è una buona idea in termini di prestazioni e di modello;

- le architetture a due livelli introducono il *layer* di alimentazione; questo livello ha il compito di estrarre, pulire e integrare i dati presenti nelle sorgenti attraverso opportuni strumenti di ETL, per poi caricarli fisicamente su un *data warehouse*; in questo modo è maggiore la separazione tra i due sistemi che, a meno dei caricamenti, diventano due ambienti completamente distinti, che possono esistere l'uno indipendentemente dall'altro e le cui interrogazioni non incidono sul carico di lavoro dell'altro. Il livello del *warehouse* può contenere i *data mart*, ovvero dei sottoinsiemi o delle aggregazioni di dati rivolte a un soggetto specifico o contenenti informazioni su una specifica area di *business*. I *mart* vengono detti dipendenti se alimentati dal *data warehouse* e indipendenti se i caricamenti avvengono direttamente dalle sorgenti;

- le architetture a tre livelli sono caratterizzate dal cosiddetto "*operational data store*" (o database riconciliato): i dati ottenuti dal processo di ETL non vengono caricati direttamente sul *data warehouse*, ma memorizzati in una banca dati intermedia. Questo database ha come

contro l'introduzione di ridondanza, ma contenendo dati integrati e consistenti permette di alimentare il *data warehouse* (o direttamente i *mart*) avendo un significato univoco per ciascun concetto di interesse per l'azienda.

Una classificazione alternativa è quella che contrappone le architetture proposte dai maggiori studiosi di *data warehouse*: si distingue tra *Corporate Information Factory*, *Dimensional Data Warehouse* e *Stand-alone data mart*. La prima è l'architettura proposta e incentivata da Inmon, che prevede un *repository* centrale detto *Enterprise Data Warehouse* progettato in terza forma normale. Questa base di dati normalizzata funge da *database* riconciliato che viene poi utilizzato per alimentare i *data mart* fisici.

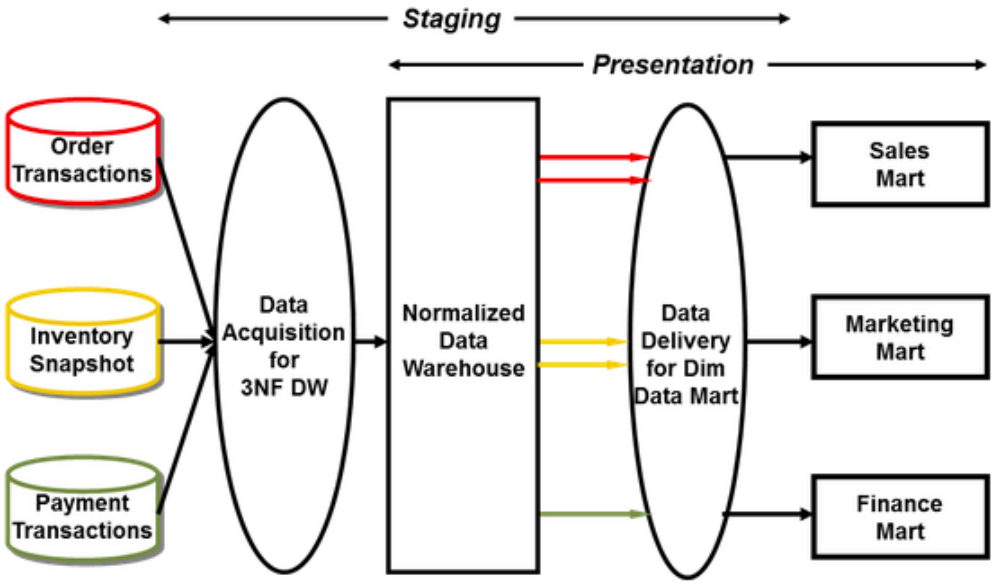


Figura 1.3: Schema logico del *Corporate Information Factory* (CIF). Figura tratta da [Ros04].

L'idea proposta da Kimball viene detta *Dimensional Data Warehouse*: anche questa prevede un *repository* centralizzato che segue però un modello dimensionale. I *data mart* non sono fisicamente distinti ma diventano separazioni logiche del *data warehouse* completo. Le dimensioni vengono dette

conformi perché condivise tra tutti i *mart* grazie a un "*bus*" di metadati valido per tutte le aree di *business*. I *tool* OLAP vanno quindi a interrogare direttamente il *repository* rendendo più semplici e rapide le *query*.

L'ultima architettura degna di nota è quella dei *data mart* indipendenti (o

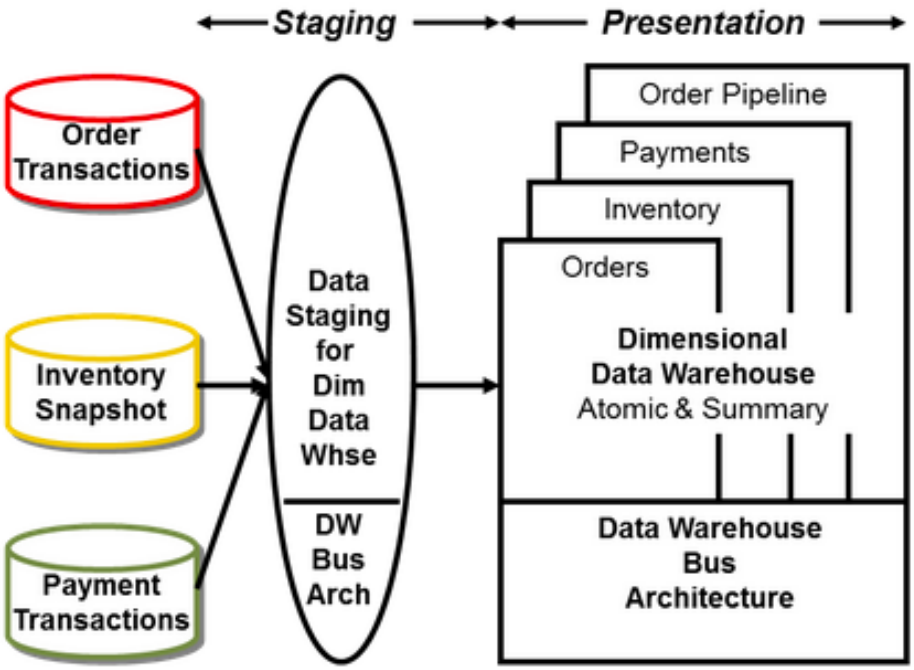


Figura 1.4: Schema logico del *Dimensional Data Warehouse*. Figura tratta da [Ros04].

stand-alone data mart). In questo caso non esiste un *repository* centralizzato che garantisca l'integrità e la consistenza dei dati, ma permette di ottenere rapidamente dei risultati a costi ridotti. Ogni *data mart* viene creato in modo indipendente dagli altri ed eventuali dimensioni comuni vengono replicate e potenzialmente rese inconsistenti.

1.2.4 Modello di implementazione

I sistemi OLAP possono essere di tre tipologie, definite dall'approccio implementativo del *data warehouse* in termini di rappresentazione del dato:

ROLAP (*Relational OLAP*), MOLAP (*Multidimensional OLAP*) e HOLAP (*Hybrid OLAP*). I sistemi ROLAP sono caratterizzati dal fatto che i dati sono memorizzati in DBMS relazionali; i ROLAP sono maggiormente scalabili rispetto ai MOLAP, ma più lenti durante l'elaborazione delle interrogazioni. Inoltre necessitano di nuove tipologie di schemi per rappresentare il modello multidimensionale in quello relazionale: in questo ambito si contrappongono *star schema* e *snowflake schema*. La differenza principale è che i primi hanno una struttura completamente denormalizzata: è presente una tabella relazionale per ciascuna dimensione primaria (chiamata *dimension table* o *lookup table*), che contiene tutti i livelli gerarchici rappresentati a modello, e una tabella per ciascun fatto (detta *fact table*), che contiene una *foreign key* per ogni *dimension table* e le misure che la caratterizzano.

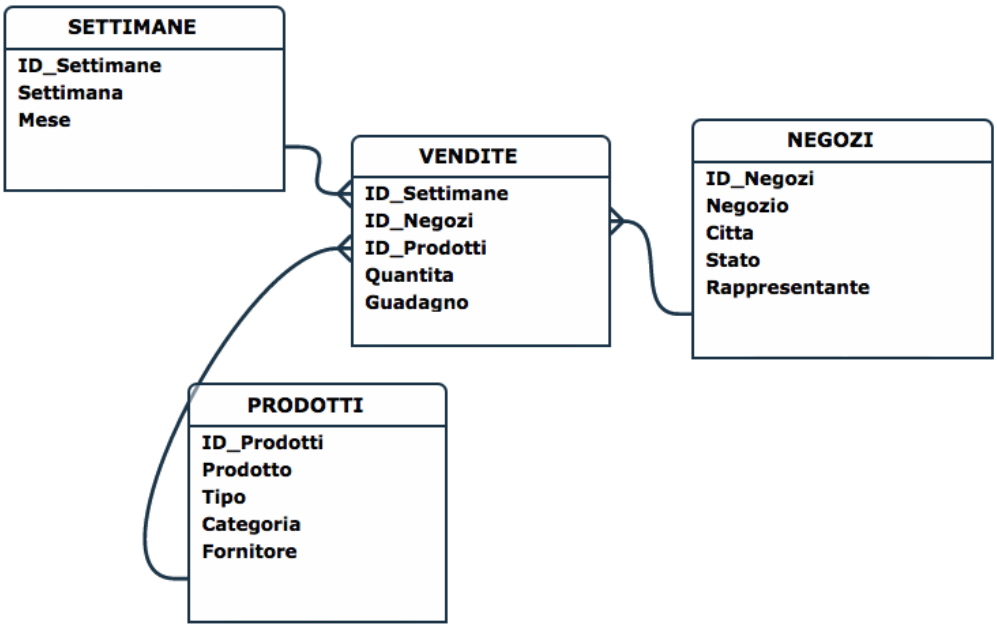


Figura 1.5: Esempio di *star schema*: il fatto delle vendite contiene tutte le *foreign key* delle *dimension table*, che non sono in terza forma normale (es. "Categoria" dipende funzionalmente da "Tipo", che non è la *primary key* della tabella). Figura liberamente tratta da [GR06].

Schemi di tipo *snowflake* tendono a normalizzare il modello, riducendo la ridondanza dei dati e garantendo maggiore integrità e consistenza; la denormalizzazione è una tecnica molto utilizzata nell'implementazione dei *data warehouse* dato che il principale problema di questo tipo di sistemi è quello delle prestazioni, essa permette infatti di ridurre notevolmente il numero di *join* da effettuare quando si aggrega il dato ad un dettaglio minore.

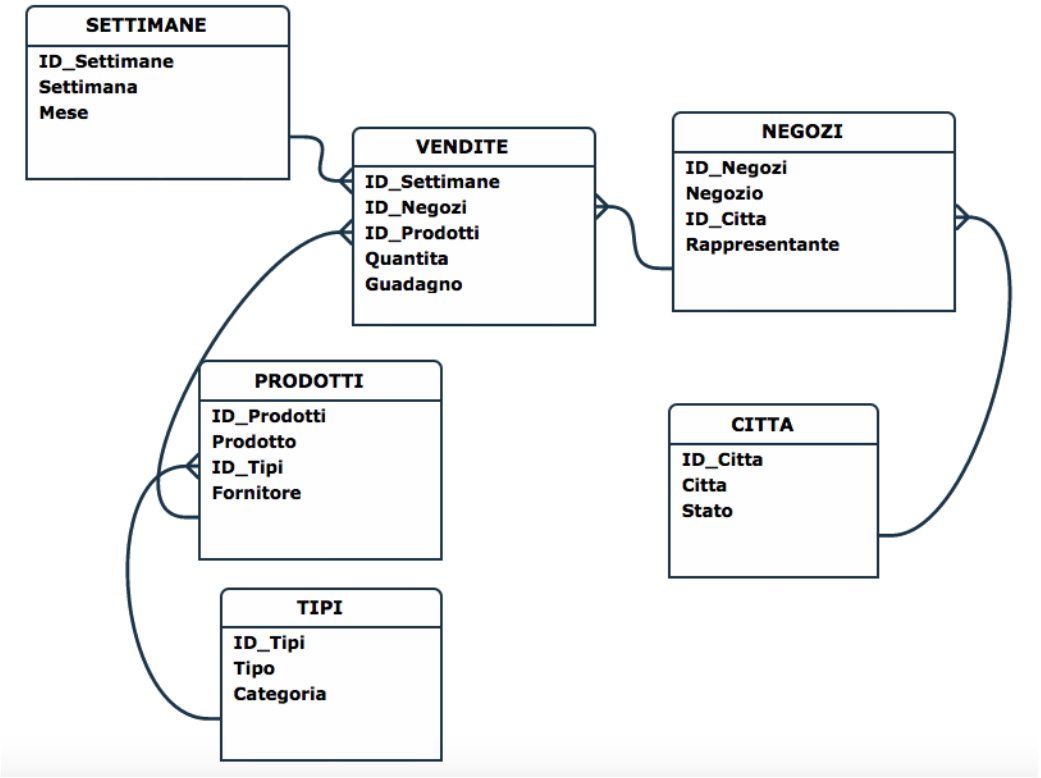


Figura 1.6: Esempio di *snowflake schema*: rispetto allo *star schema* aumenta il grado di normalizzazione delle *dimension table*. Figura liberamente tratta da [GR06].

Un'architettura ROLAP necessita di un *middleware* tra il *database* relazionale su cui risiedono i dati e il *client* OLAP su cui gli utenti eseguono le analisi: questo strumento intermedio traduce le interrogazioni degli utenti in istruzioni SQL che vengono poi eseguite sul *server* e restituite allo strumento di *front-end* che ne fornisce una visualizzazione multidimensionale.

I sistemi MOLAP sono caratterizzati da una gestione *ad hoc* ed ottimizzata dei dati, memorizzati in strutture vettoriali accedute in modo posizionale. Al contrario del ROLAP, un sistema di questo tipo non necessita del *middleware* intermedio dato che le analisi degli utenti sono direttamente definibili come operazioni multidimensionali. Come già affermato, il punto di forza di un'architettura ROLAP è la scalabilità, questo perchè il problema principale dei sistemi MOLAP è la sparsità dei dati e la conseguente dimensione delle strutture memorizzate.
Da ultimo è possibile anche definire un'architettura ibrida che permette la gestione di entrambi i modelli o prende caratteristiche da tutti e due, dette comunemente HOLAP.

Capitolo 2

Stato dell'arte

In questo capitolo vengono presentati i principali studi riguardo l'ottimizzazione del *data warehouse* a livello fisico, concentrandosi quindi sulle modalità di scelta degli indici e delle viste.

Indici

A livello teorico il B^+-tree, l'indice più utilizzato dai DBMS commerciali, non è sufficiente nel *data warehousing* poiché fornisce buone prestazioni quando la selettività dei predicati è elevata (es. cliente), mentre le interrogazioni OLAP sono spesso caratterizzate da attributi con selettività bassa (es. sesso). Vengono quindi presi in considerazione altri tipi di indici:

- Indici bitmap, utili se creati su attributi con cardinalità bassa e se associati ai B^+-tree
- Indici di join, che contengono le coppie di RID (*Row ID*) che soddisfano la condizione di join
- Indici a stella, che estendono l'indice di join riportando l'associazione tra il RID della Fact Table e i RID delle dimensioni che soddisfano i predicati

In [GRS02] viene riportato un approccio euristico per selezionare un insieme di indici candidati da creare su una implementazione ROLAP. L'idea di base è quella di simulare un ottimizzatore *rule-based* che generi piani di esecuzione basandosi su un modello di costo che tiene in considerazione informazioni sulle tabelle (come il numero di tuple, la dimensione del singolo record e il numero di pagine di disco occupate), sugli indici considerati (numero di foglie, numero di valori distinti, altezza dell'albero, etc) e informazioni di sistema (dimensione delle pagine di disco, dimensione dei buffer, etc). Infine viene proposto un algoritmo *greedy* per la scelta dell'indice migliore tra quelli candidati considerando come vincolo lo spazio su disco occupato dall'indicizzazione.

[CN97] mostra le caratteristiche di un *tool* che, partendo da un *workload* di *query* SQL, suggerisce alcuni indici creabili su determinati attributi in modo da velocizzare il carico dati fornito come input. Considerando che il numero di possibili indici creabili, a fronte di un *workload* non banale, è molto grande, l'algoritmo riduce l'insieme in tre fasi successive. Nella prima, viene creato l'insieme di indici candidati con una tecnica che gli autori chiamano *query-specific-best-configuration*; l'idea alla base dell'algoritmo è trovare gli indici candidati per ogni *query* del *workload* in modo indipendente dalle altre, assumendo che se una configurazione (insieme di indici) non fa parte della soluzione migliore per la singola *query*, è improbabile che faccia parte della soluzione ottimale per l'intero carico di lavoro. La seconda fase si basa su un'ottimizzazione (chiamata *Configuration Enumeration*) che rende possibile una valutazione non troppo costosa della bontà degli indici candidati. Infine, l'ultima parte propone un approccio iterativo per gestire la complessità derivante dagli indici creati su più colonne (data una tabella con K attributi, K! sono gli indici *multi-column* possibili): partendo da un insieme di indici *single-column*, si crea l'insieme dei possibili indici su due colonne; questo insieme, con il migliore tra gli indici *single-column*, è l'*input* per una nuova iterazione. L'algoritmo è chiamato MC-LEAD e sceglie un indice *two-column* se sulla sua colonna principale vi è un indice candidato. Una variante più

restrittiva è MC-ALL dove la soluzione contiene indici creati su entrambe le colonne.

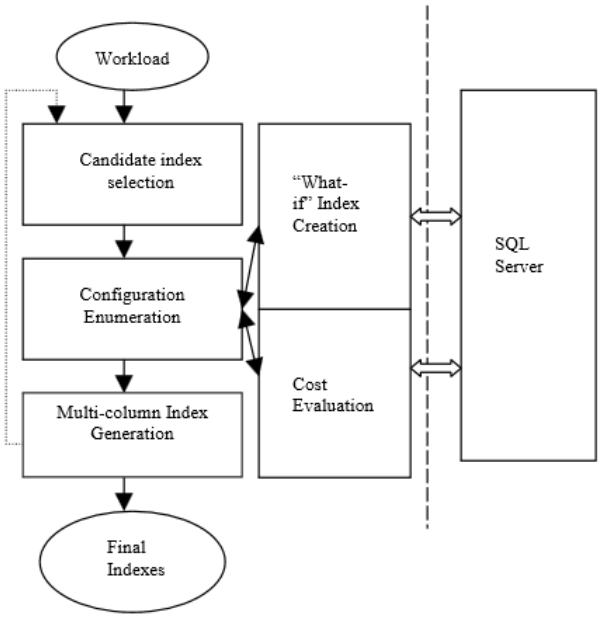


Figura 2.1: Architettura dell'*Index Tool Selection* (fonte [CN97])

Questo algoritmo è stato implementato nel progetto AutoAdmin [CDK+], il cui obiettivo è di rendere i DBMS *self-tuning*, rimuovendo la figura del *database administrator*. L'area principale che il *tool* va ad analizzare è quella della progettazione fisica del database, in particolare la scelta degli indici. Grazie al lavoro condiviso dei ricercatori del progetto AutoAdmin e il gruppo di Microsoft SQL Server è nato nel 1998 l'Index Tuning Wizard, il primo strumento di quel tipo ad essere inserito in un prodotto commerciale (da questo sono poi stati sviluppati tutti i *tool* di gestione delle performance inseriti come moduli dei DBMS dei vari *vendor*). La caratteristica che lo contraddistingue è la gestione di una componente che lavora in modo parallelo a quello di valutazione dei costi, che effettua un'analisi *what-if* sulla creazione degli indici (fig 2.1).

Viste materializzate

Un'altra modalità per migliorare il tempo di risposta delle query è la creazione di viste materializzate, ovvero strutture fisiche che riducono i tempi di accesso precomputando risultati intermedi.

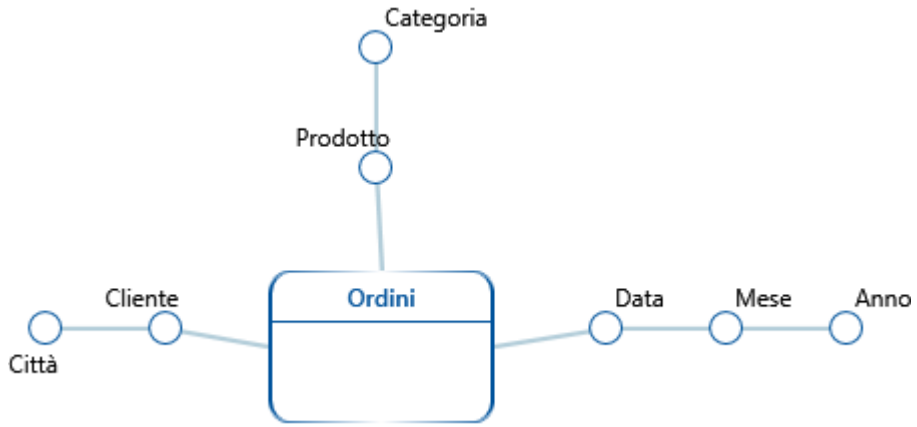


Figura 2.2: Esempio di schema degli ordini

In un sistema di *data warehousing* le analisi solitamente non vengono fatte al massimo livello di dettaglio ma su dati aggregati, che riportano informazioni di sintesi. Le viste rappresentano quindi le *fact table* contenenti i dati aggregati. Considerando la figura 2.2, possibili viste, identificate dal loro livello di aggregazione (quindi dal *group-by set*), potrebbero essere {categoria, mese, città} oppure {prodotto, anno}. La vista {prodotto, data, cliente} viene detta primaria, perché fa riferimento al *group-by set* primario, quindi al fatto non aggregato; le altre, dove il livello di aggregazione non è massimo, vengono dette secondarie.

Il lattice (o reticolo) multidimensionale è un grafico che riporta tutti i possibili livelli di aggregazione partendo dal DFM.

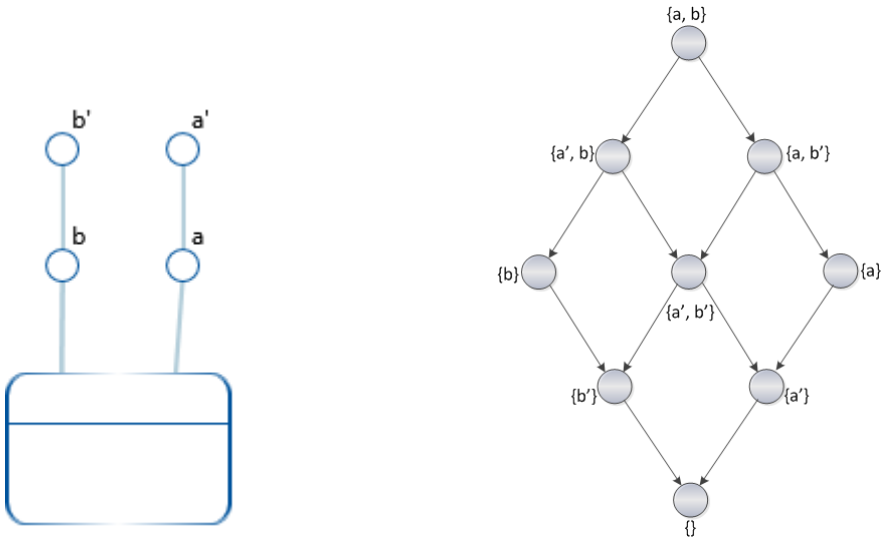


Figura 2.3: Reticolo multidimensionale corrispondente all'esempio di DFM

Se si creano più *fact table* aggregate, la scelta di quale tabella utilizzare determina il costo di esecuzione per soddisfare la *query*. Gli *aggregate navigator* sono dei moduli dei sistemi OLAP che in modo automatico scelgono quale vista interrogare tra quelle a disposizione, senza che si richieda all'utente finale di conoscere la distinzione fisica tra le varie tabelle. In questo modo l'*end-user* non interroga mai l'aggregato ma sempre il fatto al massimo livello di dettaglio, e sarà l'*aggregate navigator* ad intercettare la *query* SQL e riscriverla sulla vista migliore [Kim95].

La scelta delle viste da materializzare è un ambito di ricerca molto trattato nel mondo accademico, data la sua natura *NP-hard* [YCGY02], soprattutto quando il problema è complicato da vincoli di sistema e di risorse, come lo spazio su disco, il tempo di aggiornamento, il tempo di risposta, ecc.

In letteratura diversi sono i punti trattati in questo campo: partendo dai modi con cui determinare le viste potenziali, analizzando i *framework* usati per catturare le relazioni tra i candidati, arrivando alla scelta della funzione di costo basata su modelli matematici o su *query optimizer*.

Tutti gli algoritmi esistenti possono essere divisi in quattro categorie [ZYY01]:

Algoritmi deterministici ottengono una soluzione ottima o pseudo-ottima applicando euristiche o ricerche esaustive.

Algoritmi probabilistici (randomized algorithms) definiscono uno scenario iniziale in modo tale che differenti soluzioni siano connesse tra loro, cioè trasformate l'una nell'altra in uno step, nell'intero spazio delle soluzioni. Si modificano le soluzioni in base a determinate regole, terminando quando nessuna trasformazione è più applicabile o un tempo limite viene raggiunto.

Algoritmi evolutivi (evolutionary algorithms) utilizzano una ricerca probabilistica basata sull'evoluzione biologica, partendo da una popolazione iniziale scelta randomicamente e generando variazioni casuali con delle operazioni standard (es. *crossover* o mutazione).

Algoritmi ibridi combinano algoritmi deterministici e probabilistici in vari modi, ad esempio utilizzando il risultato dell'algoritmo deterministico come popolazione iniziale per quello evolutivo.

In [HRU96] e in [BPT97] gli autori si basano sulla creazione di un lattice multidimensionale e su specifici operatori per determinare le dipendenze tra le varie viste. Materializzare tutte le viste del reticolo non è fattibile in sistemi *enterprise* reali, visto che il numero di nodi del lattice cresce esponenzialmente ed è

$$n_{total} = \prod_i (2^{n_i} + 1) \tag{2.1}$$

dove i è il numero di dimensioni del cubo e n_i è il numero di attributi non chiave dell'i-esima dimensione. Questa formula rappresenta un *upper-bound* se le dimensioni contengono gerarchie, ma rimane comunque un valore molto grande. Entrambi i *paper* propongono degli algoritmi *greedy* analizzando,

per ogni vista, gli *ancestor*[1] : il primo utilizza un approccio iterativo per trovare le k viste del reticolo che massimizzano, tramite una ricerca locale, un guadagno B (ovvero la vista che occupa lo spazio minore), mentre il secondo utilizza due euristiche per ridurre le viste candidate; *in primis* si crea un sottoinsieme delle viste totali chiamato MDred-lattice (ridotto grazie all'analisi di un insieme di *query* Q), mentre la seconda euristica riduce ulteriormente lo spazio di ricerca poiché, al momento della creazione del MDred-lattice, evita l'analisi di viste la cui dimensione stimata è maggiore di una certa soglia.
[SDN] propone una modifica all'algoritmo BPUS (*Benefit Per Unit Space*) contenuto in [HRU96]; gli autori affermano che l'algoritmo descritto nel *paper* riporta una soluzione vicina all'ottimo, ma la sua debolezza sta nel tempo di esecuzione. La miglioria apportata (PUS, *Pick By Size*) sta nella scelta della vista da inserire nella soluzione ad ogni iterazione: non viene scelta quella con dimensione minore nello spazio locale, ma quella più piccola tra tutte le viste ancora non in soluzione. Questo permette di non calcolare ad ogni iterazione tutte le dimensioni aggiornate riducendo il tempo che l'algoritmo impiega per la sua esecuzione.
In [Gup97] viene presentato un *framework* teorico per generalizzare il problema. Si definiscono l'*AND graph*, dove ogni vista ha un'unica valutazione, l'*OR graph*, dove ogni vista può essere calcolata da tutte le altre ad essa collegate e l'*AND-OR graph*, il caso più generico. Grazie a questa riscrittura del problema, vengono quindi proposti alcuni algoritmi euristici con costo polinomiale.
Molti approcci, anche meno teorici e più attuabili, basano però la loro idea iniziale sulla riprogettazione del modello fisico.

[1] L'*ancestor* delle viste v_x e v_y è la più piccola vista che contiene tutte le informazioni necessarie per rispondere a v_x e v_y. Lo pseudocodice per determinarlo è il seguente:
A_z unione degli attributi
$\forall$ dipendenza funzionale $A_i^l \rightarrow A_i^r$
$\forall a_j \in A_i^r$
$if(a_j \cup a_i^l) \subseteq A_z$
$A_z -= a_j$

Altri metodi sono fondati su un approccio *workload-driven*. Viene analizzato il carico di lavoro del database partendo dall'idea che le *query* future non si discosteranno di molto dal *workload* passato. Un esempio è [ACN00], nel quale gli autori riportano il loro lavoro di ricerca (la cui implementazione è parte del *tuning wizard* di Microsoft SQL Server 2000) sulla selezione delle viste da materializzare, articolando il metodo in tre step: si sceglie un sottoinsieme di tabelle tra quelle contenute nel carico di lavoro tali che, se vengono create delle viste per queste tabelle, si riduce in modo significativo il costo del *workload*; quindi per ogni *query* viene proposto un insieme di viste che la risolvono e, tra queste, viene scelta la migliore; infine, queste viste vengono fuse in modo da riuscire a risolvere più *query* a partire dallo stesso aggregato.
Gli ultimi due approcci che vengono mostrati sono il *clustering-based* e quello evolutivo.
[AJD06] utilizza come input una matrice $n \times m$ dove n è il numero di dimensioni (attributi delle *dimension table*) che si ritrovano nel *workload* e m è il numero di *query* analizzate: la cella q_{ij} vale 1 se l'attributo i è presente nella *query* j, 0 altrimenti.

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
q_1	1	1	1	0	0	0	0	0	0	0	0	0
q_2	0	0	0	1	0	1	1	1	1	0	0	0
q_3	0	0	0	1	0	1	1	1	1	1	1	1
..												
..												

a_1 times.time_id — a_2 times.fiscal_day
a_3 sales.time_id — a_4 products.prod_id
a_5 products.prod_category — a_6 sales.promo_id
a_7 promotions.promo_id — a_8 sales.prod_id
a_9 promotions.promo_category — a_{10} sales.cust_id
a_{11} customers.cust_marital_status — a_{12} customers.cust_id

Figura 2.4: Input dell'algoritmo *clustering-based*

Partendo da un *clustering-context* simile a quello in figura (fig 2.4), si cerca, attraverso tecniche di classificazione non supervisionata, di creare dei *cluster* di *query*. Una *query* fa parte di un gruppo se è alta la similarità con le altre *query* della stessa classe e alta la dissimilarità con *query* degli altri *cluster*. Se venissero considerate tutte le viste così ottenute, il numero di materializzazioni sarebbe molto elevato, soprattutto se il *workload* contie-

ne molte *query*. Per questo motivo l'articolo riporta anche un processo di *merging* basato su due concetti chiave: la vista ottenuta dalla fusione deve risolvere tutte le *query* coperte da ogni vista che partecipa al *merging* e il costo dell'uso delle viste singole non deve essere troppo maggiore al costo dell'uso della vista fusa.
[ZYY01] e [ZY99] utilizzano algoritmi genetici per la risoluzione del problema. Viene generata la soluzione iniziale partendo da un insieme di piani d'accesso (ottenuti analizzando, in algebra relazionale, le *query* del *workload*) che vengono quindi rappresentati come grafi aciclici direzionati (DAG); tramite algoritmi di visita in ampiezza o in profondità, il DAG viene rimodellato come una lista ordinata di elementi. Infine la lista viene trasformata in una stringa binaria, dove lo 0 in posizione k indica che il nodo k non è materializzato e 1 altrimenti. Partendo da questa popolazione, si applicano gli operatori caratteristici degli algoritmi genetici, ovvero il *crossover*, la mutazione e la selezione, considerando una funzione di *fitness* che definisce quanto buona è una soluzione.
L'ultima analisi riguarda la funzione di costo utilizzata per determinare il grado di bontà di una vista. La creazione di un modello matematico contrasta con approcci che invocano l'ottimizzatore di *query* e utilizzano statistiche mantenute dalle tabelle di sistema. Le diverse possibilità sono determinate dai vincoli posti dal sistema stesso, come il tempo impiegato per la materializzazione (es. [Bad11]), lo spazio occupato, il tempo di risposta delle *query* OLAP o un insieme dei precedenti.
Un intero *framework* per stimare la cardinalità delle viste è mostrato in [CGR01]. Di seguito (fig 2.5) viene riportata la rappresentazione grafica dei moduli che compongono l'idea.
Quando un algoritmo per la materializzazione delle viste richiede informazioni sulla cardinalità del candidato, il *bounder* determina un limite minimo e uno massimo per le cardinalità di un insieme di viste, utilizzando dei vincoli forniti dall'utente. L'*estimator*, grazie a questi *bound*, propone una stima probabilistica sulla cardinalità del candidato. I vincoli (detti *cardinality*

contraint) sono di due tipologie:

- *upper* e *lower bound* della cardinalità di una vista
- la *k-dependency* ($X \xrightarrow{K} Y$) di due viste X e Y (k è un limite superiore del numero di tuple distinte di Y che corrispondono ad ogni tupla distinta di X)

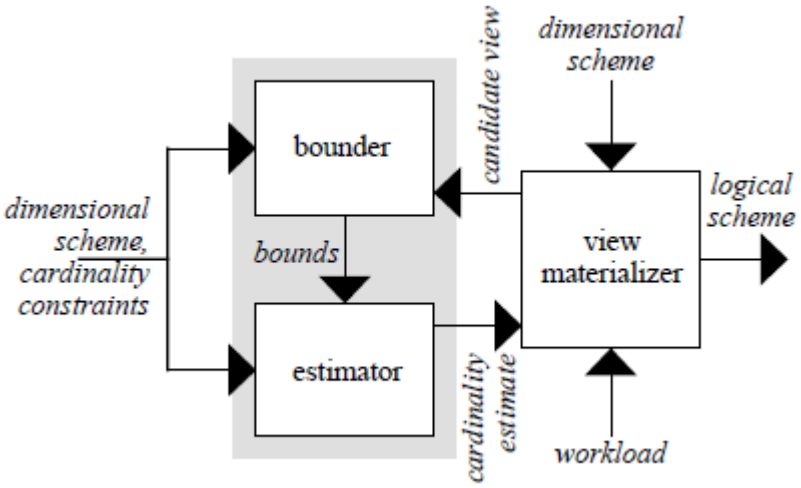


Figura 2.5: Architettura del modello di stima delle cardinalità

Capitolo 3

Formalizzazione del problema e vincoli

L'obiettivo principale del progetto di tesi è quello di creare un sistema semiautomatico per il *tuning* di un *data warehouse*, focalizzato sulla proposta delle *migliori* viste che permettono di ridurre il tempo di esecuzione delle *query* di un *workload* di input. La scelta dell'algoritmo e delle tecniche utilizzate dipendono fortemente dalla formalizzazione del problema e dall'analisi dei vincoli che si hanno sull'ambiente valutato. Di seguito vengono riportati i componenti dell'architettura funzionale su cui si basa il progetto.

Rappresentazione del carico di lavoro

Il carico di lavoro (o *workload*) rappresenta l'insieme di *query* che corrispondono alle analisi di *business* eseguite più frequentemente sul *data warehouse*. Il modello teorico delle *query* è basato su espressioni GPSJ (*Generalized Projection-Selection-Join*) del tipo $\pi_{GB,M}\ \sigma_P\ \chi$, dove χ rappresenta il *join* tra la *fact table* e le *dimension table*, σ_P indica la selezione dell'insieme di predicati P fatti sui vari attributi delle dimensioni e $\pi_{GB,M}$ è la proiezione di GB e M (GB rappresenta l'insieme di attributi del *Group-By Set* e M le misure aggregate al livello di dettaglio di GB).
Partendo quindi da una *query* relazionale come la seguente

Listing 3.1: Modello di query SQL

```
SELECT F.M, D1.X, D2.Y -- proiezioni
FROM F, D1, D2
WHERE F.ID1 = D1.ID AND F.ID2 = D2.ID -- join
AND D1.FIL = 'FILTER' -- filtri
GROUP BY D1.X, D2.Y
```

si estraggono le informazioni necessarie e si rimodella il problema attraverso gli operatori di algebra relazione come segue

$$\pi_{D1.X\ AND\ D2.Y\ AND\ F.M}\ \sigma_{filtri}\ \chi_{join}$$

A livello pratico i dati di *input*, come mostrati sopra (*listing* 3.1), sono facilmente ottenibili da qualsiasi *vendor* venga utilizzato. Tutte le applicazioni di *front end* di *business intelligence* in tecnologie ROLAP, come Oracle Business Intelligence o Microstrategy, eseguono delle *query* sul *data warehouse*, che ne memorizza tutte le informazioni necessarie per la riscrittura del *workload* secondo il modello appena descritto.
La considerazione di [ACN00], ovvero che il carico di lavoro futuro non si discosterà troppo da quello attuale, è assunta valida anche in questa tesi. L'analisi deve quindi concentrarsi sulle *query* più lente tra quelle maggiormente eseguite dagli utenti.
Infine occorre fare una precisione riguardo gli elementi del modello del carico di lavoro utilizzabili; se una vista secondaria rappresenta una vista primaria già aggregata a un certo livello di dettaglio, le informazioni necessarie per la loro creazione sarebbero solo i livelli di aggregazione di ciascuna dimensione e il fatto analizzato. Tuttavia, un ulteriore requisito del prodotto deve essere quello di far coincidere un processo di materializzazione delle viste aggregate con uno di frammentazione verticale. Secondo [GR06] il partizionamento è una tecnica di miglioramento delle prestazioni di un sistema (l'approccio nasce dal mondo relazionale/transazionale) che suddivide una tabella iniziale in più frammenti; la frammentazione verticale si distingue da quella oriz-

zontale se la tabella viene suddivisa rispettivamente a livello di attributi o a livello di tuple. Normalmente in approcci tradizionali un aggregato viene materializzato con tutti gli attributi del fatto originale: in realtà spesso molte misure hanno un valore concettuale solo se riportate a specifici livelli di aggregazione e perdono il loro significato se analizzate ad altri. L'idea è quindi quella di creare aggregati con tutti e soli gli attributi utili al carico di lavoro valutato, rimuovendo dalla tabella originale le misure non analizzate e i riferimenti a *dimension table* mai utilizzati. A livello teorico si perde la generalità dell'aggregato (se il carico di lavoro futuro conterrà *query* che richiamano, allo stesso livello di aggregazione della vista, misure non presenti, non sarà possibile risolverle attraverso l'aggregato in questione), ma si ottiene un duplice vantaggio: da un lato il risparmio di spazio che con buona probabilità sarà occupato inutilmente, almeno finchè il *workload* principale rimane quello analizzato, dall'altro il miglioramento aggiuntivo sul tempo di risposta del carico di lavoro.

Avere un'applicabilità pratica su sistemi esistenti

L'idea nasce in un ambito reale e questo aggiunge al problema dei vincoli pratici che in un ambito di ricerca sono trascurabili.
In primis occorre basarsi su modelli che possono avere un alto grado di diversità tra loro. A livello progettuale diverse scelte possono essere fatte per rappresentare la stessa idea concettuale (banalmente anche tra *star schema* e *snowflake schema*) e non è possibile incentrare l'analisi su un'unica tecnica di sviluppo. Sempre con riferimento a questo problema, occorre darsi come ipotesi che il *data warehouse*, già al momento di creazione (ma spesso anche per interventi di manutenzione), può non seguire gli standard di *best-practice* proposti a livello teorico. Di seguito verrà riportato il modello che si assume valido per la rappresentazione del *data warehouse*.
Il secondo problema, che a livello teorico è rappresentato dal costo computazionale dell'algoritmo utilizzato e a livello pratico si ripercuote sul tempo di esecuzione, è quello delle prestazioni del sistema. I *target* del prodotto sono

data mart di grandi dimensioni, difficilmente analizzabili in modo non automatico: più in generale l'obiettivo è quindi quello di migliorare il processo di *tuning*, fornendo dei risultati sub-ottimi con tempistiche ragionevoli.

Precisione delle stime

Come mostrato nel capitolo precedente, una possibile scelta riguarda il metodo di valutazione degli aggregati candidati: imponendo dei vincoli di tempo e spazio o minimizzando una funzione di costo che fornisce una stima della bontà della vista in questione. Indipendentemente dalla scelta, si vuole che le stime forniscano dei risultati attendibili e precisi.
Il problema principale della valutazione degli aggregati sta nel numero di dipendenze funzionali che si hanno tra i vari attributi. Dato un caso come quello in figura 2.3, dove ogni dimensione ha al più un figlio, il numero di possibili viste sono come nell'equazione 2.1.

Se si hanno invece n dimensioni senza gerarchia, il numero delle viste sale a 2^n.

Nel caso più generico il numero di viste si riduce grazie ai livelli gerarchici di ciascuna dimensione, ma rimane comunque un numero molto elevato.
Algoritmi già diffusi in ricerca prevedono di analizzare tutte le dipendenze funzionali tra gli attributi di ciascuna dimensione e, per ogni coppia, valutare il miglioramento che si otterrebbe materializzando il *parent* rispetto al *child* (considerando la dipendenza funzionale come *parent* $\rightarrow$ *child*).

Formalizzazione del problema

Come specificato dal capitolo 1, molte sono le possibili formalizzazioni del problema. La scelta fatta è sufficientemente generica per poter essere applicata a un contesto reale, partendo dai presupposti mostrati nella sezione sull'applicabilità in campo industriale.
Per prima cosa viene definito il modello del *data warehouse* considerato:

- lo schema deve contenere una o più tabelle identificate come fatti e una o più tabelle identificate come dimensioni

- l'architettura logica del DWH deve essere generica, ovvero la risoluzione del problema deve essere garantita sia in presenza di uno o più *data mart*, che in presenza di un *enterprise data warehouse*, così come se il DWH è organizzato secondo approcci in stile *snowflaking* o *star schema*

- deve essere possibile inferire in modo automatico la tipologia di una tabella

Se i precedenti vincoli sono soddisfatti, è possibile formalizzarvi sopra il problema nel seguente modo:
dato uno schema di fatto F *e un carico di lavoro* Q *su* F*, il problema è quello di determinare l'insieme delle viste* V *su* F *che minimizzano il costo di* Q *espresso da una funzione f(Q,V).*
Il problema iniziale si divide quindi in due fasi:

- identificazione dell'insieme C di viste candidate

- scelta del sottoinsieme V $\subseteq$ C che minimizza il costo di Q

Secondo [BPT97], una vista v_c è candidata se risponde direttamente a un'interrogazione q $\in$ Q oppure se esistono due viste candidate v_1 e v_2 di cui v_c è *least upper bound* (anche detto *ancestor*, la vista più specializzata che può rispondere ad entrambe). Gli stessi autori dimostrano che, se la funzione di costo cresce in modo monotono con la dimensione delle viste, la materializzazione di una vista non candidata porta sempre a un costo di esecuzione maggiore rispetto a quello ottenuto dalla materializzazione di una vista candidata.

Capitolo 4

AutoTuning

In questo capitolo verrà mostrata l'architettura logica del sistema implementato specificando come viene gestito il *workflow* dai vari componenti. Si entra quindi più nel dettaglio dei singoli moduli, mostrando le logiche che li governano. Conclude il capitolo una trattazione teorica dell'algoritmo, confrontato con quelli già studiati e applicati in campo accademico.

4.1 Architettura logica e *workflow*

Il sistema si sviluppa secondo un'architettura a tre livelli, gestendo quindi uno strato fisico, uno logico e uno di presentazione. Il primo ha la duplice funzione di collegarsi al *data warehouse* di analisi e quella di memorizzare in maniera persistente i dati ottenuti dagli algoritmi. Il *presentation layer* contiene l'interfaccia utente e rappresenta il componente *View* se riferito al *design pattern Model-View-Controller*. La logica funzionale è contenuta nello strato intermedio, che a sua volta si divide nei tre moduli principali che specificano l'architettura del sistema: *Extractor*, *Parser* ed *Engine*.
La caratteristica principale del sistema è che basa le proprie analisi su informazioni ottenute da *data warehouse* già creati e funzionanti. Questo particolare fa si chè funzioni sia su un'architettura di tipo *snowflake* che su uno *star schema*, che non sia legato a una specifica tecnologia e, utilizzando le

statistiche memorizzate dal DBMS, che analizzi il carico di lavoro effettivo non basandosi su stime approssimative. L'unico requisito obbligatorio è che il *data warehouse* si basi su un modello logico ROLAP, nel quale ogni analisi è riconducibile a una *query* fisica effettuata e memorizzata direttamente sulla base di dati, evitando motori multidimensionali che si collegano al mondo relazione solo al momento del caricamento dei dati nel cubo.

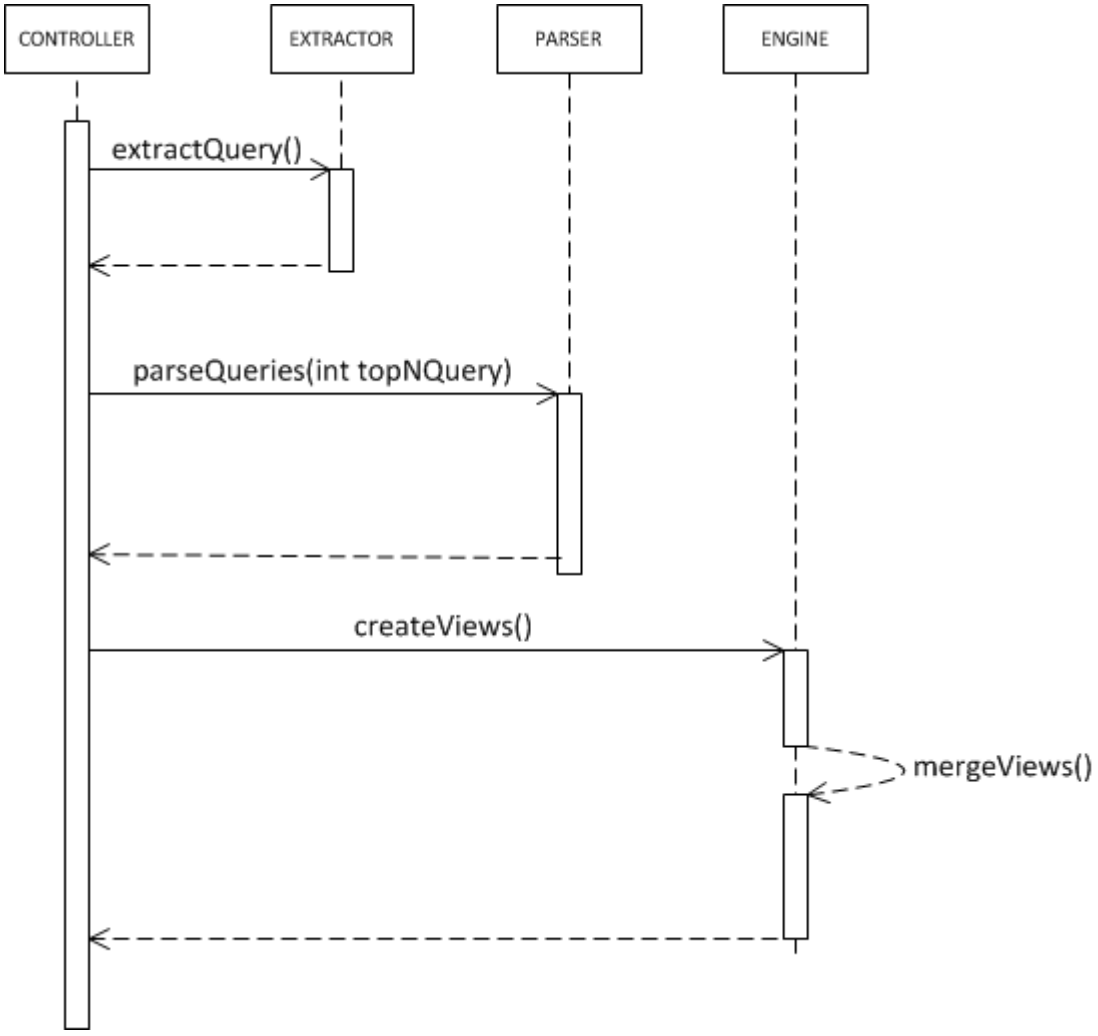


Figura 4.1: Diagramma di sequenza dello strumento

Nella figura 4.1 viene presentato il flusso standard di esecuzione dello strumento. Molte impostazioni sono opzionali ma, ad alto livello, gli *step* possono essere rappresentati come in figura. L'estrattore ottiene dal *data warehouse* le informazioni sulle *query* maggiormente "lanciate" dagli utenti di *business*, quindi filtrate sull'applicativo di *front end* utilizzato, che abbiano un tempo di esecuzione alto[1]. Queste informazioni, *in primis* il testo completo della *query*, vengono utilizzate dal *Parser* che, partendo dall'SQL analizzato, riscrive il problema secondo il modello formale descritto nel capitolo precedente. Per ciascuna *query* vengono quindi memorizzate le tabelle coinvolte, distinte tra *Fact Table* e *Dimension Table*, i *join* e il livello di aggregazione richiesto. Per ultimo il flusso passa al modulo *Engine*, che ha il compito di ricercare l'insieme di aggregati migliore per aumentare le *performance* rispetto allo specifico carico di lavoro. Come specificato più volte, l'assunzione che viene fatta è la medesima rispetto a [ACN00] ovvero che il carico di lavoro non subisce nel tempo modifiche sostanziali. Questa ipotesi può essere ritenuta valida perché le analisi sui dati aggregati realisticamente non sono soggette a cambiamenti frequenti proprio per la loro natura analitica (es. se il valutare l'analisi delle vendite suddivise per regione geografica risulta un report utile in un mese, probabilmente lo sarà anche il successivo).

Di seguito vengono dettagliati i tre moduli centrali della logica dell'applicativo, mostrandone le caratteristiche tecniche e concettuali più importanti.

[1]Per tempo di esecuzione alto si intende che superi una certa soglia definita dall'utente.

4.2 *Extractor*

L'estrattore è il modulo che regola l'*import* dei metadati dal *data warehouse*. I metadati utili allo strumento modellano il *workload* e le gerarchie logiche delle dimensioni.
Nel capitolo precedente, riguardante la formalizzazione del problema, è stata definita una struttura del carico di lavoro a livello teorico. Scendendo nel pratico, lo strumento utilizza una tabella relazionale per la rappresentazione del *workload*, che segue una struttura come nell'esempio.

ID	**SQL FULLTEXT**	**TOT TIME**	**NUM EXE**	**TIME SINGLE EXE**
1	SELECT ... FROM ... GROUP BY ...	100	4	25
2	SELECT ... FROM ... GROUP BY ...	250	50	5
...	...	...	...	...

Il SQL_ID è sicuramente univoco perché l'estrattore utilizza una tecnica di *update* del tipo "*truncate-insert*" quindi conterrà solo e soltanto le *query* ottenute da uno *snapshot* del *workload* totale di uno specifico *database*. Ad ogni SQL_ID è associato il testo completo della *query*, il tempo di esecuzione totale e il numero di esecuzioni. Un attributo derivato è il tempo di esecuzione medio della singola esecuzione, calcolato come rapporto dei due attributi precedenti; la sua utilità si limita al miglioramento delle prestazioni nell'esecuzione degli algoritmi seguenti.
Queste informazioni sono memorizzate tramite metodi diversi da tutti i principali DBMS. In Oracle è presente un componente chiamato *Automatic Workload Repository* (AWR) che colleziona, processa e tiene aggiornate le statistiche sulle *performance*. Tra le varie informazioni storicizzate vi sono le statistiche sugli oggetti acceduti ed utilizzati, i modelli statistici basati sul

tempo e gli *statement* SQL che producono il carico maggiore sul sistema, calcolato in base a criteri come tempo di esecuzione, utilizzo della CPU e numero di operazioni di I/O. L'AWR aggiorna di *default* le proprie viste una volta ogni ora e le mantiene per 8 giorni. Le *snapshot*, ovvero l'insieme di dati storici per specifici periodi, possono essere create manualmente oppure è possibile modificare le impostazioni di base per ridurre il tempo tra due consecutive (attraverso il parametro SNAP_INTERVAL), aumentare il tempo di memorizzazione dei dati (con il parametro RETENTION) e il numero di *query* raccolte ad ogni "fotografia" (parametro TOPNSQL). L'esempio seguente mostra la modifica di questi parametri in modo da ottenere un intervallo tra due *snapshot* di un'ora, un tempo di conservazione di 30 giorni e un numero di *query* memorizzate pari al massimo impostabile.

Listing 4.1: Script Oracle PL/SQL per la modifica dei parametri delle *snapshot*

```
execute
  dbms_workload_repository.modify_snapshot_settings(
                dbid => xxxxxxxxxx,
                interval => 60,
                retention => 11520,
                topnsql => 'MAXIMUM');
```

Tra le varie viste aggiornate, quelle che interessano l'estrattore sono le seguenti:

- DBA_HIST_SQLSTAT: mostra le informazioni storiche sulle statistiche SQL
- DBA_HIST_SNAPSHOT: mostra le informazioni sulle *snapshot*
- DBA_HIST_SQLTEXT: associa a ciascun identificatore SQL il corrispondente testo completo

L'unione di queste tre viste di sistema permette all'estrattore di ottenere tutte le informazioni per la memorizzazione del *workload* secondo il proprio standard, filtrando tra le *query* quelle lanciate dal sistema di *front-end* di BI utilizzato.
Microsoft gestisce in maniera analoga la memorizzazione di queste informazioni: la funzione *sys.dm_exec_sql_text* prende in *input* un identificatore della specifica query o dello specifico piano d'accesso e ritorna una tabella che associa all'id, il testo completo dell'SQL; la vista *sys.dm_exec_query_stats* associa al *token* corrispondente alla query le proprie statistiche. É possibile ottenere il numero di esecuzioni di un piano d'accesso, il tempo di CPU in microsecondi, il numero di letture e scritture, il tempo di esecuzione e molte altre informazioni.

Il secondo compito dell'*extractor* è quello di ottenere le relazioni *parent-child* tra le dimensioni e memorizzarle secondo lo schema esemplificato dalla tabella seguente:

LOOKUP TABLE	**LOOKUP TABLE PK**	**PARENT ID**	**LEVEL ID**	**LEVEL DESC**	**LEVEL NUM**
CALENDAR	DATE_ID		DATE_ID	DATE_DE	0
CALENDAR	DATE_ID	DATE_ID	MONTH_ID	MONTH_DE	1
CALENDAR	DATE_ID	MONTH_ID	SEM_ID	SEM_DE	2
CALENDAR	DATE_ID	SEM_ID	YEAR_ID	YEAR_DE	3

Questa rappresentazione corrisponde esattamente al DFM in figura 4.2.

La tabella rappresenta una modalità comoda per la memorizzazione dello schema di un *data warehouse*, ma non è normalizzata secondo le regole classiche dei *database* relazionali. Tutti i campi, eccezion fatta per il numero del livello, sono testuali. La chiave logica è rappresentata dall'insieme di

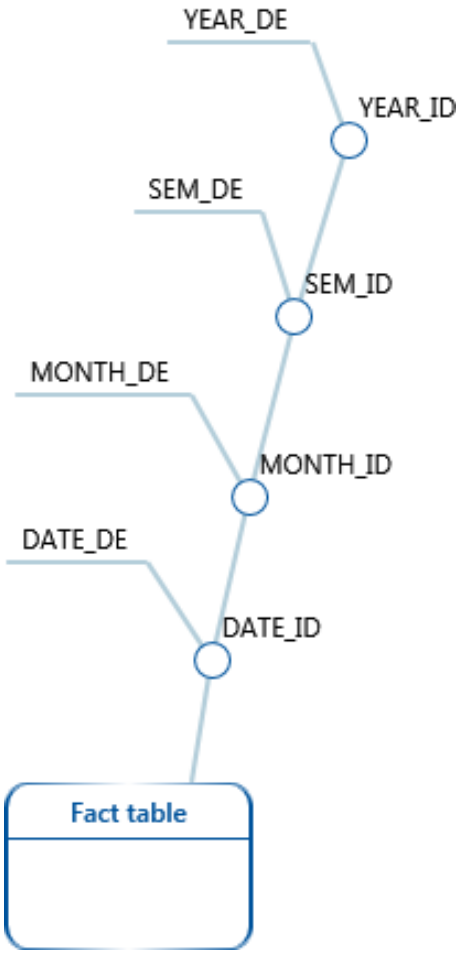


Figura 4.2: DFM di esempio raffigurante la dimensione temporale

attributi {LOOKUP_TABLE, LEVEL_ID, LEVEL_DESC}, in modo da differenziare per ciascuna dimensione di ogni *lookup table* i possibili attributi descrittivi associati. Anche in questo caso i duplicati sono evitati grazie al metodo di *update* che prevede una cancellazione totale dei dati presenti nella tabella prima del successivo inserimento, in modo da non avere contemporaneamente informazioni di schemi diversi.
La modalità più semplice per ottenere questi alberi è utilizzare le strutture gerarchiche proposte dai vari DBMS: Oracle permette la creazione delle *Hierarchy*, ovvero dei *path* di *drill* definiti tra gli oggetti, mentre SQL Server si

appoggia agli *hierarchyid*, un tipo nativo che rende semplice memorizzare e interrogare dati rappresentati come alberi. Questi metodi sono però legati ad una gestione specifica del *data warehouse* a livello fisico, mentre le ipotesi del problema prevedevano una generalizzazione totale per quanto riguarda la modellazione logica e l'implementazione fisica dei dati.
La procedura di *profiling* utilizzata basa quindi la sua analisi e il suo risultato sui dati contenuti all'interno delle *dimension table*, evitando assunzioni che dipendono dalle scelte progettuali. L'idea alla base dell'algoritmo, indipendente dallo specifico dialetto SQL, è verificare il numero di valori distinti di un attributo fissandone un secondo.

Listing 4.2: Pseudocodice della procedura di *profiling* delle gerarchie delle dimensioni

```
1  Per ogni dimension_table
2    Creo tutte le coppie ordinate di attributi (a, b)
3                  t.c. distinct(a) < distinct(b)
4    Per ogni coppia (a, b)
5      SELECT COUNT(DISTINCT b)  INTO counting
6      FROM DT
7      GROUP BY a
8      HAVING COUNT(DISTINCT b) > 1
9      Se counting is null
10        FD = FD U {a -> b}
11   FD = pruning_graph(FD)
```

Per ogni tabella non *fact*, vengono create tutte le coppie ordinate di attributi che potenzialmente rappresentano una dipendenza funzionale. Dato che il numero totale di valutazioni per una *dimension table* sarebbero, secondo il calcolo combinatorio, le disposizioni di classe 2 degli n attributi della *lookup table* (D(n,k)), quindi $\frac{n!}{(n-2)!} = n! \cdot (n-1)!$, queste vengono filtrate solo su quelle effettivamente potenziali, quindi dove il primo attributo ha un

numero di valori distinti maggiore rispetto al secondo. Per ciascuna coppia si esegue la query che rappresenta la vera logica del *profiler*, che seleziona i valori distinti del secondo attributo avendo raggruppato per il primo. Un esempio può essere fatto con gli attributi MONTH e YEAR di un'ipotetica dimensione temporale:

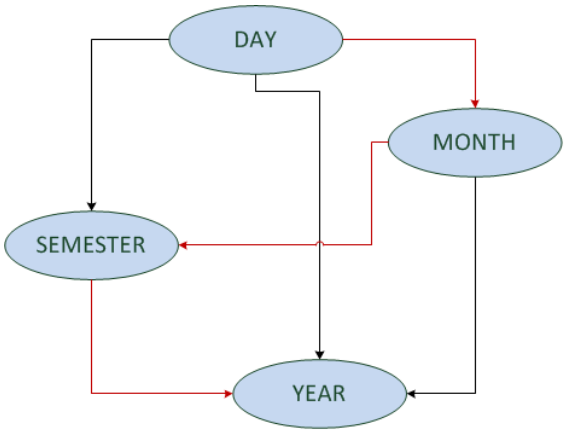
DAY	MONTH	...	YEAR
01/12/2013	12/2013	...	2013
01/01/2014	01/2014	...	2014
01/02/2014	02/2014	...	2014
02/02/2014	02/2014	...	2014

La query

```
SELECT COUNT(DISTINCT YEAR)
FROM TIME
GROUP BY MONTH
```

equivale a chiedersi quanti valori distinti dell'anno ci sono per ogni mese; fissando un qualsiasi MONTH, indipendentemente dal numero di volte che compare in tabella, ne corrisponde un unico valore di YEAR. Questo afferma che YEAR dipende funzionalmente da MONTH, quindi l'anno è figlio del mese in una struttura gerarchica ad albero. Per questioni implementative è stata aggiunta la clausola HAVING > 1 al conteggio dei valori distinti: se il primo attributo implica il secondo il risultato sarebbero tanti 1 quanti sono i valori distinti del primo; aggiungendogli il filtro, vengono scartati i risultati uguali a 1 e la dipendenza funzionale è presente se l'*output* complessivo della *query* è vuoto.

Fatta questa operazione per ogni coppia, ciò che si ottiene è un "sovrainsieme" di dipendenze funzionali e il prossimo passo è effettuare il *pruning* del grafo ottenuto (ovvero la normalizzazione o riduzione transitiva). Sempre riferendosi all'esempio precedente, il risultato che si otterrebbe dalla prima parte dell'algoritmo sarebbe il seguente grafo orientato aciclico (DAG):



X	Y
DAY	MONTH
DAY	SEMESTER
DAY	YEAR
MONTH	SEMESTER
MONTH	YEAR
SEMESTER	YEAR

Figura 4.3: Grafo orientato ottenuto dalla prima fase del *profiler*, mostrato in formato visuale e tabellare

In rosso sono riportate sia a livello di grafo che a rappresentazione tabellare le dipendenze funzionali che si vuole ottenere alla fine dell'algoritmo; le altre sono dipendenze transitive, quindi concettualmente corrette ma non utili per lo scopo.
La fase di *pruning* segue un processo ricorsivo basato su livelli: l'assegnazione dei livelli può essere fatta con l'SQL standard come nell'esempio seguente, anche se alcuni *vendor* propongono soluzioni specifiche per questi tipi di problemi, come la clausola CONNECT BY PRIOR di Oracle.

```
WITH n(parent, child, level) AS
(
    SELECT parent, child, 0 level
    FROM tmp_metadata
    WHERE parent is null
        UNION ALL
    SELECT parent, child, n.level + 1
    FROM tmp_metadata as m, n
    WHERE n.child = m.parent
)
SELECT DISTINCT * FROM n ORDER BY level
```

La chiave della tabella (di *default* l'attributo con il numero maggiore di dipendenze associate) non comparirà mai come colonna dipendente, quindi le tuple che rappresentano le dipendenze funzionali con la chiave come *left-hand side* (nell'esempio quelle che hanno DAY nel ruolo di X) vengono etichettate come di livello 1. Ricorsivamente si assegna il livello 2 a tutti i record che hanno come attributo X gli attributi Y di livello 1 e, negli *step* successivi, si aggiorna il numero[2] con il massimo livello + 1. Questo processo aggiorna la tabella precedente con il livello corrispondente a ciascuna dipendenza:

X	**Y**	**LEVEL**
DAY	MONTH	1
DAY	SEMESTER	1
DAY	YEAR	1
MONTH	SEMESTER	2
MONTH	YEAR	2
SEMESTER	YEAR	3

Per ogni valore nell'attributo Y si seleziona il *record* di livello massimo eliminando tutti quelli con livello minore (ad esempio per Y = YEAR il livello massimo è 3 ed è determinato dalla dipendenza SEMESTER → YEAR; DAY → YEAR e MONTH → YEAR sono dipendenze di livello rispettivamente 1 e 2 e vengono quindi eliminate dalla soluzione finale).

La problematica maggiore di questo algoritmo è il tempo di esecuzione, rallentato dall'analisi di molte coppie di attributi che non finiscono poi in soluzione. Per questo motivo è stata creata un'euristica *ad hoc* che si basa sulle stesse considerazioni, evitando però operazioni di I/O inutili dovute alla valutazione di dipendenze funzionali transitive già conosciute. Di seguito lo pseudocodice dell'euristica:

[2]Per esempio al secondo *step*, SEMESTER → YEAR avrà livello 2, mentre al terzo passaggio viene aggiornato a 3 dato che LEVEL{MONTH → SEMESTER} > LEVEL {DAY → SEMESTER}

Listing 4.3: Pseudocodice della procedura di *profiling* delle gerarchie delle dimensioni

```
foreach dimension_table
  M = create_matrix(A)
  while (M is not full)
    next = choose_next(M)
    if (is_functional_dependency(next))
      update(M, next.dominated, next.dominant)
  FD = FD + get_functional_dependencies(M)
```

Il primo *step* è quello di creare la matrice che rappresenta la struttura dati di base: gli attributi della *dimension_table* analizzata vengono ordinati in modo discendente rispetto al numero di valori distinti. La matrice mostra a livello di righe gli attributi dominanti (*left-hand side* della dipendenza) e in colonna i dominati (che corrispondono ai *right-hand side*). La matrice è triangolare superiore con diagonale nulla, dato che l'attributo i non dominerà mai l'attributo j se il numero di valori distinti di i è minore rispetto a quello di j. Le celle della matrice possono assumere quattro valori distinti: il valore nullo, al momento della creazione, viene attribuito alla diagonale e alla porzione inferiore; la cella (i,j) conterrà come valore informativo "SÍ" se i domina j, "✓" se la dipendenza è transitiva e "NO" altrimenti.
La successiva cella valutata (*next*) è quella che corrisponde all'attributo dominato con cardinalità maggiore non calcolato sul dominante con cardinalità minore. Se il *next* ($i \rightarrow j$) è una dipendenza funzionale, quindi la cella (i,j) assume valore "SÍ", occorre aggiornare la matrice inserendo le dipendenze transitive rispetto al dominato j; perciò j è dominato, oltre che direttamente da i, in modo transitivo da tutti i dominanti di i (siano essi corrispondenti a dipendenze funzionali dirette che a dipendenze transitive).
Di seguito viene presentato l'esempio di esecuzione dell'euristica nei tre casi di base di modellazione del DFM (l'analisi sui costrutti avanzati è riportata dopo gli esempi): all'interno di ogni cella un numero indica la sequenza di valutazione.

	A	B	C	D
A	-	SÍ[1]	✓[3]	✓[5]
B	-	-	SÍ[2]	✓[5]
C	-	-	-	SÍ[4]
D	-	-	-	-

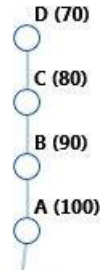


Figura 4.4: Caso migliore: le valutazioni sono 3, esattamente quante sono le dipendenze funzionali

	A	C	D	B
A	-	SÍ[1]	SÍ[3]	SÍ[6]
C	-	-	NO[2]	NO[5]
D	-	-	-	NO[4]
B	-	-	-	-

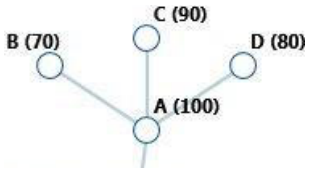


Figura 4.5: Caso peggiore: nel caso non ci siano gerarchie, tutte le possibilità vengono valutate

	A	B	C	E	F	D	G	H
A	-	SÍ[1]	✓[3]	✓[6]	✓[8]	✓[13]	✓[15]	✓[19]
B	-	-	SÍ[2]	SÍ[5]	✓[8]	✓[13]	✓[15]	✓[19]
C	-	-	-	NO[4]	NO[9]	SÍ[12]	✓[15]	✓[19]
E	-	-	-	-	SÍ[7]	NO[11]	✓[17]	✓[19]
F	-	-	-	-	-	NO[10]	SÍ[16]	✓[19]
D	-	-	-	-	-	-	SÍ[14]	✓[19]
G	-	-	-	-	-	-	-	SÍ[18]
H	-	-	-	-	-	-	-	-

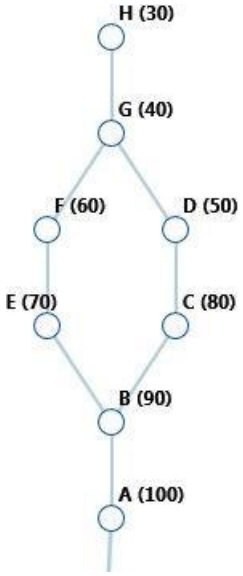


Figura 4.6: La dimensione G rappresenta una condivisione/convergenza che viene comunque individuata dall'euristica

Rifacendosi al modello DFM (*Dimensional Fact Model*), la maggior parte dei costrutti viene identificata:

- gli attributi descrittivi vengono rappresentati come figli degli identificatori, a meno che il valore di descrizione sia anch'esso distinto (relazione 1 a 1 con l'id) nel qual caso i due vengono messi in gerarchia anche rispetto agli altri attributi; anche se questa assunzione modella indistintamente surrogati e descrittivi, a livello teorico non è concettualmente errata

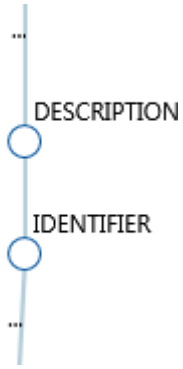


Figura 4.7: Esempio della gestione dei descrittivi da parte del *profiler*

- l'algoritmo non distingue tra archi opzionali e non; vengono quindi considerati come normali dimensioni secondarie

- attributi *cross* dimensionali, così come tutti i casi in cui un attributo dipende funzionalmente da un insieme di altre colonne, non è possibile determinarli perché l'algoritmo gestisce solo le relazioni uno-a-molti tra coppie;

- analogo è il caso di archi multipli implementati con *bridge table*; al contrario una soluzione progettuale di tipo *push-down*, modellando l'arco multiplo direttamente nella fact table, permette alle dimensioni di essere scollegate tra loro

- gerarchie incomplete e ricorsive non aggiungono complessità a livello strutturale della tabella, quindi vengono gestite dalla procedura come gerarchie semplici

- infine, costrutti quali condivisione e convergenza vengono rappresentati; la distinzione tra i due modelli può essere ipotizzata solo paragonando il numero di valori distinti di un attributo passando per un ramo della gerarchia con lo stesso numero ottenuto passando dall'altro ramo (nella figura 4.6 G è una condivisione se il numero di valori distinti è diverso sia considerando la gerarchia F-G sia quella D-G)

4.3 *Parser*

In informatica, l'analisi sintattica, anche chiamata *parsing*, è il processo che analizza uno *stream* continuo in *input* in modo da determinare la sua struttura grammaticale grazie a una data grammatica formale[3]. L'analisi sintattica, ovvero il capire la struttura di una frase, è sempre preceduta da un'analisi lessicale (*scanning*), ovvero la suddivisione del testo in *token*.
Il *lexer* (o *scanner*) è l'analizzatore lessicale e prende in *input* una sequenza di caratteri (es. una frase), la partiziona in lessemi[4] che vengono poi classificati in *token*, ovvero l'*output* del *lexer*. La differenza tra lessemi e *token* è più facilmente comprensibile con un esempio sull'esecuzione di un *lexer* costruito per un compilatore di un linguaggio di programmazione.

```
if (x == y)
    z = 0;
```

LESSEMA	TOKEN
if	IF
(	LPAR
x	ID
==	EQUALS
...	...

Il lessema rappresenta quindi il valore del *token* e viene diviso dagli altri seguendo la *maximal match rule* ("partire" è un lessema e non viene scomposto nei due "parti" e "re"). Il *lexer* di un linguaggio solitamente non viene scritto a mano ma da un *lexer generator*. Si utilizza un approccio dichiarativo per descrivere ogni *token* come un automa a stati finiti (o tramite espressioni regolari), combinandoli poi insieme per ottenere l'automa finale.
Il *parsing* è il processo di analizzare una sequenza di *token* per determinare

[3]Una grammatica formale è un sistema di regole che specificano un insieme di sequenze finite di simboli contenuti in un alfabeto finito [Zav].

[4]Il lessema è la più piccola unità significativa del lessico.

la loro struttura grammaticale rispettando una data grammatica formale. Il *parser* è il componente che si occupa di effettuare questo processo, prendendo in *input* dallo *scanner* una sequenza di *token* e fornendo in *output* un *Abstract Syntax Tree*. Quest'ultimo è un albero che rappresenta la struttura sintattica astratta di una sequenza di *token*, ovvero non riporta tutti i dettagli della struttura reale, come le parentesi che possono essere omesse o uno *statement* CASE-WHEN-ELSE viene rappresentato come un unico blocco con due sottoalberi figli che descrivono le operazioni eseguite nel ramo THEN e nel ramo ELSE. L'AST viene costruito attraverso diverse tecniche di riduzione a partire da un *parse tree*, un albero rappresentante la struttura sintattica concreta della stringa in *input* in base alla grammatica libera dal contesto[5].
Analogamente, il *Parser* dello strumento è il modulo che prende in *input* una *query* SQL e la scompone ottenendo informazioni sugli oggetti utilizzati. Il componente memorizza per ciascuna *query* le tabelle coinvolte, distinguendole tra *fact table* e *dimension table* attraverso regole di *naming* definite dall'utente, le misure associate al tipo e al livello di aggregazione, i filtri e le colonne in *join*. In questo punto vengono popolate quasi tutte le tabelle del *database* dell'applicativo, riportate nel diagramma ER in figura 4.8.

TABLE e COLUMN sono le due entità centrali del diagramma e memorizzano rispettivamente la frequenza con cui compaiono le tabelle e le specifiche colonne. Questo valore è complessivo, dato che le due tabelle non vengono mai troncate. Al contrario, COLUMN_IN_QUERY e JOIN_IN_QUERY contengono i valori riferiti a una specifica esecuzione: la prima identifica, per ogni query, le colonne che compaiono distinguendole in varie tipologie (es. "SUM", "WHERE", "SELECT", "COUNT", etc); la seconda analizza i join, memorizzando le due colonne coinvolte e il tipo di join (es. "=", ">", "≥", etc). Le ultime due, STATS_JOIN e STATS_MEASURE, non vengono

[5]La CFG (*Context-Free Grammar*) è una grammatica formale in cui ogni regola sintattica è espressa sotto forma di derivazione di un simbolo a sinistra a partire da uno o più simboli a destra.

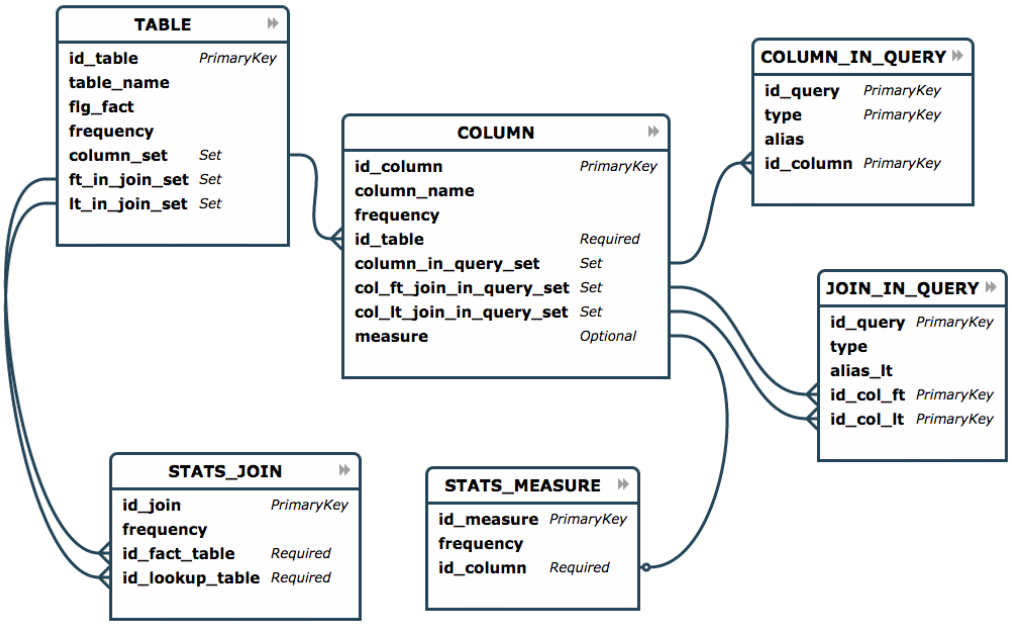


Figura 4.8: Diagramma ER della base dati alimentata dal modulo *Parser*

utilizzate dal successivo modulo per il calcolo degli aggregati, ma sono utili per monitorare lo stato e l'utilizzo del *data warehouse*, permettendo una più semplice scelta su quali elementi dismettere e quali svolgono ancora un ruolo fondamentale.

4.4 *Engine*

L'ultimo componente è quello che contiene la vera logica dello strumento: utilizza come *input* le tabelle alimentate dal *Parser* e i metadati ottenuti dall'*Extractor* e fornisce in *output*, per ogni *fact table* utilizzata dalle *query* analizzate, l'insieme di viste considerate migliori per risolvere il problema di *performance* basato sullo specifico *workload*.
L'algoritmo è diviso in due fasi: la creazione di un sottoinsieme del lattice multidimensionale e la scelta delle viste a costo minimo che coprono un *workload* prestabilito di *query*.
Ci si basa sulla considerazione di [BPT97] secondo la quale una vista v_c è candidata se risponde direttamente a una *query* oppure se esistono almeno due viste (v_1 e v_2) candidate di cui v_c è *least upper bound* (anche detto *ancestor*, la vista più specializzata che può rispondere ad entrambe).
In modo simile a [ACN00], si crea una vista per ogni *query* del *workload*; come l'algoritmo *query-specific-best-configuration* di [CN97], l'assunzione fatta è che se una soluzione non è ottima per una *query*, probabilisticamente non farà parte di quella ottimale per l'intero *workload*.
Nello pseudocodice sottostante viene formalizzata questa prima fase, ovvero la scelta dello spazio di ricerca da analizzare.

Listing 4.4: Fase 1 dell'algoritmo

```
1  for (i = 0; i < n; i++)
2      V_i = createView(q_i)
3  similarityMatrix = createSimilarityMatrix(V)
4  while (similaryMatrix is nullMatrix)
5      (V_x, V_y) = getMostSimilarViews(similarityMatrix)
6      V_i = mergeViews(V_x, V_y)
7      similarityMatrix = updateSimilarityMatrix(
8                              similarityMatrix, V, x, y)
9      V_i.setChildren(V_x, V_y)
10     i = i + 1
```

Nonostante vi sia un'associazione teorica uno-a-uno tra una vista e una *query*, definita secondo il modello del capitolo 3, alcuni accorgimenti vengono fatti per il *mapping*.
Il primo è rappresentato dalla caratteristica comune dei *data warehouse* di utilizzare surrogati associati a una o più descrizioni; solitamente ogni attributo descrittivo è legato ad un identificatore, utile per la gestione delle operazioni, mentre la descrizione viene utilizzata a livello di *report* e quindi per fornire il risultato all'utente. Le viste associate alle *query* vengono create sostituendo i descrittivi con i rispettivi identificatori, qualora l'associazione esista.
Il secondo punto riguarda il livello di dettaglio della vista: se sulla stessa dimensione due attributi con livello comune vengono inseriti in una *query*, la vista viene creata al minimo livello di dettaglio comune (un esempio in fig. 4.9).

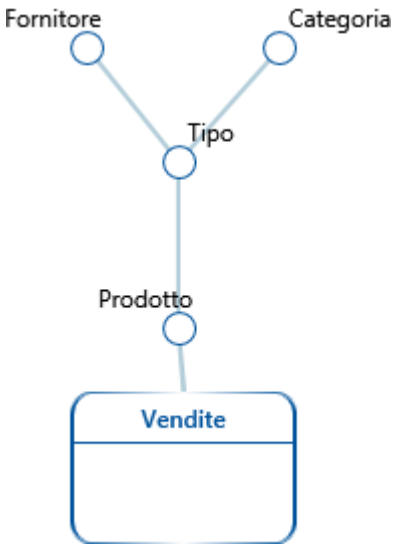


Figura 4.9: Se, nella stessa query, la dimensione Prodotto è presente a due livelli di dettaglio differenti, Fornitore e Categoria, al momento della creazione della vista l'*ancestor* Tipo viene scelto.

Partendo da queste viste di base, ne vengono iterativamente create di nuove fondendo le due maggiormente simili ancora non fuse. La funzione di similarità varia tra 0 a 1, rispettivamente da uno stato in cui le due viste non possono essere fuse a uno nel quale una delle due può essere completamente risolta dall'altra. Segue una definizione più formale della funzione:

$$\begin{cases} f_s \in [0,1] \\ f_s = 0 \; se \; F_1 \neq F_2 \\ f_s = max\left(k, N\frac{1}{|DT_1|}\right) \; se \; F_1 = F_2 \; e \; |DT_1| < |DT_2| \\ f_s = max\left(k, N\frac{1}{|DT_2|}\right) \; se \; F_1 = F_2 \; e \; |DT_1| \geq |DT_2| \end{cases}$$

L'idea della funzione è molto semplice: tante più sono le *dimension table* condivise dalle due viste, tanto più queste ultime sono simili. Una funzione di similarità tra due *query* può essere definita sia a livello di tabelle che a livello di colonne. La scelta è ricaduta sulle tabelle, considerato un *trade-off* tra tempo di calcolo della similarità e livello di attendibilità del valore ottenuto. Più nel dettaglio, dati F_1 e F_2 *fact table* rispettivamente della prima e della seconda *query*, DT_1 e DT_2 gli insiemi contenenti le *dimension table* sempre in relazione alla *query* di appartenenza e $N = |DT_1 \cap DT_2|$, la funzione di similarità è definita come il massimo tra $\frac{N}{|DT_x|}$ e k. k è un valore arbitrariamente piccolo, mentre DT_x è l'insieme di cardinalità minore tra DT_1 e DT_2. Se N è uguale alla cardinalità di DT_x, la similarità tra le due viste è massima dato che la vista con cardinalità maggiore contiene già tutte le tabelle utili a coprire l'altra. Se l'intersezione tra le tabelle si limita alla *fact table* comune, N vale 0 e la similarità è data dal valore k, che indica che le due viste sono potenzialmente fondibili ma non contengono alcuna DT comune.

La funzione di similarità è utile per popolare un'omonima matrice nxn, dove n è il numero delle *query* del *workload*. Il contenuto delle celle della *similarity matrix* riportano il valore di similarità tra le due viste. Una volta fuse le due localmente più simili, la matrice deve essere aggiornata. Viene inserito il nuovo elemento appena creato e ne viene calcolata la similarità con tutte

le altre viste del sistema. Inoltre, l'*update* della matrice prevede anche di eliminare la possibilità che le due viste su cui si è operato possano essere ulteriormente fuse, quindi la loro similarità settata a 0 rispetto alle altre. Una volta che tutte le viste vengono fuse, la matrice diventa nulla e la condizione di *stop* raggiunta. Dato che il *merging* avviene sempre e solo tra due viste, il risultato finale è un albero binario pieno[6]. Se il numero di *query* contenute nel *workload* è N, il numero di nodi del lattice, che sarà l'*input* per la seconda fase dell'algoritmo, è 2N – 1.

Il costo computazionale di questa prima fase è cubico. Il calcolo completo diviso per *step* è $N+N^2+(N-1)(N^2+3N)$. Il primo costo lineare si riferisce alla creazione di una vista per ogni *query*. N^2 indica il costo di creazione della *similarity matrix*. Infine, per ognuna delle N – 1 fusioni[7] viene ricercato il massimo ($O(N^2)$) e aggiornata la matrice ($3N$).

L'euristica proposta in [BPT97], che costruisce il reticolo multidimensionale riducendo a priori lo spazio di ricerca, viene valutata per un confronto con l'algoritmo proposto. Lo pseudo codice è il seguente:

Listing 4.5: Pseudocodice dell'euristica proposta da [BPT97]

```
1 repeat
2     Stop := True
3     foreach fd_i ϵ FD_DB
4         if (A ∩ A_i^r ≠ ∅) ∧ (ρ · size(v^{A_i^l}) < size(v^{A ∩ A_i^r}))
5             A := A − A_i^r
6             A := A ∪ A_i^l
7             Stop := False
8 until Stop
9 return A
```

dove A è un insieme di attributi del *dataset* multidimensionale, FD_{DB} è l'insieme delle gerarchie di attributi, ovvero di dipendenze funzionali

[6]Un albero binario è pieno (*full*) se ogni nodo interno ha esattamente due figli.

[7]Dato un albero binario pieno con N foglie, il numero di nodi interni è N – 1

$fd_i : A_i^l \rightarrow A_i^r$, *size($v_A$)* è una funzione che ritorna una stima del numero di tuple della vista v_A e p rappresenta la soglia sotto la quale il padre di una gerarchia dovrebbe essere usato al posto del figlio.
Il costo computazionale dipende linearmente dalle dipendenze funzionali contenute nelle dimensioni; l'idea di base è la valutazione, per la singola dipendenza gerarchica *parent* $\rightarrow$ *child*, del grado di miglioramento della soluzione con l'inserimento del *parent* rispetto al *child*.

Per tre principali motivi è stato scelto di modificare questo approccio:

- il primo, come già affermato in precedenza, è la scelta della precisione. Una funzione di stima è in contrasto con l'utilizzo delle statistiche mantenute dal DBMS ed entrambe le soluzioni hanno dei *pro* e dei *contro*; volendo una precisione maggiore, si deve fare i conti col problema del carico del *database* al momento di estrazione delle statistiche;

- il secondo motivo riguarda il numero di possibili dipendenze funzionali. In sistemi *enterprise* il numero di *dimension table* è elevato, così come il numero di attributi e di dipendenze funzionali. Il problema del costo computazionale dell'algoritmo è strettamente legato al tempo di risposta del prodotto, ma sulle tempistiche incidono anche altri fattori non direttamente gestibili dalla logica interna. Per ottenere un sistema utilizzabile in un ambiente aziendale, il fattore tempo ha un impatto notevole: le *query* che si vuole ottimizzare sono intrinsecamente lente, di conseguenza anche l'ottenere il numero di tuple o le dimensioni delle viste, se fatto analizzando le statistiche del *database*, è una stima lenta; è quindi maggiormente utile un algoritmo che riduca il più possibile lo spazio di ricerca, concentrandosi sulla scelta delle viste migliori tra quelle candidate solo in un secondo momento;

- infine la terza e principale motivazione riguarda la duplice utilità che lo strumento dovrebbe avere: oltre la scelta dell'aggregazione migliore, il prodotto deve tenere traccia degli attributi considerati dalle *query*, con

l'obiettivo di frammentare il fatto risparmiando sullo spazio che avrebbero occupato attributi non utilizzati. Inoltre, l'analisi della frequenza di utilizzo dei vari componenti del *data warehouse* (*dimension table*, *fact table*, misure, livelli di aggregazione, etc) permette di monitorarne lo stato, per procedere ad altri processi di ottimizzazione, come la dismissione di una tabella o l'eliminazione di un campo.

La seconda fase dell'algoritmo riguarda la scelta di quelle viste che riescono a coprire tutte le *query* di partenza e hanno costo minore. Di seguito viene riportato lo pseudocodice del secondo *step*:

Listing 4.6: Pseudocodice della seconda fase dell'algoritmo

```
1  Ordering functional dependencies by creation
2  ∀ f.d. (f^x : p^x → c_1^x, c_2^x, ..., c_n^x)
3    if C(p^x) < Σ_{i=1}^{N} C(c_i^x)
4      S ← S ∪ p^x
5      ∀ c_i^x ∈ S
6        S ← S - c_i^x
7    else
8      ∀ c_i^x ∈ f^x
9        S ← S ∪ c_i^x
10     Change f^y : p^y → c_1^y, c_2^y, ..., p^x, ..., c_m^y
11                    into f^y : p^y → c_1^y, ..., c_1^x, ..., c_n^x, ..., c_m^y
12 return S
```

Come già affermato, il risultato della prima fase sarà sempre un *full binary tree*, ovvero un albero binario nel quale tutti i nodi interni hanno esattamente due figli. In un albero di questo tipo, date N foglie, il numero totale di nodi interni è N – 1. Questo è anche il valore corrispondente al numero di dipendenze funzionali, considerando $p \rightarrow c_1$, c_2 una gerarchia di viste se p copre c_1 e c_2. Per ogni dipendenza funzionale tra viste, come appena definita, il padre o i figli entrano in soluzione in base al costo minore. Il

costo computazionale di questa seconda fase è, considerando il caso peggiore, quadratico. Nel dettaglio, dato sempre N come numero di *query* di partenza:

$$\sum_{i=1}^{N-1} N-1-i = (N-2)+(N-3)+...+(N-1-N+1) = \sum_{i=1}^{N-1} i-1 =$$

$$= \sum_{i=1}^{N-2} i = \frac{(N-2)(N-1)}{2} = O(N^2)$$

Una piccola considerazione viene fatta sul calcolo del costo della singola vista, dato che l'obiettivo principale rimane la minimizzazione di questo valore. Nello stato dell'arte di distingue tra costo di aggiornamento e costo di *query*: il costo di aggiornamento è definito come $\sum f_i^u \cdot C_i^u$ quindi la frequenza con cui vengono fatti gli inserimenti e gli *update* della vista moltiplicato per il costo del singolo aggiornamento, mentre il costo di interrogazione è $\sum_{q_i \in Q} f_{q_i} \cdot C_{q_i}(V)$ quindi la frequenza di interrogazione della singola *query* lanciata sul *dataset* composto dalle viste V. Il modello di costo utilizzato è semplice ma fornisce una buona valutazione empirica: non considerando l'utilizzo di altre forme di ottimizzazioni come partizioni o indici, il costo di *query* per una vista è indicato con il numero di tuple contenute, ipotizzando un *full table scan*. Altri fattori che incidono sulla bontà di una vista sono lo spazio occupato e il tempo di creazione. La funzione di costo è quindi un *trade-off* tra il tempo di risposta, lo spazio in *byte* e il tempo di aggiornamento. Considerando la scelta iniziale di avere stime precise basate sulle statistiche del DBMS, il costo dipende dai valori attendibili di numero di tuple, dimensione del singolo *record* e frequenza di interrogazione della *query*.
Un ultimo miglioramento dell'algoritmo raffina il calcolo dello spazio e del numero di tuple delle viste nella seconda fase dell'algoritmo. Secondo studi basati sul modello multidimensionale, l'analisi delle cardinalità dei singoli attributi e della sparsità del cubo risultano essenziali per migliorare l'approccio.
Dato uno schema di fatto con *pattern* primario[8] P_0, la sua sparsità è defi-

[8]Il *pattern* primario è l'insieme delle dimensioni primarie collegate al fatto.

nita dal rapporto $\frac{card(P_0)}{cardmax(P_0)}$ in cui $cardmax(P_0)$ è la massima cardinalità possibile, ovvero la produttoria delle cardinalità di dominio delle dimensioni [GR06]. Per quanto riguarda i *pattern* secondari, le tecniche per la stima della sparsità sono spesso di natura probabilistica; le più studiate partono o raffinano la formula di Cardenas, definendo dei limiti minimi e massimi per ciascuna cardinalità.
Procedendo nel calcolo del costo delle viste a ritroso, quindi nell'albero raffigurante la porzione del lattice multidimensionale dalla radice alle foglie, è possibile evitare la stima della sparsità (e quindi il numero di tuple dell'aggregato) di tutto il sottoalbero di cui v_a è *root* se $\frac{card(v_a)}{cardmax(v_a)} \geq delta\_percentuale$.
Il DFM in figura (fig 4.10) mostra un esempio delle possibili cardinalità di valori distinti di ciascun attributo della dimensione temporale e di quella riferita al prodotto.

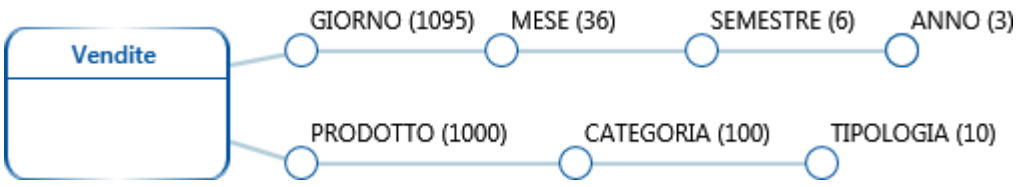


Figura 4.10: Esempio delle cardinalità dei valori distinti degli attributi per uno schema di fatto

Nel *dataset* di esempio sono modellati 3 anni e, di conseguenza, i 3 anni si ripercuotono in 6 semestri, 36 mesi e 1095 giorni. La seconda dimensione permette l'aggregazione delle vendite sulla gerarchia del prodotto. Dei 1095 giorni, supponiamo che 500 siano effettivamente valorizzati nel fatto di interesse, che equivale a dire che la *fact table* contiene 500 valori distinti per la *foreign key* temporale. Analogamente, 500 sono i prodotti venduti. Secondo le formule definite in precedenza, la cardinalità massima del fatto è 1095 * 1000 = 1095000 quando quella reale è 500 * 500 = 250000.
Lo stesso discorso viene fatto per tutti i *pattern* secondari che si vuole analizzare. Se la porzione del lattice calcolato dalla prima fase è la seguente (fig.

4.11)

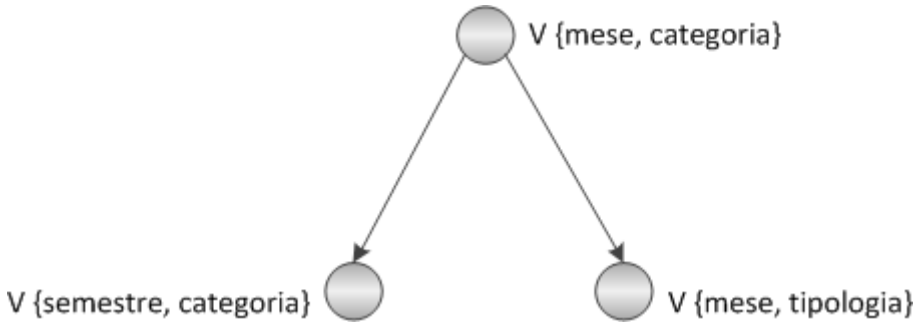


Figura 4.11: Esempio di porzione del reticolo multidimensionale con dei possibili livelli di aggregazione

e supponendo che la cardinalità di V{mese, categoria} differisce dalla produttoria dei valori distinti dei due attributi di un delta minore del 5%, le cardinalità di V{semestre, categoria} e V{anno, tipologia} possono essere stimate con un buon indice di confidenza come il prodotto delle cardinalità dei singoli attributi nelle *dimension table*.

Capitolo 5

Testing

In questo capitolo viene descritto il processo di progettazione ed implementazione del *test* dello strumento, tale processo sarà disinto in due fasi: la generazione degli aggregati e la creazione dei metadati. Vengono quindi presentati e commentati i risultati ottenuti.

5.1 Estrazione dei metadati

5.1.1 Progettazione del *testing*

Il *test* per l'estrazione dei metadati è stato eseguito su alcuni *dataset* creati *ad hoc* e aventi caratteristiche differenti tra loro.
É stata creata una tabella, che funge da *dimension table*, strutturata in tre differenti modi:

- la prima struttura, chiamata "lista lineare", corrisponde a un albero in cui ogni nodo interno ha uno e un solo figlio; dati n attributi, il numero di dipendenze funzionali e la profondità massima dell'albero è n - *1*
- la seconda struttura, chiamata "flat" (piatta), corrisponde a un albero in cui ogni nodo è figlio della radice; in questo caso, dati n attributi, l'altezza dell'albero è sempre 1 (una radice e n - *1* foglie)

- l'ultima tipologia, chiamata "ibrida", ha una struttura come riportata nella figura 5.1; la profondità varia con l'aumentare degli attributi

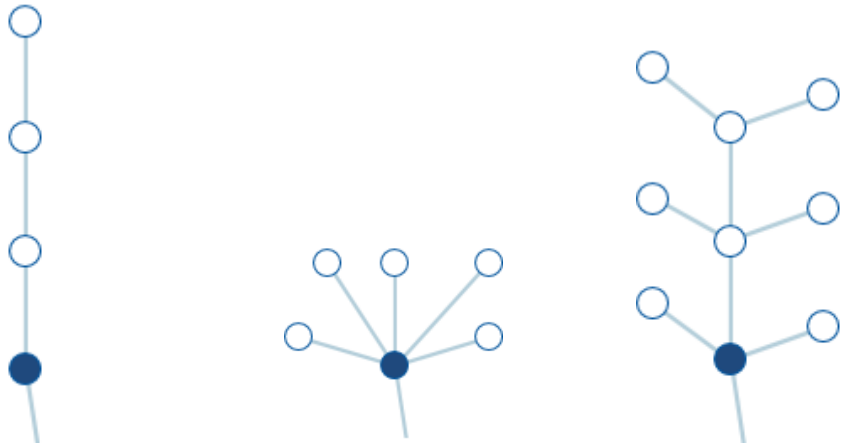


Figura 5.1: Tipologie di gerarchie utilizzate come *dataset* di *test*: la prima è la struttura chiamata "lista lineare", la seconda è quella "*flat*" e la terza è la "ibrida".

Il *check* di correttezza dei risultati è un processo obbligatoriamente manuale e richiede un intervento umano. Le problematiche principali sono due: da un lato è un processo basato sui dati, da cui si desume la composizione delle gerarchie; questo significa che possono essere frequenti i casi di falsi positivi e, in particolare, tanto più sono pochi i record delle tabelle, tanto più è alta la probabilità che ciò possa accadere (in una tabella con un record, ogni attributo dipende da tutti gli altri). Il secondo problema, come già detto, riguarda gli attributi descrittivi. Vengono considerati come normali attributi e quindi messi in gerarchia allo stesso modo dei surrogati.
I risultati mostrano il paragone tra le due euristiche (quella base e quella ottimizzata) presentate nel capitolo precedente. Entrambe sono state implementate in Java e lanciate sugli stessi *dataset* di tabelle. Anche se altre tecnologie di implementazione avrebbero presentato risultati migliori in termini assoluti, ciò che si è voluto tenere traccia da questi *test* è il paragone relativo tra i due algoritmi, sia a livello di tempistiche che a livello di qualità della so-

luzione, ovvero quanti e quali dipendenze funzionali vengono effettivamente trovate.

5.1.2 Risultati del *testing*

I *test* sopra definiti presentano dei risultati che devono essere analizzati nel dettaglio per estrarre le corrette informazioni a riguardo.
La prima distinzione può essere fatta sulla qualità del dato ottenuto in *output*: la prima delle due euristiche, ovvero quella che effettua la riduzione delle dipendenze transitive come fase seguente all'analisi di tutte le possibili coppie di attributi, presenta tre problemi principali rispetto al secondo algoritmo.
In primis, necessita di strutture temporanee (ad esempio tabelle relazionali) per memorizzare i dati intermedi, le cui scritture e letture ripetute peggiorano le prestazioni globali. L'unica struttura temporanea utilizzata dall'euristica che deduce le dipendenze transitive implicite è una matrice facilmente gestibile in memoria centrale.
Il secondo problema è che eventuali gerarchie condivise o convergenze vengono determinate solo nel caso in cui l'attributo ha un'unica distanza dalla radice dell'albero gerarchico sia passando per un *path* che per l'altro. Al contrario, tutti i casi sono determinati dal secondo algoritmo.
La terza problematica è sulla tempistiche dovute alla struttura del modello DFM: il caso peggiore, a livello teorico, per la seconda euristica è l'albero gerarchico piatto, ovvero dove la dimensione primaria, quindi la radice dell'albero, è direttamente collegata a tutte le foglie; sotto queste ipotesi, il numero di controlli effettuati è uguale per entrambi gli algoritmi. Tuttavia, il primo algoritmo non trae alcun vantaggio da una struttura dell'albero più complessa, mentre il secondo effettua meno controlli se può dedurre le dipendenze transitive interne.
I risultati sono mostrati in termini di numero di *query* effettuate, quindi ci si basa sul numero di operazioni di I/O eseguite. Il primo grafico (fig. 5.2) mostra il numero di interrogazioni dei due algoritmi all'aumentare del nu-

mero di attributi per una tabella che segue una struttura a "lista lineare". Dati n attributi, l'algoritmo base analizza $\frac{n^2-n}{2}$ coppie di attributi, mentre l'euristica ottimizzata effettua $n-1$ interrogazioni, tralasciando le $\frac{n^2-3n+2}{2}$ dipendenze funzionali sicuramente transitive. É con una struttura di questo tipo che si ottiene il miglioramento maggiore, dato che all'aumentare del numero di attributi l'algoritmo di base aumenta esponenzialmente il numero di *query* effettuate, mentre l'incremento per la seconda euristica ottimizzata è lineare.

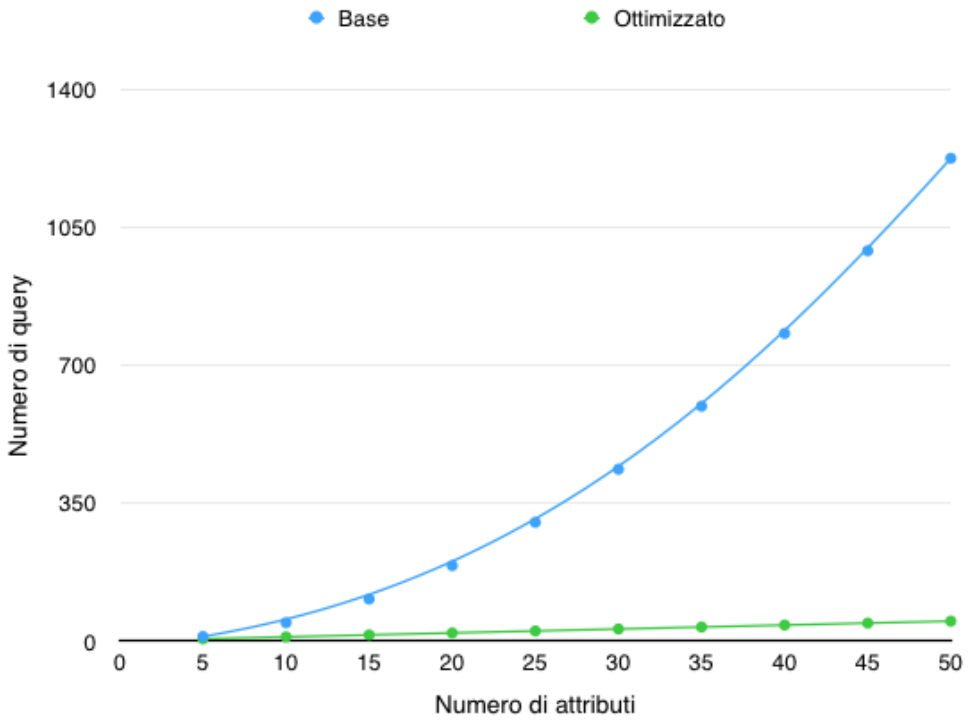


Figura 5.2: Il grafico mostra la comparazione in termini di numero di *query* tra i due algoritmi aventi come *input* tabelle con una struttura del tipo "lista lineare".

Il caso peggiore in termini di struttura della tabella si ha con la tipologia "flat". In questo caso (fig. 5.3) le due curve coincidono, dato che tutte le possibilità devono essere vagliate da entrambi gli approcci. Questo è dovuto alla mancanza di dipendenze transitive interne, che è la caratteristica per cui il secondo algoritmo performa meglio del primo in termini di numero di interrogazioni al *database*.

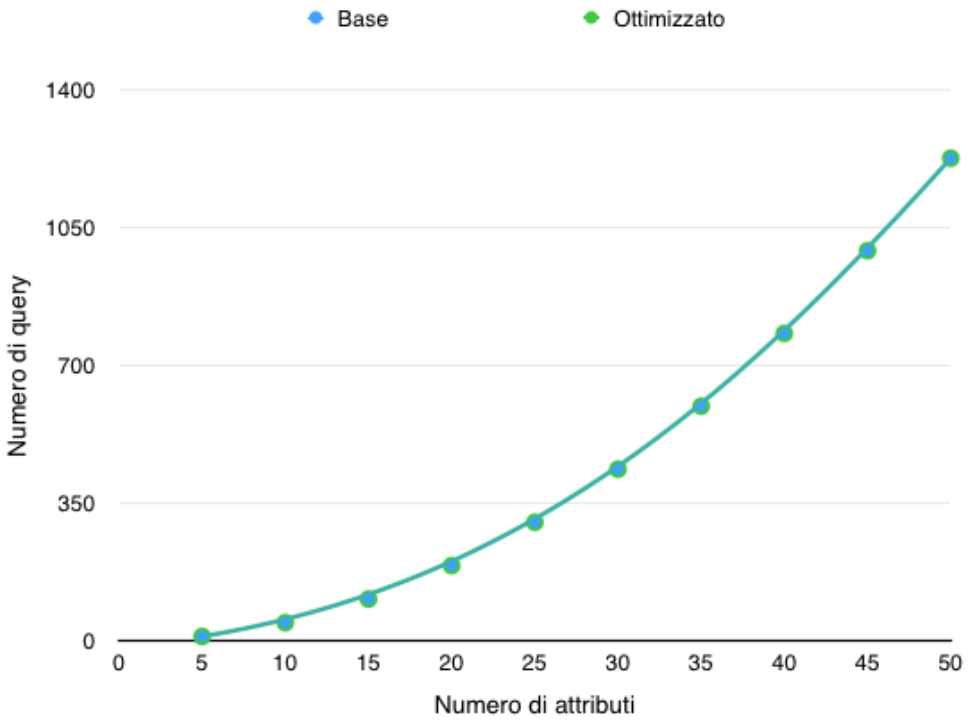


Figura 5.3: Il grafico mostra la comparazione in termini di numero di *query* tra i due algoritmi aventi come *input* tabelle con una struttura del tipo "*flat*".

L'ultima delle tre strutture analizzate è quella denominata "ibrida". In questo caso l'euristica ottimizzata ha anch'essa una tendenza esponenziale, ma con una pendenza minore (fig. 5.4). Una gerarchia strutturata in questo modo richiede un numero di *query* pari a $n * l + l - 1$, dove n è il numero di attributi ed l il livello dell'albero.

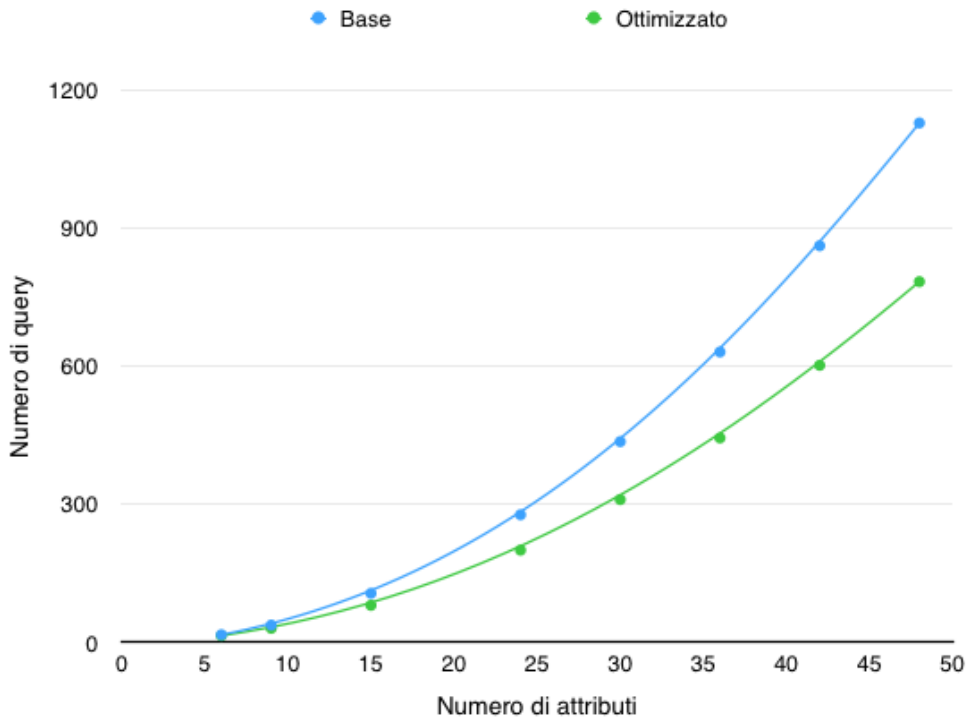


Figura 5.4: Il grafico mostra la comparazione in termini di numero di *query* tra i due algoritmi aventi come *input* tabelle con una struttura "ibrida".

Un'altro fattore di interesse su cui è stato effettuato un *test ad hoc* è rappresentato dal grafico in figura 5.5. Viene mostrato l'andamento del numero di *query* lanciate dagli algoritmi all'aumentare della profondità dell'albero. Questo *test* è stato fatto fissando a 20 il numero di attributi della tabella. Questa vista del problema permette di vedere come, fissato il numero di colonne della *dimension table*, la tipologia di gerarchia che ne definisce la struttura incide molto sul numero di operazioni di lettura eseguite dall'algo-

ritmo migliorato.

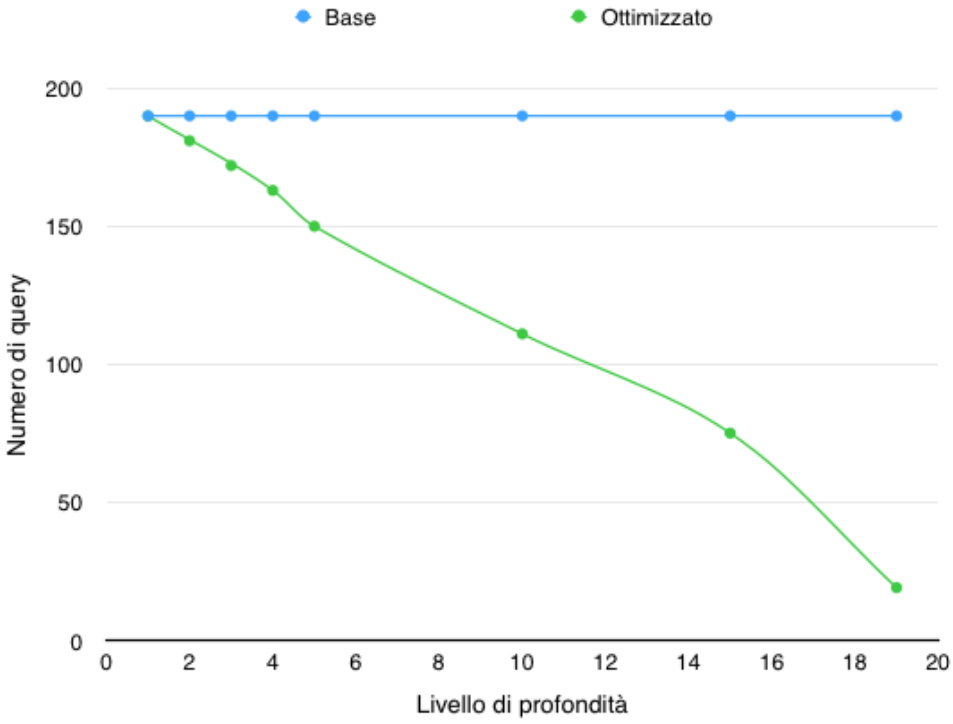


Figura 5.5: Il grafico mostra la comparazione in termini di numero di *query* tra i due algoritmi aventi come *input* tabelle con una struttura variabile (all'aumentare del livello di profondità dell'albero delle gerarchie) e con un numero fisso di attributi.

Un *test* definitivo è stato effettuato in un caso pratico di un *Enterprise Data Warehouse* (EDW) di una nota azienda del settore *retail*. La base dati contiene 177 *dimension table* e su queste è stata testata la procedura di *reverse engineering* che, partendo dai dati, cerca di estrarre le gerarchie della dimensione in termini di modello multidimensionale.
Si è notato che lo *speed up* ottenuto dall'esecuzione della seconda euristica rispetto alla prima è equivalente al 68,59 %, il cui significato è che, statisticamente, eseguendo i due algoritmi su una stessa *dimension table*, il secondo impiega circa un terzo del tempo rispetto al primo. La tabella 5.1 mostra il

paragone tra le due euristiche in termini di numero di *query* e di tempistiche, due misure estremamente correlate.

	Base	**Ottimizzato**	**Guadagno (%)**
Numero di query totale	112588	67611	39.9 %
Numero di query medio per tabella	636	382	
Tempo di esecuzione totale (sec)	12792.59	4017.64	68.59 %
Tempo di esecuzione medio per tabella (sec)	72.27	22.70	

Tabella 5.1: Estrazione delle gerarchie: base vs ottimizzato

5.2 Scelta degli aggregati

5.2.1 Progettazione del *testing*

Una vista è una *fact table* aggregata a un livello di dettaglio minore rispetto all'originale. Per testare il miglioramento apportato con l'aggiunta dell'aggregato, il processo prevederebbe uno *step* di riscrittura delle *query*, per ciascuna interrogazione che potrebbe essere risolta dalla *fact table* aggregata. Questa riscrittura obbliga una riorganizzazione completa di tutto il sistema di *reporting* del progetto.
Solitamente, in ogni sistema l'ambiente di modellazione e l'ambiente di *reporting* sono indipendenti. Molti sistemi dispongono dell'*aggregate navigation*, ovvero il processo di scelta della sorgente più efficiente a fronte di una interrogazione utente.
Di seguito vengono mostrati due esempi di come l'*aggregate navigator* lavora.
In Oracle Business Intelligence, al momento della modellazione, si associano alla stessa tabella logica diverse tabelle fisiche (ad esempio la *fact table* e l'aggregata); per ogni tabella fisica, si indica il livello di dettaglio. Al momento dell'esecuzione di un *report* o di un'analisi, una *query* logica viene creata e diverse tabelle logiche richiamate; l'*aggregate navigator* intercetta l'interrogazione e scrive la *query* fisica destinata al DBMS utilizzando le informazioni fornitogli a livello di modello.
In BusinessObject di SAP, viene utilizzata la funzione *@aggregate_aware* per definire incompatibilità tra gli oggetti. Ad esempio è possibile definire la dimensione giorno come incompatibile con la sorgente aggregata. Se di *default* l'aggregata viene scelta per prima, nel caso in cui l'analisi contenga un oggetto incompatibile, come il giorno, BusinessObject utilizza la seconda sorgente per l'interrogazione.
La scelta dell'utilizzo dell'*aggregate navigator* a discapito di *test* specifici sulle *query* ha un duplice scopo: il primo, come già detto, è quello di non modificare il modello ed evitare la riscrittura manuale della *query*, che avrebbe aggiunto un'ulteriore incognita sulla correttezza dei *test*; la seconda motiva-

zione è dettata dalla volontà di avere uno strumento nella pratica utilizzabile in ambiente aziendale. Il *test* deve rispecchiare quindi il più possibile il processo classico del mondo *enterprise* e quello più comodo per l'utilizzo che se ne fa del sistema.

Lo strumento fornisce in *output* una rappresentazione grafica degli aggregati analizzati secondo una struttura a *bubble chart* (fig. 5.6); ogni bolla rappresenta un aggregato e fornisce informazioni sulle tre coordinate valutate, ovvero il tempo di aggiornamento, lo spazio occupato (o tempo di risposta) e la frequenza di analisi. Gli aggregati che vengono mostrati in *output* sono tutti quelli analizzati e creati dall'algoritmo durante l'esecuzione su uno specifico *workload*, quindi sia quelli corrispondenti a un'unica *query* che quelli fusi per risolverne un numero maggiore. Intuitivamente gli aggregati con bolle più estese (ovvero con una maggiore frequenza) e più vicini all'origine (ovvero con un minore spazio di occupazione e un minore tempo di aggiornamento) sono quelli più efficaci. Le viste migliori, definite tali dalla minimizzazione della funzione di costo, vengono etichettate come **IBC** (*In Best Configuration*, ovvero facenti parte della configurazione migliore). Il *mapping* di tutti gli aggregati IBC permette di ridurre al massimo (massimo sempre definito dalla funzione di costo utilizzata) il tempo di esecuzione totale delle analisi del *workload*.

Ciascun aggregato può essere visualizzato più nel dettaglio e ha associata una rappresentazione in formato SQL. Da notare che, nel caso in cui il sistema sia modellato interamente con un approccio *snowflake* (alcuni strumenti di *front-end* lavorano meglio con strutture di questo tipo, come *Microstrategy*), le uniche viste da creare sono le *fact table* aggregate; se il modello è differente, quindi l'implementazione del *data warehouse* è a *star schema* (o una soluzione ibrida), nuovi aggregati devono essere creati anche per ciascuna *dimension table* di cui non è presente una versione materializzata al livello di dettaglio richiesto.

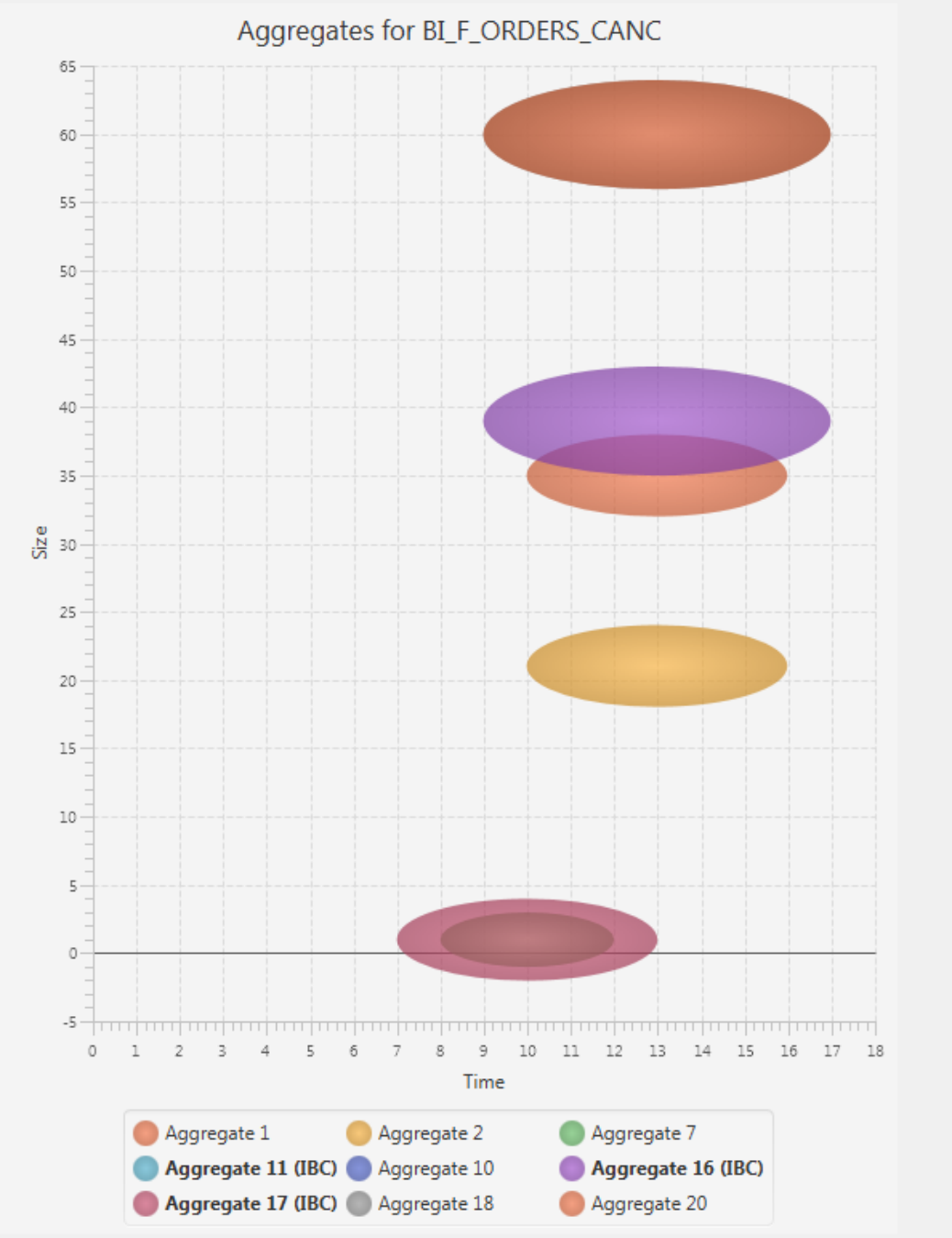


Figura 5.6: *Output* dello strumento: ogni bolla corrisponde a un aggregato valutato dall'algoritmo; quelli etichettati come IBC sono quelli considerati migliori secondo la funzione di costo.

Una volta create tutte le tabelle necessarie, deve essere effettuato il *mapping* dell'aggregato nel *software* di *front-end* utilizzato. I *test* sono stati effettuati su OBIEE 11g (Oracle Business Intelligence Enterprise Edition 11g), quindi viene presentato un esempio di modellazione dell'aggregato già fisicamente creato (si ipotizza quindi che il processo di caricamento dei dati della vista materializzata in fase di ETL sia già stato completato).
OBIEE 11g definisce la modellazione del DWH su tre livelli distinti:

- il livello fisico definisce gli oggetti e le relazioni disponibili sul *server* per la scrittura delle *query* fisiche; l'oggetto principale di questo livello è il "*database*" che definisce la sorgente dati e, in base a questa, le regole da applicarvi (possono esistere collegamenti a *database* relazionali, come SQL Server e Oracle DB, o a cubi multidimensionali, come Essbase);

- il livello logico definisce il modello in modo indipendente dalla tecnologia fisica utilizzata come sorgente dati; è il *layer* di collegamento tra il modello di *business*, sempre dimensionale, e il modello fisico, specifico in termini di tecnologie e implementazione;

- il livello di presentazione fornisce un modo per presentare il modello di *business* agli utenti.

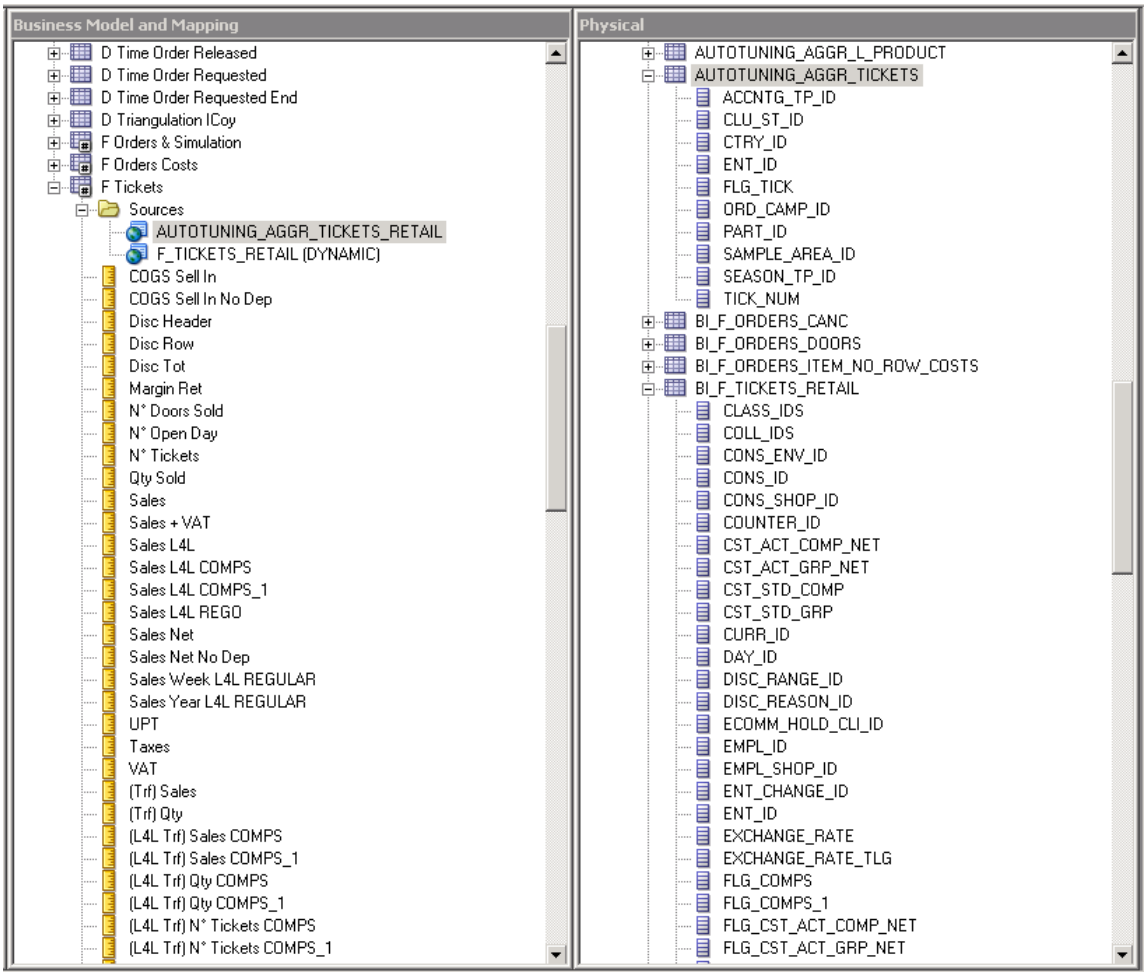


Figura 5.7: La figura mostra a sinistra il *layer* logico e a destra il *layer* fisico: nel fisico sono mappate le tabelle reali e si nota come l'aggregato "AUTOTUNING_AGGR_TICKETS" abbia un numero di attributi minori rispetto alla tabella originaria "BI_F_TICKETS_RETAIL", mentre nel logico la tabella "F Tickets" ha come sorgenti fisiche entrambe le versioni.

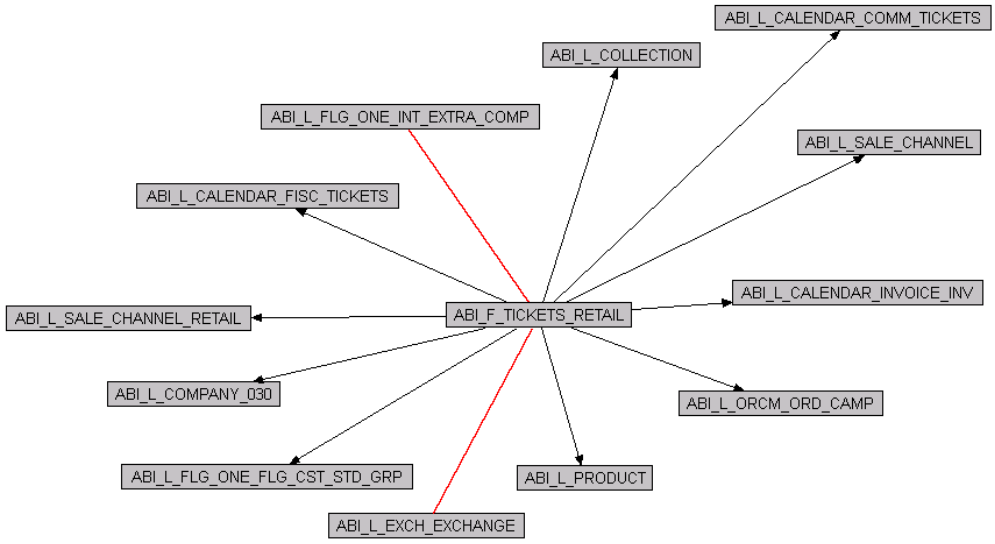


Figura 5.8: La figura mostra le relazioni tra le *lookup* e la tabella originale.

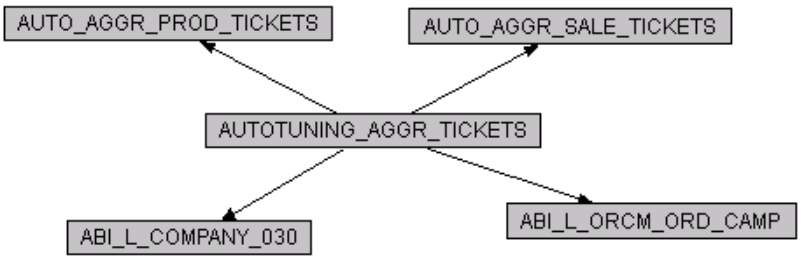


Figura 5.9: La figura mostra le relazioni tra le *lookup* e la tabella aggregata: non tutte le *dimension table* sono presenti e, inoltre, alcune sono in forma aggregata.

5.2.2 Risultati del *testing*

I risultati riportati fanno riferimento al confronto tra i tempi ottenuti su alcune analisi eseguite sul modello preso ad esempio prima e dopo il *mapping* degli aggregati *In Best Configuration* ottenuti dal sistema.
La tabella 5.2 mostra una media dei tempi di esecuzione per nove analisi di *business* corrispondenti alle rispettive *query* SQL, riportate per i due scenari successivi.

ANALISI	TEMPO ORIGINALE	TEMPO CON AGGREGATI	MIGLIORAMENTO
1	346,45 s	4,38 s	98,74 %
2	246,90 s	8,39 s	96,60 %
3	244,38 s	8,32 s	96,59 %
4	261,80 s	60,34 s	76,95 %
5	97,00 s	12,20 s	87,42 %
6	175,00 s	0,08 s	99,95 %
7	237,28 s	33,04 s	86,07 %
8	100,70 s	0,28 s	99,72 %
9	202,03 s	25,70 s	87,28 %
			92,01 %

Tabella 5.2: La tabella mostra per ciascuna analisi i tempi di esecuzione delle *query* prima e dopo l'inserimento degli aggregati proposti dallo strumento e il miglioramento ottenuto in termini percentuali.

Il miglioramento medio è del 92,01 %, ma anche in questo caso occorre fare alcune precisazioni:

- per evitare condizionamento nei risultati, il *dataset* di esempio era privo di qualsiasi struttura volta al miglioramento delle *performance*, come indici e partizionamenti. In un caso reale, il miglioramento medio ri-

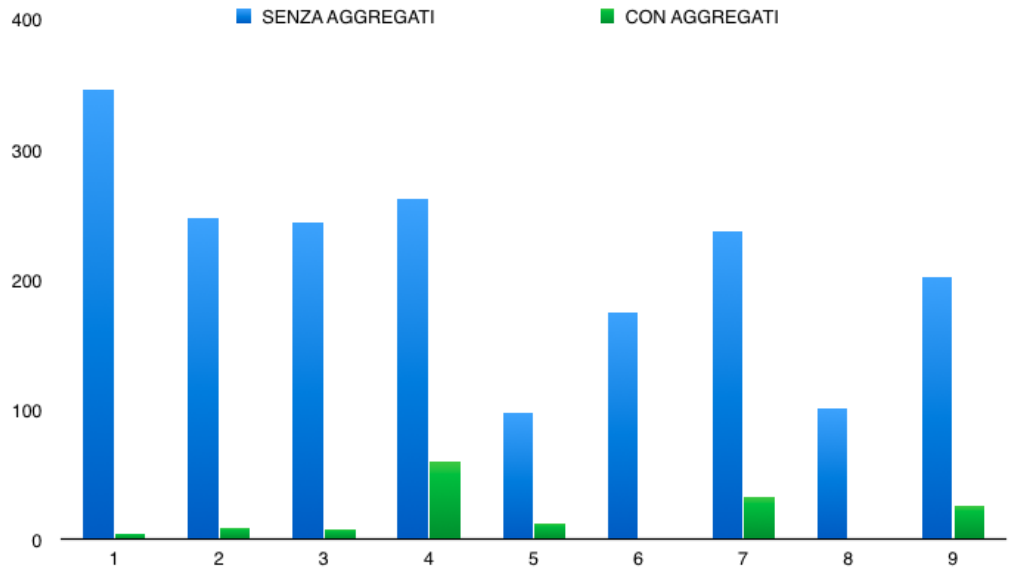


Figura 5.10: Il grafico mostra i tempi di ciascuna analisi prima e dopo l'aggiunta degli aggregati al modello.

sulterebbe più contenuto, anche se il *delta* di *speed up* perso sul tempo di esecuzione lo si ritroverebbe sul tempo di creazione degli aggregati;

- sempre in riferimento al punto precedente, tutti i livelli di *caching*, sia sullo strumento di reportistica OBIEE che sul *database*, sono stati disabilitati; lasciando a questi strumenti commerciali la decisione sull'utilizzo delle memorie *cache*, i tempi di *retrieve* dei risultati delle analisi non possono che diminuire;

- la decisione di utilizzare, per la scelta degli aggregati migliori, una funzione di costo rispetto a dei vincoli di sistema ha dei *pro* e dei *contro*; da un lato fornisce una maggiore flessibilità sulle soluzioni ottenibili ovvero, in base all'evenienza, è possibile dare maggior peso a un vincolo rispetto che a un altro; tuttavia eventuali modifiche a questi pesi fan si che i risultati cambino completamente, perciò occorrerebbe un'analisi specifica sulla qualità dell'*output* ottenuto a fronte di un cambiamento della funzione di costo.

Conclusioni

La continua crescita dell'utilizzo di soluzioni di *business intelligence* in ormai ogni tipologia di azienda denota l'importanza che assumono oggi le informazioni nel processo direzionale. In un sistema di supporto alle decisioni l'aspetto centrale rimane la qualità dei dati forniti in *output*, ma non è l'unica caratteristica del successo dei *data warehouse* oggi. Spesso il limite di questi sistemi è dovuto alla lentezza con cui vengono eseguite le analisi; per sua natura, un *data warehouse* memorizza grandi quantità di dati integrati e puliti a partire da molte sorgenti operazionali. Risolvere questa problematica è stata, ed è tutt'ora, uno dei principali oggetti di studio sull'argomento; i *data warehouse* basati su un'implementazione ROLAP si appoggiano alle tecniche di *tuning* delle *performance* proprie dei *database* relazionali, come indici e partizionamenti. Una tra le tecniche più efficaci per quanto riguarda i *data warehouse* è però quella della materializzazione delle viste, ovvero della creazione e dell'alimentazione di *fact table* aggregate a un livello di dettaglio minore rispetto a quelle di base. Questo permette di evitare molti *join* a *run-time* diminuendo drasticamente i tempi di attesa delle *query*.
Negli ultimi anni si stanno espandendo molto altre tecnologie che forniscono soluzioni alternative al problema delle *performance*, come i *database* colonnari e gli *in-memory*, ma la loro diffusione è ancora limitata, rendendo la creazione degli aggregati lo *standard de facto* per il problema del *tuning*.
Tuttavia, la scelta di quali siano i migliori aggregati generabili è un processo che viene fatto in modo manuale e che prevede un'analisi dettagliata delle interrogazioni più frequenti e una conoscenza approfondita del modello e dello

schema sottostante. Queste analisi devono inoltre essere effettuate per ogni instanza e per ogni *set* di *query* frequenti di cui si vuole effettuare il *tuning*. L'idea che si è voluta implementare in questo lavoro di tesi è quella di automatizzare questo processo in modo da semplificare il lavoro del *database administrator* riguardo ai problemi sulle *performance* del *data warehouse*. L'applicazione di algoritmi *ad hoc* già largamente studiati in campo accademico ha permesso di ottenere buoni risultati sui *test* effettuati.

Tuttavia, l'applicabilità reale di questo strumento può essere ottenuta solo a seguito di eventuali sviluppi futuri. Per prima cosa occorre integrare completamente il sottosistema che si occupa della definizione dei metadati su cui poi si basano gli algoritmi di creazione degli aggregati; il processo di definizione delle gerarchie fornisce delle proposte che devono essere validate in modo manuale e questo potrebbe essere fatto con un supporto grafico se venisse integrato in un *software* di modellazione. Inoltre, al momento l'algoritmo gestisce solo le dipendenze in cui l'insieme determinante e l'insieme dipendente sono composti da un unico attributo ciascuno; non è quindi ancora possibile determinare dipendenze in cui una coppia di attributi ne determina un terzo. Attualmente è stata implementata solamente una versione in grado di analizzare *data warehouse* che risiedono su tecnologia Oracle, quando il sistema è stato pensato per essere *vendor-indipendent*: dato che lo strumento ha una struttura modulare, questa modifica inciderebbe solo sul componente chiamato *Extractor*, nella fase di ricerca di informazioni sul *workload* del *database*. Le stesse logiche sono applicabili anche a sistemi MOLAP, quindi si potrebbe valutare un'analisi sull'implementazione del *tool* anche in questo ambito.

Attualmente AutoTuning fornisce anche dei *report* sulle statistiche di utilizzo delle *fact table*, delle misure in esse contenute e delle *lookup table* richiamate dalle analisi: potrebbe essere interessante aggiungere come funzionalità l'automatizzazione di altri processi di *tuning*, come l'analisi automatica su quali siano gli oggetti ormai non più utilizzati, ma ancora presenti ed alimentati sul *data warehouse*.

Infine, un'ultima ipotesi di evolutiva potrebbe riguardare direttamente l'al-

goritmo di ricerca degli aggregati, testando e valutando funzioni di costo più complesse che stimino il tempo di creazione dell'aggregato o il tempo di *retrieve* dei risultati d'analisi attraverso una regressione statistica i cui parametri vengano automaticamente calcolati dallo strumento.

Bibliografia

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated selection of materialized views and indexes for sql databases. 2000.

[AJD06] Kamel Aouiche, Pierre-Emmanuel Jouve, and Jérôme Darmont. Clustering-based materialized view selection. 2006.

[Bad11] K. V. Badmaeva. The algorithm of view selection for materializing in specialized data warehouses. 2011.

[BPT97] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. Materialized view selection in a multimensional database. 1997.

[CDK+] Surajit Chaudhuri, Sudipto Das, Christian Konig, Vivek Narasayya, Hyunjung Park, and Manoj Syamala. Microsoft research: Autoadmin.

[CGR01] Paolo Ciaccia, Matteo Golfarelli, and Stefano Rizzi. On estimanting the cardinality of aggregate views. 2001.

[CN97] Surajit Chaudhuri and Vivek Narasayya. An efficient, cost-driven index selection tool for microsoft sql server. *VLDB*, 1997.

[GP11] Laurence Goasduff and Christy Pettey. Gartner identifies nine key data warehousing trends for the cio in 2011 and 2012, February 2011.

[GR06] Matteo Golfarelli and Stefano Rizzi. Progettazione logica. In *Data Warehouse: teoria e pratica della progettazione*. McGraw-Hill, 2006.

[GRS02] Matteo Golfarelli, Stefano Rizzi, and Ettore Saltarelli. Index selection for data warehousing. *Proceedings 4th International Workshop on Design and Management of Data Warehouses*, pages 33–42, 2002.

[Gup97] Himanshu Gupta. Selection of views to materialized in a data warehouse. 1997.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. 1996.

[Inm92] William H. Inmon. Building the data warehouse, 1992.

[Kim95] Ralph Kimball. How to optimize your data warehouse using aggregates without driving your end users crazy., November 1995.

[Ros04] Margy Ross. The kimball bus architecture and the corporate information factory: What are the fundamental differences?, March 2004.

[SDN] Amit Shukla, Prasad M. Deshpande, and Jeffrey F. Naughton. Materialized view selection for multidimensional datasets.

[YCGY02] Xin Yao, Chi-Hon Choi, Gang Gou, and Jeffrey Xu Yu. Materialized view selection as constrained evolutionary optimization. 2002.

[Zav] Gianluigi Zavattaro. Grammatiche e linguaggi liberi dal contesto.

[ZY99] Chuan Zhang and Jian Yang. Genetic algorithm for materialized view selection in data warehouse environments. 1999.

[ZYY01] Chuan Zhang, Xin Yao, and Jian Yang. An evolutionary approach to materialized views. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS*, 31:282–294, 2001.