Università degli Studi di Modena e Reggio Emilia

Facoltà di Ingegneria – Sede di Modena

Corso di Laurea Specialistica in Ingegneria Informatica

BART: Uno strumento di analisi di business e reportistica

Relatore:
Prof. Sonia Bergamaschi

Candidato:
Mattia Bonacorsi

Anno Accademico 2005-2006

PAROLE CHIAVE:
DATA_WAREHOUSE
OLAP
BUSINESS_INTELLIGENCE
OPEN_SOURCE
MONDRIAN

INDICE

INDICE DELLE FIGURE

INDICE DELLE TABELLE

1. INTRODUZIONE

La presente tesi è stata svolta durante un periodo di stage della durata di 6 mesi presso l'azienda "Quix Srl" di Soliera.

Il periodo di stage prevedeva lo studio degli scenari applicativi e degli standard che forniscono supporto alla *business intelligence* offerti dal mercato.

L'analisi è stata concentrata sul *modello multidimensionale* OLAP, sul processo di definizione e creazione di un data warehouse e dei cubi contenuti al suo interno, sul linguaggio MDX, che permette di interrogare i dati contenuti in un cubo. Sono inoltre state affrontate le problematiche di alimentazione periodiche dei nuovi dati.

Si è quindi progettato e implementato un prototipo in grado di definire cubi di dati basati su strutture relazionali.

Durante lo stage è stato analizzato un algoritmo che, dato un database multidimensionale e un insieme di interrogazioni *caratteristiche* per un'applicazione, calcola l'insieme delle possibili aggregazioni dei dati che, compatibilmente con i vincoli del sistema, se materializzate potranno produrre vantaggi prestazionali all'esecuzione delle interrogazioni in oggetto.

Si è scelto di sviluppare il prototipo BART come applicazione Web per permettere agli utenti di interrogare il server OLAP attraverso un browser, senza dover installare altre applicazioni (gli altri componenti del sistema saranno installati su uno o più server).

Per lo sviluppo del prototipo si è scelto di utilizzare il server OLAP Mondrian come motore per risolvere le interrogazioni multidimensionali, JPivot come front-end per visualizzare e navigare i risultati delle interrogazioni, il framework Jakarta Struts per lo sviluppo di un'applicazione Web conforme al pattern MVC e Java come linguaggio di programmazione.

L'articolazione della tesi è la seguente.

Nei capitolo 2 saranno analizzate le tecnologie utilizzate per creare l'applicazione.

In particolare, nei primi due paragrafi, sarà proposta un'introduzione sui sistemi di data warehouse e un'introduzione al linguaggio MDX che permette di interrogare i dati multidimensionali.

Nei successivi 4 paragrafi saranno analizzate le tecnologie open source utilizzate per lo sviluppo del prototipo. In particolare, nei paragrafi 3 e 4 si approfondirà il server OLAP Mondrian, nel paragrafo 5 sarà proposta una presentazione della piattaforma per la *business intelligence* Pentaho che integra, tra gli altri strumenti, Mondrian, e nel paragrafo 6 sarà descritto il componente di front-end JPivot.

All'interno del paragrafo 7 sarà presentato un algoritmo per la selezione delle viste candidate alla materializzazione, all'interno del paragrafo 8 saranno presentate le principali tecniche di alimentazione dei data warehouse e, infine, nel paragrafo 9 sarà descritto il framework Jakarta Struts.

Nel capitolo 3 sarà descritto il prototipo realizzato durante il periodo di stage.

In particolare, nel primo paragrafo sarà proposta una descrizione generale del prototipo realizzato, nel paragrafo 2 saranno descritte le funzionalità del prototipo e nel paragrafo 3 i due contesti reali a cui è stato applicato il prototipo.

All'interno del paragrafo 4 saranno presentate alcune interrogazioni realizzate all'interno del prototipo e nel paragrafo 5 sarà presentata l'applicazione dell'algoritmo di selezione delle viste descritto nel capitolo 2 ad uno dei contesti analizzati.

Nel quarto capitolo saranno presentate le conclusioni e i possibili sviluppi futuri, nel quinto capitolo verrà introdotto un breve glossario con i termini utilizzati in questa tesi e nel sesto capitolo sarà presentata una bibliografia ragionata.

Nella stesura della tesi sono state rispettate le seguenti linee guida: tutti gli acronimi sono stati scritti in maiuscolo, le parole del glossario in corsivo e i nomi di strumenti utilizzati nella costruzione del prototipo con un carattere diverso (come, ad esempio, Mondrian).

2. LE TECNOLOGIE UTILIZZATE

2.1 DATA WAREHOUSE

Per definire cosa sia un data warehouse ci rifacciamo alla definizione di uno dei padri fondatori della disciplina del data warehousing, William H. Inmon:

"Il data warehouse è una collezione di dati: orientata al soggetto, integrata, non volatile, dipendente dal tempo."

I concetti chiave della definizione sono:

- orientata al soggetto: perché il data warehouse è orientato a temi specifici per l'azienda piuttosto che alle applicazioni o alle funzioni. L'obiettivo, quindi, non è più quello di minimizzare la ridondanza mediante la normalizzazione ma quello di fornire dati che abbiano una struttura in grado di favorire la produzione e fruizione delle informazioni;
- integrata: il data warehouse deve essere orientato all'integrazione della raccolta dati, in esso confluiscono dati provenienti da più sistemi transazionali e da fonti esterne. L'integrazione può essere raggiunta percorrendo diverse strade che non si escludono tra loro: omogeneità semantica di tutte le variabili, utilizzo di metodi di codifica uniformi, utilizzo delle stesse unità di *misura*, ecc…;
- non volatile: i dati contenuti nel data warehouse consentono accessi in sola lettura;
- dipendente dal tempo: i dati di un data warehouse hanno un orizzonte temporale molto più esteso rispetto a quelli archiviati nei sistemi operazionali. L'osservazione "storica" del dato è una peculiarità del data warehouse che ha dati che si legano alla *misura* tempo e si astraggono dalla pura scansione di eventi del processo aziendale.

La struttura di un generico data warehouse comprende:

- una parte di ETL (estrazione, trasformazione e caricamento) che permette la pulitura e conformazione dei dati provenienti dai vari sistemi operazionali;
- una serie di *data mart* come unità di osservazione dei processi aziendali;
- una parte di presentazione che non è altro che un insieme di applicazioni e di report.

Il modello di dati dimensionali utilizzato dai data warehouse differisce dal modello normalizzato, ma ne eredita i concetti fondamentali. Sono ancora presenti le entità, gli attributi e, quando il modello dimensionale è relazionale, le relazioni. Cadono invece tutte le altre regole di normalizzazione dei database relazionali. Si tende a denormalizzare il database appiattendo nelle dimensioni i valori ripetuti. Questo porta sicuramente ad uno spreco di spazio fisico su disco, ma per contro evita operazioni di JOIN superflue per recuperare dati distribuiti su più tabelle.

Riguardo alle chiavi primarie e alle chiavi esterne delle tabelle di dimensioni e fatti, bisogna aggiungere che è opinione assunta della comunità degli esperti di data warehouse che le chiavi dei sistemi operazionali vanno sostituite da chiavi create su *misura* in area di staging, per separare completamente le logiche dei sistemi operazionali da quelle del data warehouse,

rendendolo indipendente in tutto e per tutto. Queste chiavi, così create, sono chiamate chiavi surrogate.

Le entità nei database di data warehouse sono organizzate in tabelle dei fatti e tabelle delle dimensioni.

Le tabelle dei fatti sono le principali nel modello ed in esse sono memorizzate le *misure* delle prestazioni numeriche dell'azienda per un determinato processo aziendale. Supponiamo di analizzare la vendita di prodotti in un mercato e di scrivere l'importo e quantità delle vendite ogni giorno, per ciascun prodotto in tutti i negozi della catena. L'elenco degli attributi di dimensione definisce la grana ovvero la profondità dimensionale della tabella dei fatti. Le tabelle delle dimensioni sono quelle contenenti i descrittori della tabella dei fatti e di solito hanno molte colonne e poche righe, anche se è possibile incontrare tabelle dimensionali con milioni di righe (come ad esempio la dimensione cliente in alcuni modelli dimensionali). Spesso non è chiaro se la tabella debba essere considerata una tabella dei fatti o di dimensione; una buona regola può essere quella di chiedersi: *La tabella contiene misure che parteciperanno a calcoli?* Se la risposta è si la tabella è dei fatti, altrimenti siamo di fronte ad una tabella di tipo dimensionale.

La figura seguente mostra lo schema di un *data mart* che rappresenta i fatti di vendita di una ipotetica azienda. Si può osservare il classico schema a stella con una tabella dei fatti "vendite giornaliere", le sue *misure* "quantità" e "ammontare" e le chiavi esterne verso le varie dimensioni descrittive dei fatti.

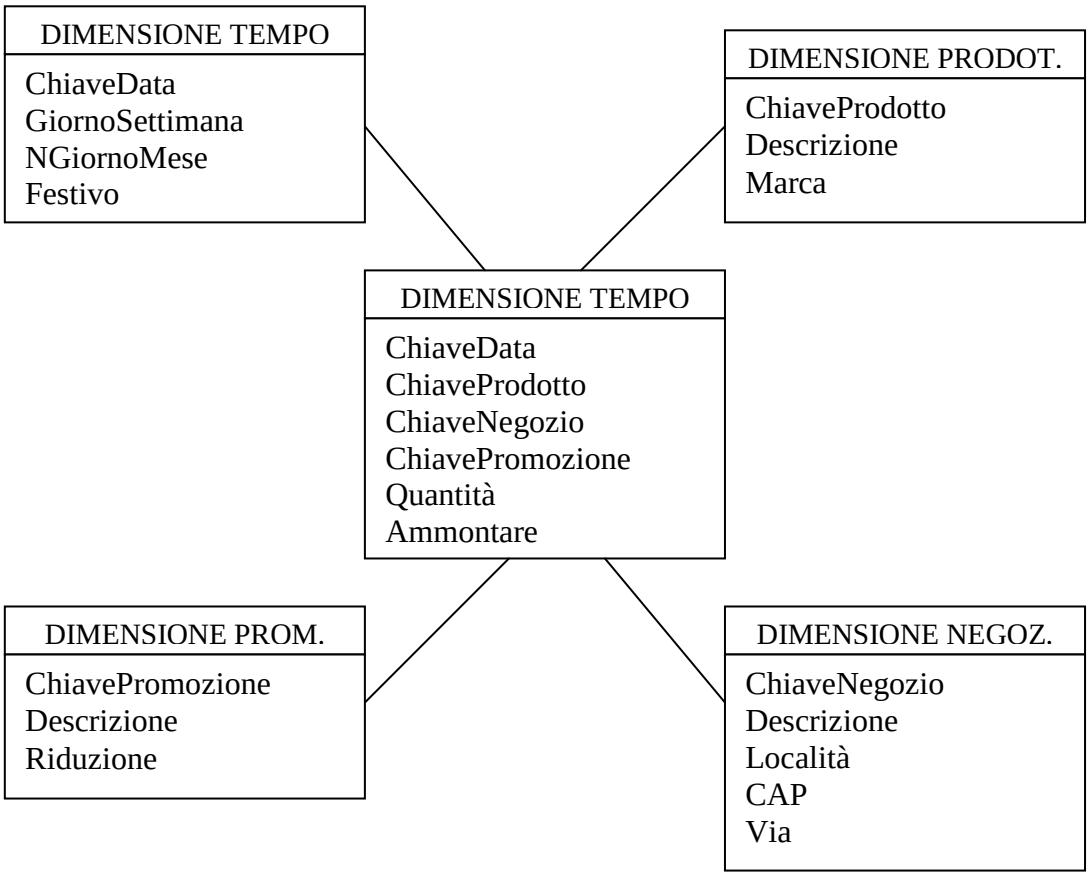


Figura 1 - Esempio di schema a stella

Non è sempre vero che bisogna attendersi ad uno schema a stella stretto. In alcuni casi è necessario utilizzare anche schemi più articolati. Quando, ad esempio, uno schema contiene dimensioni modellate con una serie di tabelle con una struttura ad albero, viene definito "a fiocco di neve".

Per la progettazione di un data warehouse si può indicare un schema in quattro passi che è stato proposto per primo dall'esperto di data warehouse Ralph Kimball.

1. Selezionare il processo da modellare: ascoltare gli utenti è il modo più efficiente per scegliere il processo aziendale ed è bene ricordare che, quando si parla di processi aziendali, non si ci riferisce a una funzione organizzativa; guardando ai reparti si rischia di duplicare le estrazioni, trasformazioni, pubblicazioni e viste dei dati.
2. Dichiarare la grana del processo aziendale: significa specificare esattamente ciò che rappresenta ogni singola riga della tabella dei fatti. La domanda che ci si deve porre è: *Come si descrive un'unica riga della tabella dei fatti?* La definizione di grana è fondamentale e se si scopre nelle fasi seguenti di aver compiuto un errore, è necessario ricominciare da questo punto.
3. Scegliere le dimensioni che si applicano alla tabella dei fatti: se la definizione della grana è chiara e soddisfacente, trovare le dimensioni è di solito banale. La domanda a cui rispondere è: *In che modo coloro che lavorano in una azienda descrivono i dati generati dal processo aziendale?*
4. Identificare i fatti numerici che popoleranno ogni riga della tabella dei fatti: la domanda per individuare le *misure* è: *Cosa stiamo misurando?* È fondamentale che tutti gli attributi della tabella dei fatti candidati come *misure* rispettino la grana. I fatti che appartengono ad una grana diversa devono trovarsi in una tabella dei fatti distinta.

L'implementazione di un sistema di data warehouse è molto complessa e va scomposta in più obiettivi raggiungibili. Pur essendo questa un'ottima regola da seguire in tutti i progetti informatici complessi, bisogna tener presente che scomporre il problema non significa creare il data warehouse in parti isolate.

Un metodo di progettazione che si è rilevato molto utile nella progettazione di sistemi di data warehouse complessi è l'architettura a bus. L'idea è quella di applicare alcuni dei paradigmi della progettazione object oriented ai data warehouse, pensando ai fatti e alle dimensioni come ad oggetti riutilizzabili all'interno di più *data mart*. Appare chiaro che nella fase di progettazione bisogna identificare fatti e dimensioni comuni, che sono detti conformati, da potersi utilizzare nei *data mart*. Perché le dimensioni ed i fatti siano conformati è necessario che abbiano chiavi di dimensioni e fatti coerenti, nomi di colonne e attributi coerenti e valori attributo e definizioni coerenti. In concreto le dimensioni e i fatti conformati sono un sottoinsieme rispetto alla dimensione più dettagliata.

Per l'individuazione delle dimensioni e dei fatti riutilizzabili viene in aiuto la matrice del bus del data warehouse che ha sulle righe i *data mart* e sulle colonne le dimensioni. I *data mart* così individuati sono definiti consolidati.

Processi aziendali	Data	Prodotto	Negozio	Promozione	Magazzino	Produttore	Contratto	Spedizioniere
Vendite al dettaglio	X	X	X	X				
Inventario al dettaglio	X	X	X					
Consegne al dettaglio	X	X	X					
Inventario al magazzino	X	X			X	X		
Consegne di magazzino	X	X			X	X		
Ordini di acquisto	X	X			X	X	X	X

Tabella 1 - Esempio di matrice Bus Data Warehouse

Parlando delle dimensioni conformate è necessario specificare che l'utilizzo del subsetting va esteso quando i modelli dimensionali affrontano analisi che richiedono livelli di dettaglio differenti su sottoinsiemi dello stesso insieme. Pensiamo ad esempio alla situazione di un data warehouse nel settore finanziario. In questo settore i prodotti possono essere eterogenei tra loro. In una banca le differenze che esistono tra un conto corrente di deposito e i depositi vincolati come i certificati di deposito sono minime ma significative (ad esempio, i depositi vincolati non hanno saldi minimi e limiti di scoperto). Un'osservazione globale con possibilità di slice e dice su ciascun prodotto è necessaria, ma sicuramente può sorgere la necessità di analisi particolari sulle caratteristiche non in comune.

A questo punto un solo *data mart* non può più rispondere alle esigenze e diventa necessario creare un nuovo *data mart* che risponda alle nuove esigenze.

2.2 MULTIDIMENSIONAL EXPRESSION (MDX)

2.2.1 INTRODUZIONE A MDX

MDX, acronimo di MultiDimensional Expressions, è un linguaggio che consente la definizione e la manipolazione di oggetti e dati multidimensionali. Per molti aspetti MDX è simile al linguaggio SQL (Structured Query Language), pur non essendo un'estensione di quest'ultimo. È, infatti, possibile ottenere alcune funzionalità disponibili in MDX anche tramite SQL, sebbene non in modo altrettanto efficace e intuitivo.

Come un'interrogazione SQL, ogni interrogazione MDX comporta una richiesta di dati (la clausola SELECT), un'origine (la clausola FROM) e un filtro (la clausola WHERE). Queste parole chiave rappresentano, insieme ad altre, gli strumenti utilizzati per estrarre parti specifiche di dati da un cubo per l'analisi. MDX, come SQL, fornisce un DDL (Data Definition Language)

per gestire le strutture dati (non trattato in questa tesi in quanto non supportato da tutti gli strumenti OLAP. Per maggiori informazioni vedi bibliografia [18]).

La visualizzazione di un set di risultati di SQL è intuitiva. Si tratta di una griglia bidimensionale di colonne e righe. La visualizzazione di un set di risultati di MDX, invece, non è altrettanto intuitiva. Poiché un set di risultati multidimensionale può includere più di tre dimensioni e visualizzarne la struttura può essere estremamente difficile. Per fare riferimento a dati bidimensionali in SQL, il nome di una colonna e l'identificatore univoco di una riga vengono utilizzati, in base al metodo più appropriato per i dati, per fare riferimento a una singola cella di dati denominata campo. MDX utilizza invece una sintassi estremamente specifica e uniforme per fare riferimento alle celle di dati, indipendentemente dal fatto che i dati formino una singola cella o un gruppo di celle.

MDX offre inoltre una gamma completa di funzioni per la manipolazione dei dati recuperati nonché la possibilità di estendere il linguaggio MDX con funzioni definite dall'utente. Lo scopo delle espressioni multidimensionali MDX è semplificare e rendere più intuitivo l'accesso ai dati di più dimensioni (vedi bibliografia [17]).

I dati multidimensionali possono essere rappresentati in strutture con più di due dimensioni. Queste strutture sono chiamate cubi. Nelle intersezioni di singoli valori delle dimensioni, possono esserci più dati, chiamati *misure*.

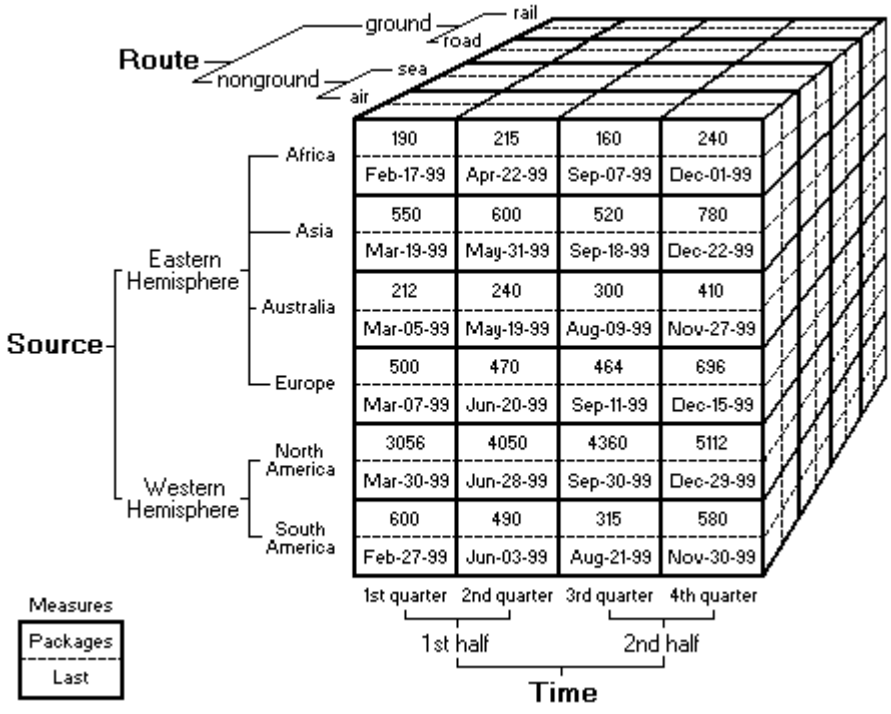


Figura 2 - Esempio di cubo

Nella figura precedente è illustrato un cubo in cui vengono utilizzate tre dimensioni, Route, Service e Time, e due *misure*, Packages e Last. Ogni dimensione è suddivisa in vari livelli, ognuno dei quali è ulteriormente suddiviso in membri. Ad esempio, la dimensione Source include il livello Eastern Hemisphere, suddiviso nei quattro membri Africa, Asia, Australia e Europa.

Come si può notare, l'esecuzione di un'interrogazione, anche su dati semplici, di un'origine dati multidimensionale può essere un'operazione complessa. Un cubo può, ad

esempio, includere più di tre dimensioni oppure anche una sola. I concetti di dimensione, livello, membro e *misura* sono importanti ai fini della comprensione della sintassi MDX.

Le dimensioni sono gli attributi strutturali dei cubi e, più specificatamente, sono gerarchie organizzate di livelli che descrivono i dati nella tabella dei fatti. Tutte le dimensioni si basano direttamente o indirettamente su tabelle. Quando si crea una dimensione da una tabella, è necessario selezionare le colonne che la definiscono. Le dimensioni sono gerarchiche e nella maggior parte dei casi i membri sono disposti in una configurazione a piramide. Il posizionamento orizzontale è determinato dai valori delle colonne sullo stesso livello nella *gerarchia* della dimensione, quello verticale dai valori delle colonne su livelli diversi nella *gerarchia* della dimensione.

Un livello è un elemento della *gerarchia* di una dimensione. I livelli descrivono la *gerarchia* dei dati, dal livello superiore (più riepilogato) al livello inferiore (più dettagliato). I livelli esistono solo all'interno delle dimensioni e ogni livello si basa su una colonna della relativa tabella dimensionale.

I livelli vengono definiti all'interno di una dimensione per specificare il contenuto e la struttura della *gerarchia* della dimensione. In altre parole, le definizioni dei livelli determinano i membri inclusi nella *gerarchia* e le posizioni relative dei membri (l'uno rispetto all'altro) all'interno della *gerarchia*.

Una *misura* è un set di valori basati su una colonna della tabella dei fatti del cubo e in genere è di tipo numerico. Le *misure*, inoltre, sono i valori di un cubo su cui si incentra l'analisi, ovvero rappresentano i dati numerici di principale interesse per gli utenti finali che esaminano un cubo. Le *misure* selezionate dipendono dai tipi d'informazioni richieste dagli utenti finali. Alcuni esempi di *misure* comuni sono le vendite, i costi, le spese e i volumi di produzione.

Una *misura* può essere ricavata da più colonne combinate in un'espressione. Ad esempio, la *misura* profitto risulta dalla sottrazione di due colonne numeriche: vendite e costi.

I membri calcolati possono essere utilizzati come *misure*. I valori dei membri calcolati vengono determinati tramite l'utilizzo di formule durante l'esplorazione del cubo, ma non sono memorizzati. I membri calcolati consentono pertanto di risparmiare spazio di archiviazione a scapito delle prestazioni. I membri calcolati vengono definiti tramite la valutazione di un'espressione MDX che si avvale dei concetti di celle, tuple e set.

Come SQL restituisce un subset di dati bidimensionali dalle tabelle, MDX restituisce un subset di dati multidimensionali dai cubi. La figura precedente raffigurante un cubo illustra che l'intersezione dei membri multidimensionali crea celle da cui è possibile ottenere dati. Per identificare ed estrarre tali dati, sia che si tratti di una singola cella sia che si tratti di un blocco di celle, MDX utilizza un sistema di riferimento denominato tupla. Le tuple elencano le dimensioni e i membri per identificare singole celle e sezioni più estese del cubo. Poiché ogni cella è un'intersezione di tutte le dimensioni del cubo, le tuple possono identificare in modo univoco ogni cella del cubo. Ad esempio, nella figura precedente la tupla:

```
(Source.[Eastern Hemisphere].Africa,
                            Time.[2nd half].[4th quarter],
                                                  Route.Air, Measures.Packages)
```

identifica una cella in cui il valore è 240.

La tupla identifica in modo univoco una sezione del cubo. Non è necessario che faccia riferimento a una cella specifica né che riguardi tutte le dimensioni di un cubo. Un insieme ordinato di tuple è denominato insieme. Di seguito è riportata una descrizione di un insieme di tuple del cubo illustrato nella figura precedente:

```
{ (Time.[1st half].[1st quarter]), (Time.[2nd half].[3rd quarter]) }
```

Inoltre, è possibile creare un set con nome. Un set con nome è un set con un alias, che semplifica la comprensione dell'interrogazione MDX e, se è particolarmente complessa, la relativa elaborazione.

I membri calcolati sono membri basati non su dati, ma su espressioni valutate in MDX, cioè un membro calcolato è un membro di una dimensione il cui valore viene calcolato in fase di esecuzione utilizzando un'espressione specificata dall'utente durante la sua definizione.

I membri calcolati possono anche essere definiti come *misure*. Solo le definizioni dei membri calcolati vengono memorizzate: i valori vengono calcolati in memoria quando necessario per rispondere a un'interrogazione. I membri calcolati consentono di aggiungere membri e *misure* a un cubo senza aumentarne la dimensione. Sebbene i membri calcolati debbano essere basati su dati, ad esempio membri già esistenti nel cubo, è possibile creare espressioni complesse combinando questi dati con operatori aritmetici, numerici e svariate funzioni.

Nel linguaggio MDX è disponibile una gamma completa di funzioni per la creazione di membri calcolati, che offrono un'estrema flessibilità nella manipolazione di dati multidimensionali (vedi bibliografia [17]).

2.2.2 CONFRONTO TRA SQL E MDX

La sintassi di MDX appare, a un primo sguardo, riconducibile alla sintassi di SQL. Per molti versi, le funzionalità fornite da MDX sono simili a quelle di SQL. Comunque, ci sono alcune vistose differenze tra SQL e MDX. Questo paragrafo è inteso come una guida alle differenze concettuali tra SQL e MDX dal punto di vista di uno sviluppatore SQL.

- La principale differenza tra SQL e MDX è l'abilità di MDX di referenziare dimensioni multiple. Anche se è possibile usare SQL per interrogare dei cubi, MDX fornisce comandi che sono stati progettati specificatamente per recuperare dati con un qualsiasi numero di dimensioni. SQL riferisce a solo due dimensioni, colonne e righe, quando processa una query. Visto che SQL è stato progettato per trattare solo dati a due dimensioni in forma tabellare, i termini "colonna" e "riga" hanno significato solo in questo contesto. MDX, al contrario, può processare dati con una, due, tre o più dimensioni in un'unica interrogazione. Visto che molteplici dimensioni possono essere usate in MDX, ogni dimensione viene inserita in un'asse. I termini "colonna" e "riga" vengono semplicemente usate come alias per i primi due assi. Inoltre ci sono altri assi che hanno un alias, ma il nome associato non ha un reale significato. MDX supporta questi alias per poterli utilizzare in fase di output. Purtroppo, molti strumenti di reportistica OLAP sono incapaci di visualizzare i risultati di un'interrogazione con più di due assi (come vedremo più avanti anche JPivot rientra in questa categoria).

- In SQL, la clausola SELECT è usata per definire il layout delle colonne per la query e la clausola WHERE è usata per definire i layout delle righe, mentre in MDX, la clausola SELECT viene usata per definire diversi assi dimensionali e la clausola WHERE viene usata per restringere i dati multidimensionali di una specifica dimensione o di un specifico membro.
- In SQL, la clausola WHERE è usata per filtrare i dati restituiti dalla query. In MDX, la clausola WHERE è usata per fornire uno "slice" dei dati restituiti. Anche se i due concetti sono simili, non sono equivalenti. In SQL, la clausola WHERE può contenere una lista arbitraria di elementi che possono essere o meno recuperati tra i risultati. In MDX, al contrario, il concetto di "slice" significa che ogni membro contenuto nella clausola WHERE identifica una diversa porzione dei dati. Vista la struttura dei dati multidimensionali, non è possibile richiedere uno "slice" di membri multipli di una stessa dimensione.
- Il processo di creazione di un'interrogazione SQL è differente dal processo di creazione di un'interrogazione MDX. Il creatore di una query SQL visualizza e definisce strutture a due dimensioni e scrive query che coinvolgono una o più tabelle. Al contrario, il creatore di una query MDX, usualmente visualizza e definisce strutture multidimensionali e scrive query che coinvolgono un solo cubo.
- La visualizzazione dei risultati di una query SQL, è intuitiva. La visualizzazione di un risultato di una query MDX, non è intuitiva visto che i risultati possono avere più di tre dimensioni.
- Infine SQL e MDX condividono una sintassi simile. La sintassi MDX è più robusta e può diventare più complessa.

2.2.3 QUERY BASE IN MDX

Un'interrogazione MDX base è strutturata fondamentalmente come segue:

```
SELECT [<axis_specification>
       [, <axis_specification>...]]
  FROM [<cube_specification>]
[WHERE [<slicer_specification>]]
```

In MDX, la clausola SELECT viene usata per specificare il dataset che contiene un sottoinsieme dei dati multidimensionali. Per specificare un dataset, una query MDX deve contenere informazioni riguardo.

- Il numero di assi. Nelle query MDX si possono specificare fino a 128 assi.
- I membri di ogni dimensione da includere in ciascun asse della query MDX.
- Il nome del cubo da usare come contesto della query MDX.
- I membri su cui eseguire operazioni di "slice".

Queste informazioni possono essere complesse. Il seguente esempio verrà usato per spiegare le varie parti di una query MDX base.

```
SELECT { [Measures].[Unit Sales], [Measures].[Store Sales] } ON COLUMNS,
   { [Time].[1997], [Time].[1998] } ON ROWS
```

```
FROM Sales
WHERE ( [Store].[USA].[CA] )
```

La query più semplice in MDX contiene la clausola SELECT, la clausola FROM e opzionalmente la clausola WHERE. La clausola SELECT determina quali sono gli assi della query. Nell'esempio sono stati definiti due assi, uno associato all'alias COLOMNS e uno associato all'alias ROWS (che corrispondono rispettivamente all'asse 1 e all'asse 2).

La clausola FROM determina quale sorgente dati multidimensionale viene usata per estrarre i dati per popolare il risultato della query. Nell'esempio precedente il cubo "Sales".

La clausola WHERE, opzionalmente, determina in quale dimensione o in quale membro usare un'operazione di "slice". Questo restringe l'estrazione dei dati. Nell'esempio precedente, la clausola WHERE riduce la dimensione "Store" ai soli prodotti immagazzinati in California.

Uno statement MDX supporta altre parole chiave opzionali, come WITH, e l'uso di funzioni MDX per costruire membri calcolati.

2.2.4 MEMBRI, TUPLE ED INSIEMI

Un membro è un elemento in una dimensione che rappresenta una o più occorrenze dei dati. Un membro è il livello più basso di referenza, quando si descrivono delle celle dei dati nei cubi. Per esempio, nella figura seguente è stato evidenziato la parte di cubo che rappresenta il membro Time.[2nd half].[3rd quarter].

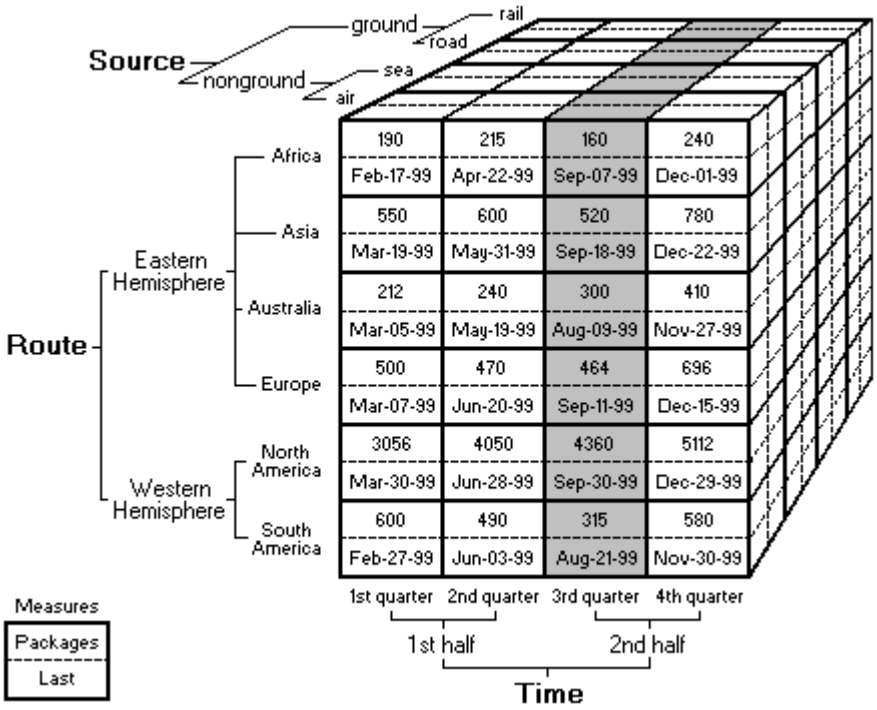


Figura 3 - Esempio di slice di un cubo

Le parentesi quadre ("[" e "]")sono utilizzate se il nome del membro contiene degli spazi bianchi o dei numeri. Comunque la dimensione "Time" è una parola unica, e le parentesi

quadre sono state utilizzate ugualmente. Il membro evidenziato nel diagramma può essere rappresentato da:

```
[Time].[2nd half].[4th quarter]
```

La parentesi quadra chiusa può venire usata come carattere di escape in MDX nel caso in cui il nome del membro contiene la parentesi quadra chiusa (la parentesi quadra aperta ha invece significato particolare solo come inizio di un nome di un membro).

```
[Premier [150]] 98]
```

Un membro può essere referenziato attraverso il suo nome o attraverso la sua chiave. Nell'esempio precedente, il membro è stato referenziato con il suo nome. In generale, il nome di un membro può essere duplicato in caso di dimensioni con nomi dei membri non univoci, o può cambiare in caso di cambiamento della dimensione.

Un metodo alternativo per riferire ai membri è utilizzare la loro chiave. La chiave di un membro viene usata dalla dimensione per specificare l'identità di un dato membro. Il carattere "e commerciale" ("&") è usato in MDX per differenziare una chiave di un membro dal suo nome, come nell'esempio che segue:

```
[Time].[2nd half].&[Q4]
```

In questo caso, la chiave del membro con nome "4th quarter" è "Q4". Referenziare un membro tramite la chiave assicura l'identificazione del membro anche in caso di cambiamento delle dimensioni e in caso di nomi di membri non univoci.

I membri possono anche essere creati, come parte di una query MDX, per restituire dati basati su espressioni valutate a tempo di esecuzione insieme ai dati memorizzati nel cubo da interrogare. Questi membri vengono chiamati membri calcolati e forniscono un punto di potenza e flessibilità di MDX. La parola chiave WITH viene usata nelle interrogazioni MDX per definire membri calcolati. Per esempio, se fossimo interessati a fornire una stima del "PackagesForecast" ottenuta come la *misura* dei "Packages" incrementata del 10%, possiamo creare un membro calcolato che fornisce questa informazione e la usa come un qualsiasi altro membro del cubo, come dimostrato nell'esempio seguente:

```
WITH MEMBER [Measures].[PackagesForecast] AS '[Measures].[Packages] * 1.1'
```

MDX fornisce un certo numero di funzioni per ricercare membri da altre entità MDX, come dimensioni e livelli, che permettono di non riferirsi esplicitamente al membro. Per esempio la funzione "FirstChild" permette di recuperare per tutti i membri di una dimensione o di un livello, il primo membro figlio. Ad esempio, per la dimensione Time, possiamo esplicitamente referenziare il primo membro come:

```
Time.[1st half]
```

o possiamo utilizzare la funzione "FirstChild" per referenziare lo stesso membro come mostrato nel seguente esempio:

```
Time.FirstChild
```

Una tupla viene usata per definire uno "slice" dei dati di un cubo. È composta da una collezione ordinata di membri appartenenti a una o più dimensioni. Una tupla viene usata per

specificare sezioni dei dati multidimensionali dal cubo. Una tupla composta da un membro per ogni dimensione del cubo identifica una singola cella nel cubo. Da un punto di vista alternativo, una tupla è un vettore di membri. I seguenti diagrammi illustrano alcuni esempi di tuple.

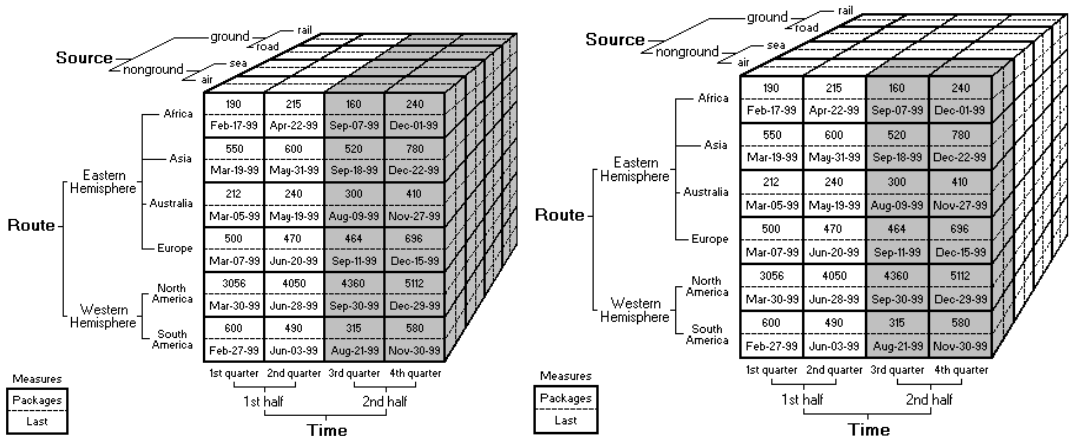


Figura 4 - Esempio di tuple

Il primo cubo rappresenta la tupla (Time.[2nd half]) mentre il secondo la tupla (Time.[2nd half], Route.nonground.air).

In MDX, le tuple sono costruite con modalità diverse a seconda della loro complessità. Se una tupla è composta da un singolo membro di una singola dimensione, spesso chiamata tupla semplice, la sintassi seguente è accettabile:

```
Time.[2nd half]
```

Se una tupla è composta di membri appartenenti a più dimensioni, la tupla deve essere rappresentata racchiusa tra parentesi tonde, come mostrato nell'esempio seguente.

```
(Time.[2nd half], Route.nonground.air)
```

Una tupla composta di un singolo membro può essere anche inclusa tra le parentesi, ma non è necessario. Le tuple sono spesso raggruppate in insiemi per essere usate nelle interrogazioni MDX. Ci sono molte funzioni MDX che ritornano tuple che possono essere usate ovunque dove queste ultime sono accettate.

Una tupla può comprendere membri di dimensioni diverse oppure più membri provenienti dalla stessa dimensione. Il termine dimensionalità viene usato per indicare le dimensioni descritte dai membri contenuti in una tupla.

Un insieme è una collezione ordinata di zero, una o più tuple. Un insieme è comunemente usato per definire gli assi e le operazione di "slide". L'esempio riportato qui di seguito mostra un insieme formato da due tuple.

```
{ (Time.[1st half], Route.nonground.air),
                                    (Time.[2nd half], Route.nonground.sea) }
```

Un insieme può contenere più di un'occorrenza della stessa tupla. Il seguente insieme è accettabile.

```
{ Time.[2nd half], Time.[2nd half] }
```

Un insieme riferisce all'intero insieme delle possibili combinazioni di membri, ottenuto combinando le tuple o i valori contenuti delle celle che le tuple rappresentano, a seconda del contesto di utilizzo. Una considerazione importante è che un insieme composto da una singola tupla non è una tupla, viene interpretato come un insieme da MDX. Certe funzioni MDX accettano tuple come parametri e generano un errore se un insieme formato da una singola tupla gli viene passato come parametro. Quindi, tuple e insiemi formati da una singola tupla non sono intercambiabili.

L'esplicito elenco di tuple non è l'unico modo per avere un insieme. MDX supporta una grande varietà di funzioni che ritornano un insieme. Ne è un esempio, l'operatore di due punti (":") che permette di usare il naturale ordinamento dei membri per creare un insieme. Per esempio, la seguente dichiarazione è un insieme.

```
{[1st quarter]:[4th quarter]}
```

La dichiarazione precedente costruisce lo stesso insieme della seguente.

```
{[1st quarter], [2nd quarter], [3rd quarter], [4th quarter]}
```

L'operatore di due punti è una funzione inclusiva, i membri dai due lati dell'operatore sono inclusi nei risultati restituiti.

Altre funzioni MDX che ritornano degli insiemi possono essere utilizzate come parte di un'espressione i cui elementi sono separati da una virgola. Per esempio, le seguenti espressioni MDX sono valide:

```
{Time.Children}
{Time.Children, Route.nonground.air}
{Time.Children, Route.nonground.air, Source.Children}
```

Come le tuple, gli insiemi possono avere dimensionalità. Un insieme è composto di tuple, quindi la dimensionalità dell'insieme è espressa dalle dimensionalità di ogni tupla contenuta nell'insieme. Per questo motivo, le tuple di un insieme devono avere la stessa dimensionalità. In altre parole, questo esempio non funziona:

```
{ (Time.[2nd half], Route.nonground.air),
                                    (Route.nonground.air, Time.[2nd half]) }
```

Come si può notare l'ordine delle tuple in un insieme è importante. Influisce, ad esempio, nell'ordine di innesto delle dimensioni all'interno di un'asse. La prima tupla rappresenta la prima dimensione, più esterna, la seconda quella immediatamente più all'interno e così via.

Un insieme con nome è un insieme per cui è stato creato un alias. Gli insiemi con nome sono usati molto comunemente nelle interrogazioni MDX complesse per scriverla facilmente e migliorare la manutenzione.

2.2.5 ASSI E DIMENSIONI "SLICE"

Quando si formula una query MDX, un'applicazione tipicamente guarda il cubo e divide gli insiemi di dimensioni in due sottoinsiemi:

- dimensioni degli assi: per le quali i dati vengono recuperati per membri multipli;
- dimensioni di "slice": per i quali i dati vengono recuperati per un membro singolo.

Visto che le dimensioni degli assi e degli "slice" possono essere costruiti da molteplici dimensioni del cubo che possono essere interrogate, questi termini sono usati per differenziare le dimensioni impiegate dal cubo che possono essere interrogate dalle dimensioni create nel cubo ritornato dalla query MDX.

Per esempio, assumiamo che esista un cubo chiamato "TestCube" con due dimensioni chiamate "Route" e "Time". Visto che le *misure* del cubo sono parte della dimensione "Measures", questo cubo ha tre dimensioni in totale. La query fornisce la matrice con la *misura* "Packages" comparata alle dimensioni "Route" e "Time".

Nella seguente query MDX, le dimensioni di "Route" e "Time" vengono usate come dimensioni degli assi e la dimensione delle *misure* come dimensione di "slice". La funzione "Members" indica che i membri della dimensione o del livello devono essere usati per costruire un insieme, senza elencare esplicitamente tutti i membri delle dimensioni date.

```
SELECT { Route.nonground.Members } ON COLUMNS,
       { Time.[1st half].Members } ON ROWS
FROM TestCube
WHERE ( [Measures].[Packages] )
```

La griglia risultante di valori può essere rappresentata come una tabella che contiene la *misura* dei "Packeges" a ogni intersezione degli assi dimensionali "columns" e "rows".

MDX valuta gli assi e le dimensioni di "slice" prima, costruisce la struttura dei risultati e poi recupera le informazioni dal cubo che è stato interrogato. Le dimensioni di "slice" sono simili alle dimensioni degli assi in questo ambito, ma hanno delle limitazioni che non condividono con le dimensioni degli assi.

Le dimensioni degli assi determinano gli spigoli dell'insieme multidimensionale dei risultati. MDX usa la clausola SELECT per specificare le dimensioni degli assi da assegnare a un particolare asse. Nella sintassi seguente, ogni tag <axis_specification> definisce la dimensione di un'asse. Il numero di assi nel dataset è uguale al numero di <axis_specification> contenuti nella query. Una query MDX può supportare fino a 128 specificazioni di assi, ma pochissime interrogazioni MDX usano più di 5 assi (la maggior parte 2).

```
<axis_specification> ::= <set> ON <axis_name>
<axis_name> ::= COLUMNS | ROWS | PAGES |
                                        SECTIONS | CHAPTERS | AXIS(<index>)
```

Ogni dimensione di asse è associata a un numero: 1 per l'asse x, 2 per l'asse y, 3 per l'asse z e così via. Il valore indicato come <index> è il numero dell'asse. Per i primi 5 assi esistono degli alias che sono rispettivamente COLUMNS, ROWS, PAGES, SECTIONS e CHAPTERS. Una query MDX non può saltare degli assi. Per questo, una query che include uno o più <axis_name> non può escludere gli assi precedenti o intermedi. Per esempio, una query non può avere l'asse ROWS senza avere l'asse COLUMNS o avere COLUMNS e PAGES senza avere ROWS.

Comunque, è possibile specificare una clausola SELECT senza assi (una clausola SELECT vuota). In questo caso, tutte le dimensioni sono di "slice" e la query MDX restituisce una sola cella.

2.2.6 MEMBERS

Selezionare i membri di una dimensione, di una *gerarchia* o di un livello è il punto di partenza più comune per altre operazioni. L'operatore .Member prende la dimensione, la *gerarchia* o il livello specificato alla sua sinistra, e i suo risultato è l'insieme di tutti i membri associati allo scopo specificato. Per esempio, il risultato di [Customer].Members sarà l'insieme di tutti i Customer, mentre il risultato di [Product].[Product Category].Members restituisce tutti i membri del livello Product Category nella dimensione Product. Per esempio, la query

```
SELECT { [Scenario].Members } ON COLUMNS,
       { [Store].Members } ON ROWS
FROM Budgeting
```

visualizza tutti i membri della dimensione Scenario lungo le colonne e tutti i membri della dimensione Store lungo le righe.

2.2.7 CROSSJOIN()

In molti casi può essere utile a considerare il prodotto cartesiano di membri (o tuple) di due differenti insiemi. La funzione CrossJoin() è la via più diretta per ottenere tutte le possibili combinazioni dei membri di due insiemi. Per esempio, se volessimo visualizzare sulle colonne di una query tutte le categorie di Product per ogni mese dell'anno 1999, per generare l'insieme in questione si può utilizzare la seguente espressione:

```
CrossJoin({[Time].[Jan 1999]:[Time].[Dec 1999]},
                                    {[Product].[Product Category].Members})
```

La funzione CrossJoin() prende in input solo due insiemi. Per calcolare il prodotto cartesiano tra più insiemi, come per esempio Scenario, Time e Product è possibile innestare più funzioni CrossJoin() in uno dei seguenti modi:

```
CrossJoin([Time].Members, CrossJoin([Scenario].Members, [Product].Members))
CrossJoin(CrossJoin([Time].Members, [Scenario].Members), [Product].Members)
```

La funzione CrossJoin() è standard in MDX. In alcuni OLAP server la funzione ha una tecnica non standard per ottenere la stessa cosa:

```
{ [Time].Members } * { [Scenario].Members } * { [Product].Members }
```

Uno degli usi più comuni della funzione CrossJoin() è di combinare un singolo membro di una dimensione con un insieme di membri di un'altra dimensione, o di creare un insieme in cui una particolare *misura* viene riferita da un insieme di tuple di un'altra dimensione. Quando una formula di una *misura* calcolata include il numero di celle non vuote di un'altra *misura*, questo costrutto è necessario. Comunque è sempre preferibile quando è necessario costruire tuple su dimensioni multiple utilizzare il costruttore di range. Per esempio, per esprimere l'intervallo "I toothpaste negli store da 1 a 10", si può scrivere un'espressione come la seguente:

```
([Product].[Toothpaste],{[Geography].[Store 1]:[Geography].[Store 10]})
```

La stessa espressione può essere scritta con l'uso della funzione di CrossJoin(), come segue:

```
CrossJoin({[Product].[Toothpaste]},
                              [Geography].[Store 1]:[Geography].[Store 10])
```

2.2.8 FILTER()

Operatori come CrossJoin() e ":" aiutano a costruire insiemi. Al contrario, Filter() riduce i membri contenuti a un insieme includendo nel risultato solo gli elementi che verificano alcuni criteri. La funzione Filter() prende come parametri di ingresso un'espressione insieme e un'espressione booleana e ritorna il sottoinsieme dell'insieme indicato come primo parametro i cui membri soddisfano l'espressione booleana indicata come secondo parametro. Per esempio l'espressione:

```
Filter({[Product].[Product Category].Members}, [Measures].[Sales]>=500)
```

ritorna l'insieme di tutti i membri delle categorie di prodotti che hanno associato un valore della *misura* vendite maggiore di 500. Un'espressione booleana qualunque può essere usata per filtrare gli insiemi. Per esempio, l'espressione:

```
Filter({[Product].[Product Category].Members},
               ([Measures].[Sales] >= 1.2 * [Measures].[Costs])
                                          AND [Measures].[Sales] >=150)
```

restituisce l'insieme di tutte le categorie di prodotti che hanno associate delle vendite maggiore di 150 e delle vendite maggiore del 120% dei costi.

La funzione Filter() lavora su un insieme generico, non solo su insiemi con membri di una dimensione specifica, quindi la seguente espressione ritorna l'insieme di tutte le tuple (Categorie di prodotti e Città) che hanno associato una quantità di vendite maggiore di 500:

```
 Filter( CrossJoin([Product].[Product Category].Members,
                          [Store].[City].Members), [Measures].[Sales]>=500)
```

Per determinare il valore delle vendite associate a ciascuna categoria di prodotti o di ogni tupla (Categoria di prodotti, Città), bisogna tenere in considerazione le atre dimensioni associate con la *misura* delle vendite. Per esempio, nel primo dei tre esempi di uso della funzione Filter() non è stato considerata la dimensione temporale a cui sono associate le vendite. Per specificare tutte le dimensioni addizionali che sono necessarie basta introdurre un insieme nell'espressione booleana. Per esempio, per specificare che si è interessati alle vendite dell'anno 2000 a Baton Rouge, possiamo scrivere:

```
Filter({[Product].[Product Category].Members},
        ([Measures].[Sales],[Time].[2000], [Store][Baton Rouge, LA) >=500)
```

In un modo più avanzato possiamo esprimere la stessa condizione come segue:

```
Filter( CrossJoin({[Time].[2000], [Store][Baton Rouge, LA) },
           [Product].[Product Category].Members),[Measures].[Sales] >=500)
```

La differenza tra i due metodi consiste nell'insieme che restituiscono come risultato. Infatti, nel primo caso verrà restituito un insieme di categorie di prodotti, mentre nel secondo caso verrà restituito un insieme di tuple del tipo(Anno, Luogo, Categoria di prodotto). La differenza riguarda solo il formato dei risultati, in quanto nel secondo caso, anno sarà sempre

uguale a 2000, luogo sempre uguale a Baton Rouge e le categorie di prodotti saranno le stesse contenute nel risultato della prima modalità.

2.2.9 ORDER()

Per mettere le tuple di un insieme in un determinato ordine basato sui valori ad esso associati, è necessario usare la funzione Order(). La funzione Order() prende come parametri un insieme, un criterio di ordinamento e un flag opzionale che indica quale principio di ordinamento (crescente o decrescente e includere o ignorare relazioni gerarchiche tra le tuple). La funzione Order() ritorna un insieme che consiste nei membri dell'insieme indicato come primo parametro in un nuovo ordine. Per esempio, dato l'insieme delle categorie di prodotti, e volendo ordinarli in ordine decrescente secondo il profitto realizzato nell'anno 2000, possiamo scrivere la seguente espressione:

```
Order([Product].[Product Category].Members, ([Measures].[Profit],
                          [Time].[2000], [Customer][All Customers]), BDESC)
```

Visto che la funzione Order() lavora sulle tuple, possiamo ordinare le combinazioni di dimensioni a cui siamo interessati a seconda di qualche *misura*. Per esempio, la seguente espressione ordina ogni tupla del tipo (Categoria di prodotto, Città) a seconda del profitto associato.

```
Order( Filter( CrossJoin(
   [Product].[Product Category].Members,[Store].[City]. Members),
   [Measures].[Sales] >= 500),
   ([Measures].[Profit],[Time].[2000],[Customer][All Customers]),
   BDESC)
```

2.3 MONDRIAN: UN OLAP SERVER OPEN SOURCE

2.3.1 CHE COS'È MONDRIAN?

Mondrian e' un motore per On Line Analytical Processing (OLAP) sviluppato in Java. Implementa le funzionalità indispensabili all'analisi dati (aggregazione, drill-down drill-through, slicing, dicing) ed e' in grado di eseguire query espresse in MDX leggendo i dati da un RDBMS e presentando i risultati in forma multidimensionale per mezzo di una API Java.

La connessione alla base di dati di Data Warehouse avviene via JDBC, il che rende indipendente Mondrian dal particolare RDBMS utilizzato. Lo schema multidimensionale della base dati può essere sia a stella che a ficco di neve, e la sua descrizione viene fornita al motore sotto forma di un file XML.

Mondrian e' progettato per delegare al DBMS tutte le funzionalità che questo e' in grado di eseguire al meglio, in particolare l'aggregazione e l'utilizzo, ove consentito, di viste materializzate per ottimizzare la velocità di risposta.

Le raffinate strategie di caching consentono buone prestazioni in termini di velocità di esecuzione. Mondrian è un componente della Pentaho BI Platform.

2.3.2 LIVELLI DEL SISTEMA MONDRIAN

Il sistema OLAP Mondrian consiste in quattro livelli; cominciando da quello più vicino all'utente abbiamo:

- presentation layer;
- dimensional layer;
- star layer;
- storage layer.

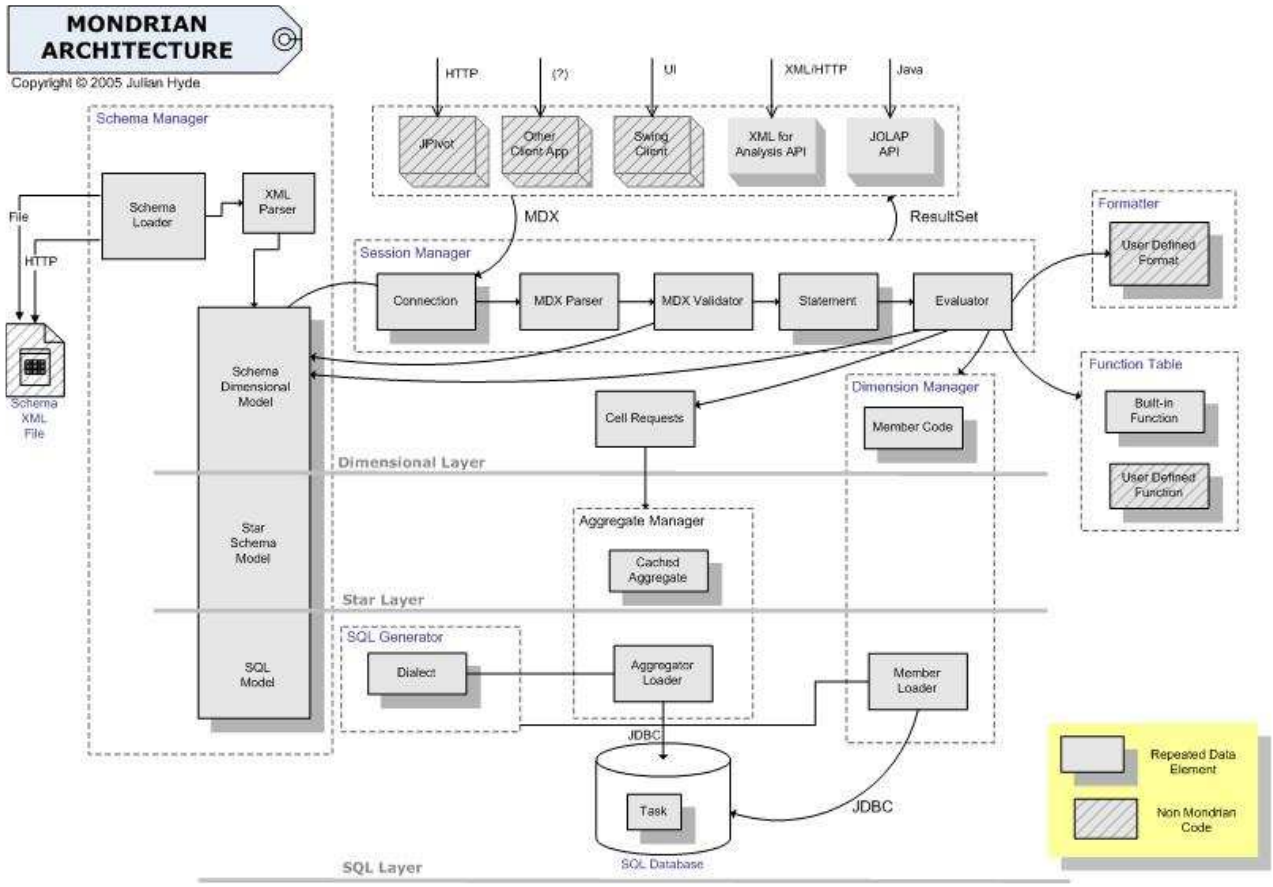


Figura 5 - Architettura del server OLAP Mondrian

Il livello di presentazione determina cosa vede l'utente finale sul suo monitor e come può interagire per formulare nuove domande. Ci sono molti modi per rappresentare dati multidimensionali come ad esempio tabelle con pivoting, istogrammi, grafici a torta, ecc… e modi avanzati come mappe cliccabili e grafici dinamici. Quello che tutte queste

rappresentazioni hanno in comune è la grammatica multidimensionale di dimensioni, *misure* e celle in base alle quali il livello di presentazione formula le domande all'OLAP server.

Il secondo livello è il dimensional layer. Questo livello analizza, valida ed esegue un'interrogazione MDX. Una query viene valutata in molteplici passi. Prima vengono valutati gli assi, poi i valori delle celle all'interno degli assi. Il query transformer permette all'applicazione di manipolare query esistenti per costruire uno statement MDX invece di ripartire da zero per ogni richiesta. I *metadati* descrivono il modello dimensionale e come viene mappato nel modello relazionale.

Il terzo livello è lo star layer ed è responsabile del mantenimento di una cache di aggregazioni. Un'aggregazione è un insieme di valori (celle) in memoria, qualificate da un insieme di colonne con il valore delle dimensioni. Il dimensional layer manda una richiesta per un insieme di celle. Se le celle richieste non sono contenute nella cache, l'aggregation manager invia una richiesta allo storage layer.

Lo storage layer è un RDBMS. È responsabile di fornire aggregazioni di celle e i membri delle tabelle dimensionali. Questo componente può essere installato sulla stessa macchina o essere distribuito tra più macchine. Il livello 2 e 3, invece, compongono il Mondrian server e devono essere sulla stessa macchina. Lo storage layer può essere installato su un'altra macchina e accessibile tramite una connessione remota JDBC. In un sistema multi-utente il livello di presentazione può esistere su ognuna delle macchine degli end-user.

2.3.3 STRATEGIE DI MEMORIZZAZIONE ED AGGREGAZIONE

Un OLAP server può generalmente essere catalogato come appartenete ad uno di questi tipi:

- MOLAP (Multidimensional OLAP): il server memorizza tutti i suoi dati su disco in strutture dati ottimizzate per l'accesso multidimensionale. Tipicamente i dati sono memorizzati in array densi che richiedono solo 4 o 8 byte per cella.
- ROLAP (Relational OLAP): il server memorizza i dati in un database relazionale. Ogni riga nella fact table ha una colonna per ogni dimensione e per ogni *misura*.

Tre tipi di dati devono essere memorizzati: i dati della fact table, gli aggregati e le dimensioni. I database MOLAP memorizzano i dati della fact table in formato multidimensionale, ma se hanno molte dimensioni i dati risultano sparsi e non si hanno buone prestazioni. Un sistema HOLAP (Hybrid OLAP) risolve i problemi lasciando i dati più spezzettati nel database relazionale, ma memorizza le aggregazioni in formato multidimensionale.

Aggregati pre-computati sono necessari per grandi insiemi di dati, altrimenti certe query non potrebbero essere risolte senza leggere l'intero contenuto della fact table. Aggregazioni MOLAP sono spesso un'immagine delle strutture dati costruite in memoria, divise in pagine e memorizzate su disco, Aggregati ROLAP sono memorizzati in tabelle. In alcuni sistemi ROLAP questi sono esplicitamente gestiti dall'OLAP server, in altri sistemi, le tabelle sono dichiarate come viste materializzate e sono usate quando l'OLAP server processa query con l'esatta combinazione di colonne nella clausola group by.

Il componente finale delle strategie di aggregazione è la cache. La cache mantiene le aggregazioni pre-elaborate in memoria quindi le query seguenti potranno accedere alle celle senza accedere al disco. Se la cache mantiene i dati richiesti a un livello più basso di aggregazione, si possono elaborare i dati richiesti con una operazione di roll-up.

La cache è, probabilmente, la parte più importante della strategia di aggregazione perché è adattativa. È difficile scegliere un insieme di aggregazioni pre-elaborate che diano uno speed-up al sistema senza usare enormi quantità di disco, particolarmente quando ci sono molte dimensioni o se l'utente richiede aggregati non predicibili. Un dimensionamento ragionevole della cache permette al sistema di comportarsi adeguatamente in caso di query non predicibili con alcuni o nessun aggregato pre-elaborato.

La strategia di aggregazione usata da Mondrian è la seguente:

- la fact table è memorizzata nel DBMS;
- per aggregare i dati viene utilizzata la clausola group by del SQL;
- se il RDBMS supporta viste materializzate e l'amministratore del database sceglie di creare viste materializzate per particolari aggregazioni, allora Mondrian usa esse per semplicità. Idealmente il manager delle aggregazioni di Mondrian sarà consapevole delle viste materializzate e quale di queste, a seconda delle particolari aggregazioni, è più economica.

2.3.4 API

Mondrian fornisce delle API per permettere alle applicazioni lato client di eseguire le query. Visto che non ci sono API universalmente accettate per eseguire query OLAP, le API di Mondrian sono le prime realizzate propriamente per lo scopo. Comunque, chiunque ha usato le connessioni JDBC le troverà familiari. La differenza principale è il linguaggio per la query. Mondrian usa MDX per specificare le query dove JDBC usa SQL. Il seguente frammento di codice Java si connette a Mondrian, esegue una query e stampa i risultati.

```
import mondrian.olap.*;
import java.io.PrintWriter;

Connection connection =
       DriverManager.getConnection("Provider=mondrian;" +
                "Jdbc=jdbc:odbc:MondrianFoodMart;" +
                "Catalog=/WEB-INF/FoodMart.xml;",null,false);
Query query = connection.parseQuery("SELECT "+
  "{[Measures].[Unit Sales],[Measures].[Store Sales]} on  columns," +
  "{[Product].children} on rows FROM [Sales] " +
  "WHERE ([Time].[1997].[Q1], [Store].[CA].[San Francisco])");
Result result = connection.execute(query);
result.print(new PrintWriter(System.out));
```

Una Connection è creata via DriverManager, con una strada simile a JDBC. Una query, analogamente a uno Statement con JDBC, viene creata facendo il parsing di una stringa MDX. L'oggetto Result è analogo al ResultSet per JDBC, tranne che per il fatto che tratta dati multidimensionali. Questi consistono di assi e celle, invece di colonne e righe.

Per completare con standard emergenti, sono state aggiunte due API a Mondrian:

- JOLAP è uno standard emergente per il processo JSR ed è diventata parte di J2EE;
- XML for Analysis è uno standard per accedere a un OLAP server via SOAP (Simple Object Access Protocol). Esso vuole permettere a un componente non-Java come Excel di eseguire query con Mondrian.

2.3.5 PERFORMANCE DI MONDRIAN

Come in tutti i progetti di data warehouse, un aspetto da tenere sempre presente è quello delle prestazioni. Mondrian ha tra i suoi scopi quello di avere buone prestazioni, basandosi sulla propria architettura e mantenendo l'obbiettivo d'essere multi-piattaforma.

Il processo di settaggio delle performance di Mondrian è una combinazione di configurazioni che riguardano la progettazione, l'hardware, il database e altre configurazioni. Per cubi molto grandi, le performance sono guidate principalmente dall'hardware, dal sistema operativo e dalla configurazione del database piuttosto che da Mondrian.

Per quello che riguarda Mondrian, le attenzioni da considerare per migliorare le prestazioni sono:

- avere la responsabilità del disegno fisico del database, rispettando i requisiti del data warehouse e del specifico *data mart*;
- progettare l'applicazione in modo efficiente:
  - separare l'ambiente d'esecuzione di Mondrian dal DBMS;
  - se possibile, separare i processi dedicati all'interfaccia utente dall'ambiente in cui Mondrian fa caching;
- avere un hardware adeguato al DBMS;
- configurare il sistema operativo al meglio per il DBMS;
- aggiungere viste materializzate o tabelle di aggregati per supportare specifiche interrogazioni MDX;
- configurare il DBMS con indici sia nelle tabelle delle dimensioni che nella fact table;
- configurare la cache di Mondrian. Più è grande meglio è.

Una parte importante del processo di configurazione del database, consiste nell'abilitare le funzioni di "SQL Tracing" e di scrittura dei log file. Queste funzioni sono di aiuto per sapere quali interrogazioni SQL vengono eseguite e quanto tempo impiegano a essere eseguite. Con questi dati si può intervenire sulla configurazione del database e si reitera il processo.

Per migliorare le prestazioni, si consiglia in particolar modo di:

- creare indici sulle chiavi primarie e sulle chiavi esterne delle tabelle;
- assicurarsi che le colonne siano dichiarate come "NOT NULL" dove possibile;
- se la tabella ha una chiave primaria composta, sperimentare l'effetto che producono indici su differenti parti della chiave. Ad esempio, se la chiave primaria è (a, b, c), creare una chiave unica su questi tre attributi e crearne altre due non uniche su (b, c) e (c, a).

Mondrian, allo stato attuale usa l'espressione 'count(distinct …)' nelle query per calcolare la cardinalità delle dimensioni e dei livelli, oltre che per le *misure* che hanno la funzione count come aggregatore. Per questi campi, un indice può migliorare le prestazioni delle interrogazioni,

anche se il miglioramento dipende dal DBMS utilizzato e quale strategia utilizza per ottimizzare la funzione count.

Il modo migliore per migliorare le prestazioni di Mondrian è di costruire un insieme di tabelle di aggregati che coesistono con la fact table. Queste tabelle di aggregati contengono *misure* preaggregate costruite a partire dalla fact table.

Alcuni DBMS, come Oracle, possono creare automaticamente queste aggregazioni e materializzare alcune viste e mantenere sincronizzate le tabelle e le viste create. Altrimenti, sarà il progettista a dover mantenere le tabelle di aggregati attraverso una serie di operazioni di INSERT durante il processo di aggiornamento del data warehouse.

Non è affatto semplice scegliere le tabelle giuste da aggregare. Per ogni fact table ci sono molte possibili aggregazioni. Ad esempio, se consideriamo sei dimensioni e tre livelli per ogni dimensione possiamo avere 64 = 1296 possibili aggregazioni. Se il progettista aggiunge una tabella di aggregati, Mondrian può utilizzarla per migliorare le performance.

Se Mondrian non trova tabelle di aggregati non è un problema. Scegliere le tabelle di aggregati da creare è una parte importante del processo di configurazione delle performance. Il processo di configurazione delle performance, è un processo iterativo che si conclude quando il progettista è soddisfatto delle prestazioni che ha ottenuto. I passi da seguire sono:

- scegliere alcune interrogazioni che rappresentano le richieste tipiche di un utente finale;
- eseguire l'insieme delle interrogazioni di esempio e prendere nota del tempo che impiegano per essere eseguite. Rieseguire nuovamente le interrogazioni e notare l'effetto che produce la cache;
- se le performance non sono ancora accettabili e i dati sono molti è probabile che siano necessarie delle tabelle di aggregati;
- decidere le tabelle di aggregati da creare. Se è attiva la funzione di "SQL tracing", guardare le clausole di GROUP BY eseguite può aiutare nella decisione;
- registrare le tabelle di aggregati all'interno del catalogo, creare la tabella all'interno del database, popolare la tabella con i dati e creare gli indici;
- riavviare Mondrian per svuotare la cache e per rileggere lo schema, e ripetere i passi sopra descritti.

Riguardo alla problematica della selezione delle viste da materializzare sarà proposto più avanti (paragrafo 2.7.1) un algoritmo per risolvere il problema.

2.3.6 TABELLE DEGLI AGGREGATI IN MONDRIAN

Al contrario di molti server OLAP, Mondrian non memorizza dati sul disco, lavora solo con i dati del database relazionale e, quando ha letto un po' di dati, li memorizza nella cache. Questo garantisce semplificazioni nell'installazione di Mondrian, ma limita le sue prestazioni quando viene utilizzato con grandi moli di dati.

Consideriamo cosa succede quando un utente richiede il report delle vendite di quest'anno. Questo report contiene un unico dato: il totale delle vendite, di tutti i prodotti, in

tutte le regioni durante l'anno 2005. Per prima cosa, per poter reperire questo numero, Mondrian genera una query come la seguente:

```
SELECT sum(store_sales)
FROM sales_fact, time
WHERE sales_fact.time_id = time.time_id
AND time.year = 2005
```

e la invia al DBMS. Il DBMS impiega alcuni minuti per eseguirla visto che deve andare a leggere milioni di record dalla fact table e aggregarli in una singola cella. Chiaramente, in questo caso e in quelli simili è necessario un preaggregato dei dati.

Una tabella degli aggregati coesiste con la fact table, e contiene *misure* preaggregate costruite a partire dalla fact table. Viene registrata nello schema utilizzato da Mondrian, in modo che Mondrian possa scegliere quando usare la fact table e quando la tabella degli aggregati, a seconda della particolare interrogazione da eseguire.

La progettazione delle tabelle degli aggregati è un'arte molto particolare. Esistono diverse ricerche, sia empiriche che teoriche, che trattano come strutturare le tabelle degli aggregati.

Per spiegare cosa sono le tabelle degli aggregati consideriamo il semplice schema a stella riportato nella figura successiva.

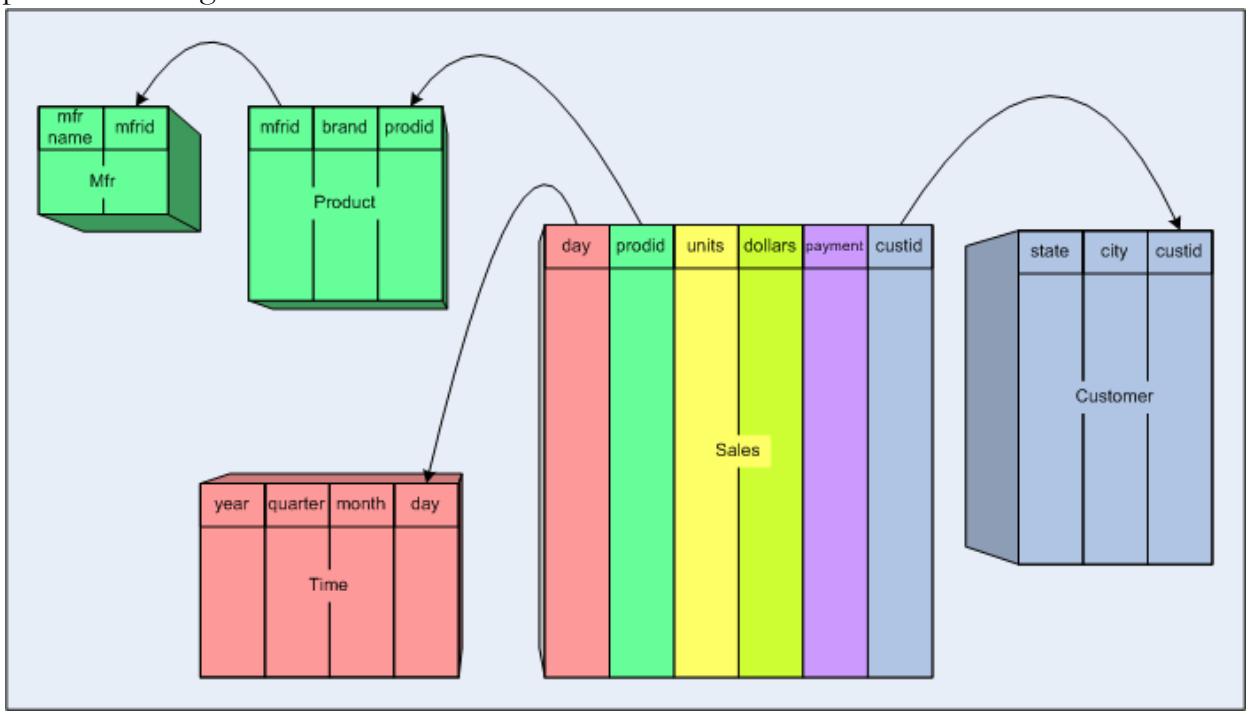


Figura 6 - Schema a stella di riferimento per le tabelle di aggregati

Lo schema a stella ha una singola fact table chiamata Sales con due *misure* (unit e dollars) e quattro tabelle dimensionali (Product, Mfr, Customer, Time e Customer). Utilizziamo questo schema per creare il seguente *modello multidimensionale*:

- il cubo [Sales] ha due *misure* [Unit sales] e [Dollar sales];
- la dimensione [Product] ha i livelli [All Products], [Manufacturer], [Brand], [Prodid];
- la dimensione [Time] ha i livelli [All Time], [Year], [Quarter], [Month], [Day];
- la dimensione [Customer] ha i livelli [All Customers], [State], [City], [Custid];

- la dimensione [Payment Method] ha i livelli [All Payment Methods], [Payment Method].

Da notare che la dimensione [Product] è una dimensione a fiocco di neve (che è sparsa tra le due tabelle Product e Mfr) e che la dimensione [Payment Method] è contenuta all'interno della fact table.

Consideriamo il primo esempio di tabella di aggregazione come mostrato in figura:

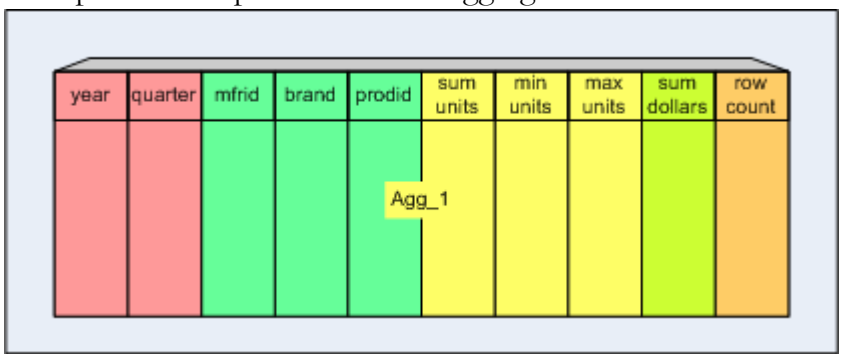


Figura 7 - Primo esempio di tabella degli aggregati

La tabella contiene le colonne dello schema originale che sono state combinate in un'unica tabella.

- La dimensione [Time] è stata collassata nella tabella degli aggregati omettendo le colonne del mese e del giorno.
- Le due tabelle della dimensione [Product] sono state collassate nella tabella degli aggregati.
- La dimensione [Customer] viene persa.
- Per ogni *misura* della fact table di partenza, ci sono più colonne di *misure* nella tabella degli aggregati (sum units, min units, max units, sum dollars).
- È stata aggiunta la colonna della *misura* [row count] che rappresenta il numero di righe della fact table di partenza che sono state collassate in quella riga della tabella degli aggregati.

Questa aggregazione può essere dichiarata in Mondrian come segue:

```
<Cube name="Sales">
  <Table name="sales">
    <AggName name="agg_1">
      <AggFactCount column="row count"/>
       <AggMeasure name="[Measures].[Unit Sales]" column="sum units"/>
       <AggMeasure name="[Measures].[Min Units]" column="min units"/>
       <AggMeasure name="[Measures].[Max Units]" column="max units"/>
       <AggMeasure name="[Measures].[Dollar Sales]" column="sum dollars"/>
       <AggLevel name="[Time].[Year]" column="year"/>
       <AggLevel name="[Time].[Quarter]" column="quarter"/>
      <AggLevel name="[Product].[Mfrid]" column="mfrid"/>
       <AggLevel name="[Product].[Brand]" column="brand"/>
       <AggLevel name="[Product].[Prodid]" column="prodid"/>
    </AggName>
  </Table>
  <!-- Rest of the cube definition -->
</Cube>
```

Un secondo esempio di tabella degli aggregati può essere il seguente:

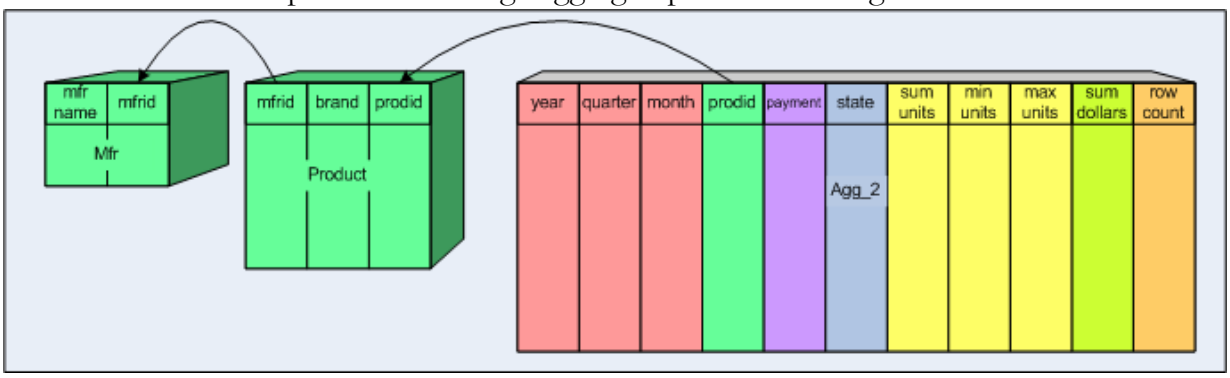


Figura 8 - Secondo esempio di tabella degli aggregati

Corrisponde alla seguente dichiarazione:

```
<Cube name="Sales">
  <Table name="sales">
    <AggName name="agg_1" ... />
    <AggName name="agg_2">
      <AggFactCount column="row count"/>
      <AggForeignKey factColumn="prodid" aggColumn="prodid"/>
      <AggMeasure name="[Measures].[Unit Sales]" column="sum units"/>
      <AggMeasure name="[Measures].[Min Units]" column="min units"/>
      <AggMeasure name="[Measures].[Max Units]" column="max units"/>
      <AggMeasure name="[Measures].[Dollar Sales]" column="sum dollars"/>
      <AggLevel name="[Time].[Year]" column="year"/>
      <AggLevel name="[Time].[Quarter]" column="quarter"/>
      <AggLevel name="[Time].[Month]" column="month"/>
      <AggLevel name="[Payment Method].[Payment Method]" column="payment"/>
      <AggLevel name="[Customer].[State]" column="state"/>
    </AggName>
  </Table>
  <Dimension name="Product">
    <Hierarchy hasAll="true" primaryKey="prodid" primaryKeyTable="Product">
      <Join leftKey="mfrid" rightKey="mfrid">
        <Table name="Product"/>
        <Table name="Mfr"/>
      </Join>
      <Level name="Manufacturer" table="Mfr" column="mfrid"/>
      <Level name="Brand" table="Product" column="brand"/>
      <Level name="Name" table="Product" column="prodid"/>
    </Hierarchy>
  </Dimension>
  <!-- Rest of the cube definition -->
</Cube>
```

Alcune dimensioni sono state collassate come [Time] al livello [Quarter], [Customer] al livello [State] e [Payment Method] al livello [Payment Method], ma la dimensione [Product] ha mantenuto la sua originale forma a fiocco di neve.

L'elemento <AggForeignKey> viene usato per dichiarare quale colonne utilizzare per il collegamento con la tabella dimensionale.

Una fact table può avere zero, una o più tabelle degli aggregati associate. Ogni tabella degli aggregati è associata con una sola fact table. Queste aggregano i dati delle *misure* della fact

table su una o più dimensioni. Come esempio, consideriamo una particolare colonna della fact table che rappresenta il numero di vendite di alcuni prodotti in uno specifico giorno, in uno specifico Store, allora la tabella degli aggregati può creare la somma di questa informazione raggruppando a livello di mese, invece che a livello di giorno. Questo aggregato occuperà ragionevolmente 1/30 della dimensione della fact table (assumendo vendite comparabili in ogni giorno del mese). Adesso, richiedendo un'interrogazione MDX che richiede informazioni a livello di mese (o trimestre, o anno), eseguire la query sulla tabella degli aggregati sarà più veloce e produrrà gli stessi risultati.

Successivamente, possiamo creare non solo aggregati a livello di mese, ma anche per il livello di Store, stato. Se consideriamo 20 Store per ogni stato, la tabella degli aggregati occuperà 1/600 della tabella fact table originale. Le interrogazioni interessate solo ai livelli mese e al livello stato o loro livelli superiori useranno questa tabella.

A questo punto viene immediato chiedersi: quando viene eseguita una query, quale tabella degli aggregati viene utilizzata? Questo dipende da quali *misure* sono necessarie e a quale livello di aggregazione. La fact table originale ha sempre le *misure* corrette e il livello corretto. Ma questo può essere vero per una o più tabelle degli aggregati. In questo caso, la tabella degli aggregati con il minor costo di lettura e il minor numero di righe viene utilizzata per rispondere all'interrogazione.

Mondrian supporta due tecniche di aggregazione che sono chiamate dimensioni perse e dimensioni collassate. Per la creazione di ogni tabella degli aggregati possono essere applicate indipendentemente a ogni dimensione.

Una dimensione persa è una dimensione che viene completamene saltata nella tabella degli aggregati. Le *misure* che saranno memorizzate nella tabella saranno aggregate per tutti i valori della dimensione persa. Un possibile esempio, è una fact table con dimensioni Tempo, Locazione e Prodotto e *misura* Vendite, e una tabella degli aggregati che non ha la dimensione Locazione. Il valore delle vendite sarà quindi aggregato su tutte le Locazioni. Una tabella degli aggregati in cui tutte le dimensioni sono perse è possibile, ma conterrà una singola riga con la *misura* aggregata per tutto.

```
Fact_table(time_id, product_id, location_id, measure)
lost(time_id)
dim_table(product_id,location_id,measure(agg_over_time),fact_count)
fully_lost_dim_table(measure(agg_over_everything),fact_count)
```

La seconda tecnica d'aggregazione fornita è quella delle dimensioni collassate. Ricordando che le chiavi dimensionali nelle fact table riferiscono al livello più basso nella *gerarchia* della dimensione, per una dimensione collassata, la chiave dimensionale nella tabella degli aggregati è sostituita con un insieme di livelli dimensionali. Per esempio, se la dimensione tempo viene riferita dalla fact table a livello di giorno e in una tabella degli aggregati sono incluse solo le colonne mese, trimestre, e anno.

```
Time_dimension_table(time_id,day,month,quarter,year)
Fact_table(time_id,measure)
Collapsed_dim_table(month,quarter,year,measure(agg_month_level),fact_count)
```

In letteratura esistono altri metodi per creare tabelle degli aggregati, ma non sono supportati da Mondrian.

Una tabella degli aggregati deve essere costruita, generalmente, non in tempo reale, ma, ad esempio, tutte le notti per utilizzare i report di analisi nel giorno successivo. Considerando le dimensioni perse e le dimensioni collassate, ipotizzando N livelli per ogni dimensione, sono possibili N + 1 aggregazioni per ogni dimensione (N collassati e una persa). In più le dimensioni possono essere aggregate in modo indipendente, quindi è facile raggiungere un alto numero di possibili tabelle degli aggregati.

Chiaramente, non si vuole creare tutte le possibili tabelle degli aggregati. Quali creare dipende da due considerazioni:

- Considerare le query che vengono eseguite (dipendente dall'applicazione). Se molte query richiedono l'aggregazione per mese e per stato, questa aggregazione viene creata.
- Considerare la posizione gerarchica del livello per cui si aggrega (indipendente dall'applicazione). Considerando una dimensione di aggregazione, dal livello più basso di aggregazione verso quello successivo, generalmente, ha un più alto coefficiente di diminuzione delle righe che tra il livello N e quello N + 1 con N > 1. Questo perché il primo livello di aggregazione può essere usato per ogni query, anche se richiedono un livello successivo e perché il fan-out tende a incrementarsi ai livelli più bassi.

Per esempio, consideriamo la costruzione di una tabella di aggregati per le vendite del 1997. La fact table sarà composta come segue:

```
sales_fact_1997 (product_id, time_id, customer_id, promotion_id, store_id,
store_sales, store_cost, unit_sales)
```

nel primo esempio consideriamo una aggregazione con dimensione persa Tempo.

```
CREATE TABLE agg_l_05_sales_fact_1997 (
                            product_id INTEGER NOT NULL,
                            customer_id INTEGER NOT NULL,
                            promotion_id INTEGER NOT NULL,
                            store_id INTEGER NOT NULL,
                            store_sales DECIMAL(10,4) NOT NULL,
                            store_cost DECIMAL(10,4) NOT NULL,
                            unit_sales DECIMAL(10,4) NOT NULL,
                            fact_count INTEGER NOT NULL
);
INSERT INTO agg_l_05_sales_fact_1997 (product_id, customer_id,
  promotion_id, store_id, store_sales, store_cost,  unit_sales, fact_count)
SELECT product_id, customer_id, promotion_id, store_id,
   SUM(store_sales) AS store_sales, SUM(store_cost) AS store_cost,
   SUM(unit_sales) AS unit_sales, COUNT(*) AS fact_count
FROM sales_fact_1997
GROUP BY product_id, customer_id, promotion_id, store_id;
```

L'esempio seguente è di una tabella degli aggregati con la dimensione tempo collassata a livello mese.

```
CREATE TABLE agg_c_14_sales_fact_1997 (
                            product_id INTEGER NOT NULL,
                            customer_id INTEGER NOT NULL,
                            promotion_id INTEGER NOT NULL,
                            store_id INTEGER NOT NULL,
                            month_of_year SMALLINT(6) NOT NULL,
```

```
                            quarter VARCHAR(30) NOT NULL,
                            the_year SMALLINT(6) NOT NULL,
                            store_sales DECIMAL(10,4) NOT NULL,
                            store_cost DECIMAL(10,4) NOT NULL,
                            unit_sales DECIMAL(10,4) NOT NULL,
                            fact_count INTEGER NOT NULL
);
INSERT INTO agg_c_14_sales_fact_1997 (product_id, customer_id,
   promotion_id, store_id, month_of_year, quarter, the_year, store_sales,
   store_cost, unit_sales, fact_count)
SELECT BASE.product_id, BASE.customer_id, BASE.promotion_id,
   BASE.store_id, DIM.month_of_year, DIM.quarter, DIM.the_year,
   SUM(BASE.store_sales) AS store_sales,SUM(BASE.store_cost) AS store_cost,
   SUM(BASE.unit_sales) AS unit_sales, COUNT(*) AS fact_count
FROM  sales_fact_1997 AS BASE, time_by_day AS DIM
WHERE BASE.time_id = DIM.time_id
GROUP BY BASE.product_id, BASE.customer_id, BASE.promotion_id,
   BASE.store_id, DIM.month_of_year, DIM.quarter, DIM.the_year;
```

In questo caso possiamo notare che la chiave esterna time_id della fact table originale è stata sostituita dalle colonne: month_of_year, quarter e the_year nella tabella degli aggregati.

Di seguito riportiamo un altro esempio di tabella degli aggregati in cui abbiamo due dimensioni perse (Store e Promotion) e una collassata (Tempo a livello di trimestre).

```
CREATE TABLE agg_lc_100_sales_fact_1997 (
                            product_id INTEGER NOT NULL,
                            customer_id INTEGER NOT NULL,
                            quarter VARCHAR(30) NOT NULL,
                            the_year SMALLINT(6) NOT NULL,
                            store_sales DECIMAL(10,4) NOT NULL,
                            store_cost DECIMAL(10,4) NOT NULL,
                            unit_sales DECIMAL(10,4) NOT NULL,
                            fact_count INTEGER NOT NULL
);
INSERT INTO agg_lc_100_sales_fact_1997 ( product_id, customer_id,
   quarter,  the_year, store_sales, store_cost, unit_sales, fact_count)
SELECT BASE.product_id, BASE.customer_id, DIM.quarter, DIM.the_year,
   SUM(BASE.store_sales) AS store_sales,SUM(BASE.store_cost) AS store_cost,
   SUM(BASE.unit_sales) AS unit_sales, COUNT(*) AS fact_count
FROM sales_fact_1997 AS BASE, time_by_day AS DIM
WHERE BASE.time_id = DIM.time_id
GROUP BY BASE.product_id, BASE.customer_id, DIM.quarter, DIM.the_year;
```

Mondrian deve conoscere le tabelle degli aggregati per poterle utilizzare. È possibile definire una tabella degli aggregati esplicitamente oppure configurare opportune regole per riconoscere più tabelle degli aggregati in una volta sola.

Come Mondrian riconosce i nomi delle tabelle degli aggregati e delle colonne è la prima cosa da sapere per crearle.

Per farlo introduciamo il concetto di regole. Delle regole sono dei modelli, progettate per lavorare per tutte le tabelle dei fatti e dei nomi delle colonne. Un esempio di regole per un modello è "abc_${name}_xyz" che è parametrizzata con la variabile name. Per esempio, se il nome è "john", il modello diventa "abc_john_xyz".

Il motore delle espressioni regolari può essere usato per definire una grammatica delle espressioni accettate.

Per riconoscere le tabelle delle aggregazioni, Mondrian deve essere in grado di mappare le chiavi esterne e le *misure* della fact table nelle corrispondenti colonne della tabella delle aggregazioni. In più Mondrian deve essere in grado di identificare la colonna di conteggio degli aggregati e le possibili colonne dei livelli che possono comparire in caso di dimensioni collassate. La seguente descrizione è la rappresentazione per identificare un nome di una tabella delle aggregazioni riconosciuta automaticamente da Mondrian.

```
agg_.+_${fact_table_name}
```

che viene parametrizzata con il nome della fact table. Questa regola significa che il nome di una tabella degli aggregati viene composto concatenando il prefisso "agg_" con uno o più caratteri identificativi della tabella con "_" e con il nome della fact table a cui si riferiscono.

Un esempio sono le tabelle degli aggregati collegati alla fact table "sales_fact_1997":

- agg_l_05_sales_fact_1997;
- agg_c_14_sales_fact_1997;
- agg_lc_100_sales_fact_1997;
- agg_c_special_sales_fact_1997;
- AGG_45_SALES_FACT_1997.

Dopo aver riconosciuto che un determinato nome rispetta la regola precedente, si aspetta di trovare colonne con uno specifico nome.

- La prima colonna testata è quella di nome "fact_count".
- Come seconda cosa, cerca colonne che fanno match con le chiavi esterne della fact table. Le colonne che fanno match vengono annotate. Se una chiave esterna non fa match, significa che quella dimensione può essere stata collassata.
- A questo punto cerca di trovare corrispondenze con i nomi dei livelli. Le regole utilizzate sono:

```
${hierarchy_name}_${level_name}
${hierarchy_name}_${level_column_name}
${usage_prefix}${level_column_name}
${level_column_name}
```

- A questo punto cerca di trovare corrispondenze con i nomi delle *misure*. Le regole utilizzate sono:

```
${measure_name}
${measure_column_name}
${measure_column_name}_${aggregate_name}
```

Un altro modo per definire aggregazioni è in modo esplicito. Di seguito viene proposto un esempio di dichiarazione di una aggregazione esplicita.

```
<Cube name="Sales">
<Table name="sales_fact_1997">
  <AggExclude name="agg_c_14_sales_fact_1997" />
  <AggExclude name="agg_lc_10_sales_fact_1997" />
  <AggExclude name="agg_pc_10_sales_fact_1997" />
```

```
  <AggName name="agg_c_special_sales_fact_1997">
    <AggFactCount column="FACT_COUNT"/>
    <AggIgnoreColumn column="admin_one"/>
    <AggIgnoreColumn column="admin_two"/>
    <AggForeignKey factColumn="product_id" aggColumn="PRODUCT_ID" />
    <AggForeignKey factColumn="customer_id" aggColumn="CUSTOMER_ID" />
    <AggForeignKey factColumn="promotion_id" aggColumn="PROMOTION_ID" />
    <AggForeignKey factColumn="store_id" aggColumn="STORE_ID" />
    <AggMeasure name="[Measures].[Unit Sales]" column="UNIT_SALES_SUM" />
    <AggMeasure name="[Measures].[Store Cost]" column="STORE_COST_SUM" />
    <AggMeasure name="[Measures].[Store Sales]" column="STORE_SALES_SUM" />
    <AggLevel name="[Time].[Year]" column="TIME_YEAR" />
    <AggLevel name="[Time].[Quarter]" column="TIME_QUARTER" />
    <AggLevel name="[Time].[Month]" column="TIME_MONTH" />
  </AggName>
  <AggPattern pattern="agg_sales_fact_1997_.*">
          ....
    <AggExclude name="agg_sales_fact_1997_olddata" />
    <AggExclude pattern="agg_sales_fact_1997_test.*" />
  </AggPattern>
</Table>
....
</Cube>
```

L'elemento "AggExclude" definisce le tabelle che non devono essere considerate come tabelle degli aggregati della fact table. Nell'esempio riportato qui sopra, Mondrian ignora le tabelle con nome "agg_c_14_sales_fact_1997", "agg_lc_10_sales_fact_1997" e "agg_pc_10_sales_fact_1997". Successivamente sono riportati degli elementi di tipo "AggName" che identificano il nome delle tabelle da considerare come tabelle degli aggregati della fact table, nel caso sopra riportato "agg_c_special_sales_fact_1997" , con le regole di mapping per i nomi delle colonne.

I due elementi "AggIgnoreColumn" sono usati per specificare che le colonne di nome "admin_one" e "admin_two" devono essere ignorate da Mondrian. Se non fossero così mappate, Mondrian comunicherebbe un errore alla fine della procedura di identificazione indicando che il mapping è incompleto.

L'elemento "AggForeignKey" definisce il mapping tra le colonne della tabella degli aggregate e le chiavi dimensionali della fact table di base.

Gli elementi "AggMeasure" e "AggLevel" mappano rispettivamente le colonne che definiscono delle *misure* aggregate e le colonne che mappano dei livelli di una dimensione.

Una gerarchia padre figlio è uno speciale tipo di *gerarchia* dove ogni membro può avere un livello arbitrario. Il classico esempio di gerarchia padre-figlio è l'organigramma degli impiegati.

Quando si lavora con le gerarchie padre-figlio, l'obbiettivo è aggregare le *misure* dei membri figli nei membri padri. Per esempio, quando consideriamo l'impiegato "Bill" che è a capo di un reparto, non vogliamo visualizzare il salario di "Bill", ma il salario di "Bill" sommato con quello di tutti i suoi dipendenti che lavorano in quel reparto. È difficile generare interrogazioni SQL che effettuano queste aggregazioni, quindi Mondrian fornisce speciali strutture, chiamate "closure table", che contengono il contesto espanso delle gerarchie.

Una "closure table" ha un utilizzo simile alle tabelle degli aggregati: consiste i una copia ridondante dei dati contenuti nel database, organizzati in modo tale che Mondrian possa accedervi in modo efficiente. Una tabella degli aggregati migliora le prestazioni in aggregazione, mentre una "closure table" serve per computare in modo molto più efficiente le aggregazioni con gerarchie padre figlio.

Supponiamo che uno schema contenga un fact table con molti dati e una gerarchia padre-figlio. È possibile che tabelle degli aggregati e "closure table" lavorino insieme per migliorare le prestazioni.

Consideriamo il seguente esempio.

```
Cube: [Salary]
Dimensions: [Employee], with level [Employee],
            [Time], with levels [Year], [Quarter], [Month], [Day]
Fact table: salary (employee_id, time_id, dollars)
Parent-child dimension table: employee (employee_id, supervisor_id, name)
```

employee		
supervisor_id	*employee_id*	*name*
null	1	Frank
1	2	Bill
2	3	Eric
1	4	Jane
3	5	Mark
2	6	Carla

Tabella 2 - Esempio di tabella dimensionale per una gerarchia padre-figlio

```
Closure table: employee_closure (employee_id, supervisor_id, depth)
```

employee_closure		
supervisor_id	*employee_id*	*distance*
1	1	0
1	2	1
1	3	2
1	4	1
1	5	3
1	6	2
2	2	0
2	3	1

2	5	2
2	6	1
3	3	0
3	5	1
4	4	0
5	5	0
6	6	0

Tabella 3 - Esempio di tabella di chiusura di una gerarchia padre-figlio

```
Regular dimension table: time (year, month, quarter, time_id)
```

L'opzione più semplice è creare tabelle degli aggregati che fanno join a livello delle foglie della gerarchia padre-figlio. La seguente tabella degli aggregati è per i membri foglia della *gerarchia* degli impiegati e per il livello anno della dimensione tempo.

```
Aggregate table: agg_salary_Employee_Time_Year (employee_id,
                                                    time_year, sum_dollars)
INSERT INTO agg_salary_Employee_Time_Year
SELECT salary.employee_id, time.year AS time_year,
                                          sum(salary.dollars) AS sum_dollars
FROM salary, time
WHERE time.time_id = salary.time_id
GROUP BY salary.employee_id, time.year
```

Mondrian può usare questa tabella degli aggregate per recuperare i salari degli impiegati del livello foglia. Ma siccome la tabella degli aggregati ha la stessa chiave esterna della fact table, Mondrian può automaticamente fare il join tra le due tabelle e aggregare gli impiegati efficientemente.

Un'opzione più avanzata è combinare la "closure table" e la tabella degli aggregati in una sola, come nell'esempio che segue.

```
Aggregate table: agg_salary_Employee$Closure_Time_Year
                                    (supervisor_id, time_year, sum_dollars)
INSERT INTO agg_salary_Employee$Closure_Time_Year
SELECT ec.supervisor_id, time.year AS time_year,
                                          sum(salary.dollars) AS sum_dollars
FROM employee_closure AS ec, salary, time
WHERE ec.supervisor_id = salary.employee_id
AND time.time_id = salary.time_id
GROUP BY ec.employee_id, ec.supervisor_id, time.year
```

La tabella degli aggregati "agg_salary_Employee$Closure_Time_Year" contiene il salario di ogni impiegato, aggregato per includere membri figli diretti e indiretti, aggregati per il livello anno della dimensione tempo.

Questo funziona basandosi su un trucco interno di Mondrian. Quando Mondrian vede una "closure table" crea una dimensione ausiliaria. Nel caso della *gerarchia* degli impiegati e della sia "closure table", la dimensione verrà chiamata [Employee$Closure].

```
Dimension [Employee$Closure], levels [supervisor_id], [employee_id]
```

Quando viene valutata una cella di una query MDX che usa la *misura* salario aggregata, Mondrian trasforma le coordinate della cella della dimensione [Employee] nelle corrispondenti coordinate nella dimensione [Employee$closure]. Questa trasformazione avviene prima che Mondrian inizi a cercare tabelle degli aggregati utilizzabili, quindi se le tabelle degli aggregati contengono il nome della *gerarchia* ausiliaria, viene riconosciuta e utilizzata.

Se più di una tabella degli aggregati può essere utilizzata per una particolare query, Mondrian deve decidere quale utilizzare. Se in una tabella degli aggregati ha la stessa granularità della query, Mondrian utilizzerà quella. Se non esiste, Mondrian sceglierà la tabella degli aggregati con minore granularità e aggregherà su di essa. In generale, Mondrian sceglie la tabella degli aggregati con il minor numero di righe, che tipicamente è quella più aggregata.

C'è un'importante eccezione per le aggregazioni di tipo distinct count: esse non possono essere aggregate su una dimensione arbitraria. Per capire perché, consideriamo il caso di una catena di supermercati che ha due Store nella stessa città. Supponiamo che lo Store A abbia 1000 visite da 800 visitatori distinti nel mese di giugno mentre lo Store B ha 1500 visite da 900 visitatori distinti nello stesso mese. Chiaramente le due zone hanno un totale di 2500 visite nel mese di giugno, ma non sappiamo da quanti visitatori distinti. Possiamo dire che sono almeno 900 e possono arrivare a essere 1700, ma assumendo che alcuni visitatori hanno visitato entrambi gli Store, il totale sarà un valore intermedio. I visitatori distinti sono un esempio di *misura* di tipo distinct count che non può essere dedotta da aggregati intermedi.

Ci sono casi speciali in cui è accettabile l'aggregazione di *misure* di tipo distinct count. Supponiamo di conoscere che in giugno, gli Store A e B combinati sono stati visitati da 600 donne distinte e 700 uomini distinti. In questo caso possiamo dedurre che gli Store sono stati visitati da 1300 persone distinte visto che l'insieme delle donne e quello degli uomini sono disgiunti. In termini tecnici, il sesso è funzionalmente dipendente dall'identificatore della persona.

La regola per aggregare *misure* di tipo distinct count può essere la seguente: una *misura* di tipo distinct count sulla chiave k può essere calcolata aggregando totali intermedi più granulali solo se l'attributo che viene aggregato è dipendente funzionalmente da k.

Tranne che in questo caso speciale, è difficile creare abbastanza tabelle degli aggregati per soddisfare tutte le possibili query. Quando valuta una *misura* di tipo distinct count, Mondrian può usare solo le tabelle degli aggregati che hanno la stessa granularità della richiesta, o andare sulla fact table originale.

Questo ha importanti implicazioni nella progettazione delle tabelle degli aggregati. Se una applicazione fa uso intensivo di *misure* di tipo distinct count, sarebbe necessario creare una tabella degli aggregati per ogni granularità che viene usata.

2.4 DEFINIRE SCHEMI IN MONDRIAN

Uno schema definisce un database multidimensionale. Contiene il modello logico, costituito da cubi, gerarchie e membri, e le relazioni tra questo modello e quello fisico dei dati.

Gli schemi di Mondrian sono scritti in file XML. Gli elementi principali di uno schema sono i cubi, le *misure* e le dimensioni.

- Un cubo è una collezione di dimensioni e di *misure* di una particolare area.
- Una *misura* è una quantità che può essere interessante misurare, per esempio le unità vendute di un prodotto, o il costo d'inventario d'oggetti.
- Una dimensione è un attributo o un insieme di attributi sui quali si possono dividere le *misure* in sottocategorie. Per esempio, si può essere interessati a dividere i prodotti che sono stati venduti per il loro colore, lo Store in cui sono stati venduti, …

Quello che segue è la definizione di un semplice schema.

```
<Schema>
 <Cube name="Sales">
    <Table name="sales_fact_1997"/>
    <Dimension name="Gender" foreignKey="customer_id">
       <Hierarchy hasAll="true" allMemberName="All Genders"
                                                  primaryKey="customer_id">
           <Table name="customer"/>
          <Level name="Gender" column="gender" uniqueMembers="true"/>
       </Hierarchy>
    </Dimension>
    <Dimension name="Time" foreignKey="time_id">
       <Hierarchy hasAll="false" primaryKey="time_id">
           <Table name="time_by_day"/>
           <Level name="Year" column="the_year" type="Numeric"
                                                    uniqueMembers="true"/>
          <Level name="Quarter" column="quarter" uniqueMembers="false"/>
          <Level name="Month" column="month_of_year" type="Numeric"
                                                     uniqueMembers=false/>
        </Hierarchy>
</Dimension>
     <Measure name="Unit Sales" column="unit_sales" aggregator="sum"
                                          formatString="#,###"/>
     <Measure name="Store Sales" column="store_sales"
                      aggregator="sum"             formatString="#,###.##"/>
     <CalculatedMember name="Profit" dimension="Measures"
     formula="[Measures].[Store       Sales]-[Measures].[Store       Cost]">
        <CalculatedMemberProperty name="FORMAT_STRING"
                                            value="$#,##0.00"/>
     </CalculatedMember>
   </Cube>
</Schema>
```

Questo schema contiene un solo cubo, chiamato "Sales". Questo cubo ha due dimensioni, "Time" e "Gender" e due *misure* "Unit Sales" e "Store Sales". Possiamo scrivere la seguente interrogazione MDX sullo schema.

```
SELECT {[Measures].[Unit Sales], [Measures].[Store Sales]} ON COLUMNS,
   {[Time].[1997].[Q1].descendants} ON ROWS
FROM [Sales]
WHERE [Gender].[F]
```

I risultati della query potrebbero essere quelli riportati nella tabella seguente.

[Time]	*[Measures].[Unit Sales]*	*[Measures].[Store Sales]*
[1997].[Q1]	0	0
[1997].[Q1].[Jan]	0	0
[1997].[Q1].[Feb]	0	0
[1997].[Q1].[Mar]	0	0

Tabella 4 - Esempio di risultato di una interrogazione MDX

2.4.1 DEFINIZIONE DI UN CUBO

Un cubo è una collezione con nome di *misure* e dimensioni. La sola cosa che hanno in comune le *misure* e le dimensioni è la fact table che contiene le colonne con le *misure* e contiene le referenze a tutte le tabelle dimensionali.

```
<Cube name="Sales">
  <Table name="sales_fact_1997"/>
  ...
</Cube>
```

La fact table viene definite usando l'elemento <Table>. Se la fact table non è contenuta nello schema di default è possibile specificare in modo esplicito la sua locazione usando l'attributo schema, per esempio:

```
<Table schema="dmart" name="sales_fact_1997"/>
```

È possibile utilizzare i costruttori <View> e <Join> per costruire interrogazioni SQL più complesse. Il cubo sales considerato in precedenza definisce due *misure*, "Unit sales" e "Store Sales".

```
<Measure name="Unit Sales" column="unit_sales"
                 aggregator="sum" datatype="Integer" formatString="#,###"/>
<Measure name="Store Sales" column="store_sales"
             aggregator="sum" datatype="Numeric" formatString="#,###.00"/>
```

Ogni *misura* ha un nome che identifica la colonna nella fact table e un aggregatore. L'aggregatore è normalmente "sum" ma può essere anche "count", "min", "max", "avg" o "distinct count". L'attributo opzionale data-type specifica come la cella deve essere rappresentata nella cache di Mondrian e come deve essere restituita per analisi via XML. L'attributo data-type può avere i valori "String", "Integer" o "Numeric" dove quest'ultimo è il default, eccetto che per le funzioni di aggregazione "count" e "distinct count" che è "Integer".

L'attributo opzionale formatString specifica come il valore dell'attributo deve essere visualizzato.

Una *misura* può avere un attributo caption che deve essere il valore di ritorno della funzione Member.getCaption() che codifica particolari caratteri che altrimenti non potrebbero essere visualizzati.

```
<Measure name="Sum X" column="sum_x" aggregator="sum" caption="&#931; X"/>
```

Definiamo:

- un membro come un punto in cui le dimensioni sono determinate da un preciso insieme di valori degli attributi;
- una *gerarchia* è un insieme di membri organizzati in una struttura per convenienza di analisi. Per esempio, la *gerarchia* degli Store consiste in nome dello Store, città, regione e nazione. La *gerarchia* permette di avere dei totali intermedi;
- un livello è una collezione di membri che hanno la stessa distanza dalla radice della *gerarchia*;
- una dimensione è una collezione di gerarchie che vengono discriminate dallo stesso attributo della fact table.

Per ragioni di uniformità, le *misure* sono trattate come un membro di una speciale dimensione chiamata "Measures".

Un esempio di dimensione è il seguente:

```
<Dimension name="Gender" foreignKey="customer_id">
  <Hierarchy hasAll="true" primaryKey="customer_id">
    <Table name="customer"/>
    <Level name="Gender" column="gender" uniqueMembers="true"/>
  </Hierarchy>
</Dimension>
```

Questa dimensione consiste in una singola *gerarchia*, composta da un unico livello chiamato "Gender". I valori della dimensione potranno essere recuperati dalla colonna "gender" della tabella "customer". Questa colonna conterrà i valori 'F' e 'M', quindi la dimensione è costituita dai membri [Gender].[F] e [Gender].[M].

Per ogni vendita, la dimensione "gender" è il sesso del cliente che ha effettuato l'acquisto. Questo viene espresso con l'operazione di join con la tabella dei fatti tra i campi "sales_fact_1997.customer_id" e "customer.customer_id".

Per effettuare un operazione di join tra una tabella dimensionale e una tabella dei fatti sono necessari due attributi. L'elemento <Dimension> nell'esempio precedente contiene una proprietà di nome "foreignKey", che specifica il nome del campo nella tabella dei fatti, mentre la proprietà "primaryKey" dell'elemento <Hierarchy> specifica la chiave primaria della tabella dimensionale.

Se la *gerarchia* è distribuita su più tabelle, è possibile specificare la proprietà "primaryKeyTable" per specificare in quale tabella è contenuta la chiave primaria.

L'attributo "uniqueMembers" viene usato per ottimizzare la generazione delle interrogazione SQL. Se il progettista sa che un dato valore del campo di un livello di una dimensione è unico rispetto a tutti i valori possibili di quella colonna corrispondenti a tutti i valori del livello precedente setterà l'attributo a vero, altrimenti a falso. In termini di dipendenze funzionali, diremo che un livello potrà avere l'attributo "uniqueMembers" settato a vero se e solo se l'attributo della tabella dimensionale che contiene i valori di quel livello implica funzionalmente il valore dell'attributo della colonna della tabella dimensionale che contiene i valori del livello padre rispetto a quello considerato. Per esempio, in una dimensione Tempo, a livello di Mese setteremo la proprietà a falso perché lo stesso mese compare in diversi anni. Al contrario in una *gerarchia* di prodotti formata dai livelli "Classe del prodotto" e "Nome del

prodotto", possiamo essere sicuri che il nome del prodotto è univoco (implica funzionalmente la classe del prodotto) e quindi setteremo la proprietà a vero. Se non si è sicuri se un livello contenga membri unici o meno, è bene settare sempre l'attributo a falso. Per il primo livello della *gerarchia*, deve essere sempre settato a vero perché questo livello non ha un livello padre.

Per default, tutte le gerarchie contengono un primo livello chiamato "(All)", che contiene un unico membro di nome "(All <hierarchyName>)". Questo membro è un antenato di tutti i membri della *gerarchia* e rappresenta il totale globale. Questo membro viene anche usato come membro di default della *gerarchia*, cioè viene usato per calcolare i valori delle celle quando la dimensione non viene inclusa in nessun asse e in nessun filtro. Gli attributi "allMemberName" e "allLeveName" sovrascrivono i nomi di default di questi due elementi.

Se l'elemento <Hierarchy> ha l'attributo hasAll settato a falso se il livello "All" è stato soppresso. In questo caso, il membro di default di questa dimensione sarà il primo membro del primo livello. Per esempio, in una *gerarchia* tempo, sarà il primo anno presente. Cambiare il membro di default può generare confusione quindi viene normalmente lasciato settato a vero.

2.4.2 DEFINIZIONE DI DIMENSIONI DI TIPO TEMPO

Le dimensioni tempo, basati sui livelli Anno/Mese/Settimana/Giorno sono codificate in maniera differente negli schemi Mondrian per permettere l'esecuzione di funzioni correlate come:

- ParallelPeriod([level[, index[, member]]]);
- PeriodsToDate([level[, member]]);
- WTD([member]);
- MTD([member]);
- QTD([member]);
- YTD([member]);
- LastPeriod(index[, member]).

Le dimensioni tempo hanno l'attributo type="TimeDimension". Il ruolo del livello in una dimensione tempo viene indicata dall'attributo levelType.

levelType value	*Meaning*
TimeYears	Level is a year
TimeQuarters	Level is a quarter
TimeMonths	Level is a month
TimeDays	Level represents days

Tabella 5 - Nomi dei livelli di una gerarchia Tempo

Di seguito riportiamo un esempio di dimensione tempo.

```
<Dimension name="Time" type="TimeDimension">
  <Hierarchy hasAll="true" allMemberName="All Periods" primaryKey="dateid">
      <Table name="datehierarchy"/>
      <Level name="Year" column="year" uniqueMembers="true"
                                      levelType="TimeYears" type="Numeric"/>
```

```
    <Level name="Quarter" column="quarter" uniqueMembers="false"
                                                  levelType="TimeQuarters" />
    <Level name="Month" column="month" uniqueMembers="false"
                      ordinalColumn="month" nameColumn="month_name"
                                    levelType="TimeMonths" type="Numeric"/>
    <Level name="Week" column="week_in_month" uniqueMembers="false"
                                                    levelType="TimeWeeks" />
     <Level name="Day" column="day_in_month" uniqueMembers="false"
                       ordinalColumn="day_in_month" nameColumn="day_name"
                                     levelType="TimeDays" type="Numeric"/>
  </Hierarchy>
</Dimension>
```

Da notare nella *gerarchia* Tempo dell'esempio precedente l'uso degli attributi "ordinalColumn" e "nameColumn" negli elementi <Level>. Questi attributi modificano la visualizzazione del livello. L'attributo "ordinalColumn" specifica la colonna della tabella dimensionale che fornisce un ordinamento ai membri di un dato livello, mentre l'attributo "nameColumn" specifica la colonna da cui prendere i valori da visualizzare.

2.4.3 DEFINIZIONE DI DIMENSIONE CON PIÙ GERARCHIE

Una dimensione può contenere più di una *gerarchia*:

```
<Dimension name="Time" foreignKey="time_id">
   <Hierarchy hasAll="false" primaryKey="time_id">
      <Table name="time_by_day"/>
      <Level name="Year" column="the_year" type="Numeric"
                                                    uniqueMembers="true"/>
      <Level name="Quarter" column="quarter" uniqueMembers="false"/>
      <Level name="Month" column="month_of_year" type="Numeric"
                                                   uniqueMembers="false"/>
   </Hierarchy>
   <Hierarchy name="Time Weekly" hasAll="false" primaryKey="time_id">
      <Table name="time_by_week"/>
      <Level name="Year" column="the_year" type="Numeric"
                                                    uniqueMembers="true"/>
      <Level name="Week" column="week" uniqueMembers="false"/>
      <Level name="Day" column="day_of_week" type="String"
                                                   uniqueMembers="false"/>
  </Hierarchy>
</Dimension>
```

Da notare che la prima *gerarchia* non ha un nome. Per default, una *gerarchia* ha lo stesso nome della sua dimensione. Le due gerarchie di esempio non hanno molto in comune, non hanno neanche la stessa tabella dimensionale, ad eccezione che riferiscono alla stessa chiave dimensionale nella fact table. La ragione principale di inserire due gerarchie all'interno della stessa dimensione è di avere più senso per l'utente finale: infatti, l'utente finale saprà che non potrà usare le due gerarchie contemporaneamente nella stessa query.

2.4.4 DEFINIZIONE DI DIMENSIONI DEGENERATE

Una dimensione degenerata è una dimensione talmente semplice da non richiedere la creazione di una tabella dimensionale. Consideriamo, per esempio, la seguente fact table:

product_id	*time_id*	*payment_method*	*customer_id*	*store_id*	*item_count*	*dollars*
55	20040106	Credit	123	22	3	$3.54
78	20040106	Cash	89	22	1	$20.00
199	20040107	ATM	3	22	2	$2.99
55	20040106	Cash	122	22	1	$1.18

Tabella 6 - Esempio di fact table con una gerarchia collassata

E supponiamo di creare una tabella dimensionale per i valori della colonna "payment_method"

payment_method
Credit
Cash
ATM

Tabella 7 - Esempio di una tabella dimensionale di una dimensione collassabile

La tabella dimensionale ha solo tre valori e non include informazioni addizionali, ma introduce il costo di un'operazione di join extra. In questo caso è possibile creare una dimensione degenerata. Per fare ciò, dichiariamo una dimensione senza una tabella e Mondrian assumerà che le colonne saranno quelle della fact table.

```
<Cube name="Checkout">
  <Table name="checkout"/>
  <Dimension name="Payment method">
      <Hierarchy hasAll="true">
         <Level name="Payment method" column="payment_method"
                                                    uniqueMembers="true" />
      </Hierarchy>
  </Dimension>
</Cube>
```

Da notare che, siccome non ci sono operazioni di join, l'attributo "foreignKey" dell'elemento Dimension non è necessario, come pure l'attributo "primaryKey" dell'elemento Hierarchy.

2.4.5 DEFINIZIONE DI TABELLE INLINE

Il costrutto <InlineTable> permette di definire degli insiemi di dati all'interno dello schema. È necessario dichiarare il nome delle colonne, il loro tipo e un insieme di righe. Come per gli elementi <Table> e <View> è necessario fornire un alias univoco all'interno di tutto lo schema. Per esempio:

```
<Dimension name="Severity">
   <Hierarchy hasAll="true" primaryKey="severity_id">
      <InlineTable alias="severity">
          <ColumnDefs>
            <ColumnDef name="id" type="Numeric"/>
            <ColumnDef name="desc" type="String"/>
         </ColumnDefs>
         <Rows>
            <Row>
               <Value column="id">1</Value>
               <Value column="desc">High</Value>
            </Row>
            <Row>
               <Value column="id">2</Value>
               <Value column="desc">Medium</Value>
            </Row>
            <Row>
               <Value column="id">3</Value>
               <Value column="desc">Low</Value>
            </Row>
         </Rows>
      </InlineTable>
      <Level name="Severity" column="id" nameColumn="desc"
                                                    uniqueMembers="true"/>
  </Hierarchy>
</Dimension>
```

Ha come risultato quello di definire una tabella chiamata "severity" come mostrato nella tabella seguente:

id	*desc*
1	High
2	Medium
3	Low

Tabella 8 - Esempio del risultato della dichiarazione di una InlineTable

e di definire una dimensione sulla tabella così generata.

2.4.6 DEFINIZIONE DI DIMENSIONI IN SCHEMI A FIOCCO DI NEVE

Fino ad ora abbiamo visto come costruire cubi basati su una fact table e su alcune tabelle dimensionali. Questo modo di strutturare gli elementi di un cubo è noto con il nome di schema a stella.

In ambiti più complessi è possibile che le dimensioni siano basate su più tabelle, con un percorso di join fino alla fact table ben definito. Gli schemi che contengono questo tipo di dimensioni sono noti come schemi a ficco di neve, e possono essere definiti all'interno di Mondrian con l'operatore <Join>. Per esempio:

```
<Cube name="Sales">
```

```
  ...
   <Dimension name="Product" foreignKey="product_id">
      <Hierarchy hasAll="true" primaryKey="product_id"
                                                primaryKeyTable="product">
         <Join leftKey="product_class_key" rightAlias="product_class"
                                                rightKey="product_class_id">
            <Table name="product"/>
            <Join leftKey="product_type_id" rightKey="product_type_id">
                 <Table name="product_class"/>
                <Table name="product_type"/>
            </Join>
         </Join>
     <!-- Level declarations ... ->
     </Hierarchy>
  </Dimension>
</Cube>
```

Questo frammento di schema definisce una dimensione "Product" composta da tre tabelle. La tabella dei fatti utilizza l'attributo "product_id" per eseguire il join con l'attributo "product_id" della tabella "product" (Specificato nell'elemento Hierarchy). Quest'ultima utilizza il campo "product_class_key" per effettuare il join con la tabella "product_class" utilizzando l'attributo "product_class_id". Infine, il join tra "product_class" e "product_type" viene fatto sugli attributi "product_type_id" delle due tabelle. Nell'esempio precedente sono necessari due elementi <Join> innestati l'uno dentro l'altro visto che l'operando <Join> considera come tabella del lato sinistro del join come quella riferita all'elemento in cui è contenuto.

2.4.7 DEFINIZIONE DI DIMENSIONI CONDIVISE TRA PIÙ CUBI

All'interno di schemi Mondrian è possibile definire dimensioni condivise tra più cubi. All'interno delle definizioni di queste dimensioni si omette di specificare il nome del campo che riveste il ruolo di chiave dimensionale nella fact table. Successivamente, all'interno di ogni cubo, l'elemento <DimensionUsage> specifica quale dimensione utilizzare, eventualmente cambia il nome della dimensione e specifica la chiave dimensionale da utilizzare nella fact table. Un esempio di uso di dimensioni condivise può essere il seguente:

```
<Dimension name="Store Type">
   <Hierarchy hasAll="true" primaryKey="store_id">
      <Table name="store"/>
      <Level name="Store Type" column="store_type" uniqueMembers="true"/>
   </Hierarchy>
</Dimension>
<Cube name="Sales">
   <Table name="sales_fact_1997"/>
   ...
   <DimensionUsage name="Store Type" source="Store Type"
                                                   foreignKey="store_id"/>
</Cube>
<Cube name="Warehouse">
```

```
   <Table name="warehouse"/>
    ...
   <DimensionUsage name="Store Type" source="Store Type"
                                          foreignKey="warehouse_store_id"/>
</Cube>
```

2.4.8 OTTIMIZZAZIONI DI UNO SCHEMA

La definizione di uno schema indica a Mondrian la modalità per raggiungere i dati, ma non come andarli a leggere. È possibile applicare una serie di ottimizzazioni alla generazione delle interrogazioni per aumentare le prestazioni. Ad esempio:

- se una dimensione ha pochi membri al suo interno, Mondrian lo inserirà nella cache al suo primo utilizzo, per non andarlo più a richiedere in seguito. Per specificare cosa significa pochi membri è necessario specificare la proprietà mondrian.rolap.LargeDimensionThreshold;
- se una dimensione (o più precisamente un livello di una dimensione) è presente sulla fact table, Mondrian non esegue join;
- se due dimensioni accedono alla stessa tabella attraverso lo stesso percorso di join, allora Mondrian effettuerà il join una sola volta. Ad esempio, "Gender" e "Age" possono essere colonne della tabella "customer".

2.4.9 DEFINIZIONE DI GERARCHIE PADRE-FIGLIO

Una *gerarchia* convenzionale ha un rigido insieme di livelli, e ogni membro di un livello ha un padre membro del livello superiore. Una *gerarchia* di tipo padre-figlio ha un solo livello (se non si considera il livello speciale "All"), ma ogni suo membro può avere antenati dello stesso livello. Un esempio classico è quello della rappresentazione della *gerarchia* degli impiegati in un'azienda:

```
<Dimension name="Employees" foreignKey="employee_id">
  <Hierarchy hasAll="true" allMemberName="All Employees"
                                                  primaryKey="employee_id">
     <Table name="employee"/>
     <Level name="Employee Id" uniqueMembers="true" type="Numeric"
                 column="employee_id" nameColumn="full_name"
                          parentColumn="supervisor_id" nullParentValue="0">
      <Property name="Marital Status" column="marital_status"/>
      <Property name="Position Title" column="position_title"/>
      <Property name="Gender" column="gender"/>
      <Property name="Salary" column="salary"/>
      <Property name="Education Level" column="education_level"/>
      <Property name="Management Role" column="management_role"/>
    </Level>
  </Hierarchy>
</Dimension>
```

Gli attributi importanti in questa descrizione sono "parentColumn" e "nullParentValue". Il primo attributo specifica il nome della colonna che collega il membro corrente con il suo

superiore. L'elemento <ParentExpression> figlio dell'elemento <Level> è equivalente all' attributo "parentColumn", ma permette di definire espressioni SQL arbitrarie. Il secondo attributo è il valore che indica che un membro non ha antenati. Il valore di default è "null", ma visto che alcuni DBMS non indicizzano i valori "null", i progettisti possono usare altri valori come, ad esempio, la stringa vuota, 0 o -1.

Un problema collegato con le gerarchie padre-figlio, è l'ammontare di lavoro che deve svolgere Mondrian per computare i totali delle celle. Supponiamo che la tabella degli impiegati contenga i seguenti dati:

employee		
supervisor_id	*employee_id*	*full_name*
null	1	Frank
1	2	Bill
2	3	Eric
1	4	Jane
3	5	Mark
2	6	Carla

Tabella 9 - Esempio di tabella dimensionale di una gerarchia padre-figlio

Se vogliamo calcolare il budget totale per i salari di Bill, dobbiamo aggiungere i salari di "Eric" e "Carla" (che hanno come superiore "Bill") e di "Mark" (che ha come superiore "Eric"). Normalmente Mondrian utilizza delle interrogazioni SQL con clausole GROUP BY per calcolare i totali, ma non esiste un costruttore SQL per le gerarchie. Quindi, per default, Mondrian genera un'interrogazione per ogni supervisore, per recuperare il totale di tutti i successori diretti (figli) di quel supervisore.

Questo approccio presenta alcuni svantaggi. Per prima cosa, le prestazioni non sono ottime se la *gerarchia* contiene più di un centinaio di membri. Poi, visto che Mondrian implementa l'aggregatore DISTINCT COUNT, non è possibile definire *misure* definite con quest'ultimo aggregatore in cubi che contengono gerarchie padre-figlio.

Per risolvere questi problemi Mondrian supporta il meccanismo della tabella di chiusura. Una tabella di chiusura, è una tabella SQL che contiene un record per ogni relazione impiegato-supervisore e la distanza gerarchica che esiste tra i due. Riprendendo l'esempio precedente:

employee_closure		
supervisor_id	*employee_id*	*distance*
1	1	0
1	2	1
1	3	2
1	4	1
1	5	3
1	6	2

2	2	0
2	3	1
2	5	2
2	6	1
3	3	0
3	5	1
4	4	0
5	5	0
6	6	0

Tabella 10 - Esempio di tabella di chiusura

In un catalogo, l'elemento <Closure> mappa il livello in una tabella.

```
<Dimension name="Employees" foreignKey="employee_id">
  <Hierarchy hasAll="true" allMemberName="All Employees"
                                          primaryKey="employee_id">
    <Table name="employee"/>
    <Level name="Employee Id" uniqueMembers="true" type="Numeric"
              column="employee_id" nameColumn="full_name"
                    parentColumn="supervisor_id" nullParentValue="0">
      <Closure parentColumn="supervisor_id" childColumn="employee_id">
        <Table name="employee_closure"/>
      </Closure>
      <Property name="Marital Status" column="marital_status"/>
      <Property name="Position Title" column="position_title"/>
      <Property name="Gender" column="gender"/>
      <Property name="Salary" column="salary"/>
      <Property name="Education Level" column="education_level"/>
      <Property name="Management Role" column="management_role"/>
   </Hierarchy>
</Dimension>
```

Questa tabella permettere di calcolare i totali delle *misure* con interrogazioni SQL standard. Per ottimizzare l'esecuzione dell'interrogazione, è raccomandabile dichiarare due indici, uno sulla chiave della tabella e uno sul campo che contiene l'identificatore del padre dell'elemento corrente.

```
CREATE UNIQUE INDEX employee_closure_pk ON employee_closure (
   supervisor_id,
   employee_id
);
CREATE INDEX employee_closure_emp ON employee_closure (
   employee_id
);
```

Questa tabella deve essere ripopolata ogni volta che la *gerarchia* viene cambiata. Mondrian non si occupa di questo compito. Di seguito riportiamo un esempio di store procedure che ricalcala la tabella di chiusura.

```
CREATE PROCEDURE close_employee()
```

```
BEGIN
  DECLARE distance int;
  TRUNCATE TABLE employee_closure;
  SET distance = 0;
  INSERT INTO employee_closure (supervisor_id, employee_id, distance)
     SELECT employee_id, employee_id, distance
     FROM employee;
  REPEAT
     SET distance = distance + 1;
     INSERT INTO employee_closure (supervisor_id, employee_id, distance)
        SELECT employee_closure.supervisor_id, employee.employee_id,
                                                                distance
        FROM employee_closure, employee
        WHERE employee_closure.employee_id = employee.supervisor_id
        AND employee_closure.distance = distance - 1;
   UNTIL (ROW_COUNT() == 0))
   END REPEAT
END
```

2.4.10 DEFINIZIONE DI PROPRIETÀ DI UN MEMBRO

Le proprietà di un membro sono definite dall'elemento <Property> all'interno dell'elemento <Level>. Un esempio di dichiarazione di una proprietà di un membro è la seguente:

```
<Level name="MyLevel" column="LevelColumn" uniqueMembers="true"/>
  <Property name="MyProp" column="PropColumn"
                                formatter="com.acme.MyPropertyFormatter"/>
<Level/>
```

Dopo che le proprietà sono state definite nello schema, è possibile usarle nelle interrogazioni MDX attraverso la funzione member.Properties("propertyName"). Ad esempio:

```
SELECT {[Store Sales]} ON COLUMNS,
  TopCount(Filter([Store].[Store Name].Members,
  [Store].CurrentMember.Properties("Store Type") = "Supermarket"),
  10,
  [Store Sales]) ON ROWS
FROM [Sales]
```

Mondrian cerca di dedurre il tipo della proprietà. Se il nome della proprietà è una stringa costante, il tipo viene determinato basandosi sulla definizione della proprietà all'interno del catalogo ("String", "Numeric" o "Boolean"). Se il nome della proprietà è una espressione (ad esempio CurrentMember.Properties("Store"+"Type")), Mondrian ritorna un valore senza tipo.

2.4.11 MISURE E INSIEMI CALCOLATI

Supponiamo di volere creare un valore che non proviene da una colonna della fact table, ma da una formula MDX. Una possibile modalità è quella di usare la clausola WITH MEMBER, come segue:

```
WITH MEMBER [Measures].[Profit] AS
    '[Measures].[Store Sales]-[Measures].[Store Cost]',
    FORMAT_STRING = '$#,###'
SELECT {[Measures].[Store Sales], [Measures].[Profit]} ON COLUMNS,
    {[Product].Children} ON ROWS
FROM [Sales]
WHERE [Time].[1997]
```

Un'altra possibilità che non prevede di dover includere la formula in tutte le applicazioni, è di definire la formula nello schema, come parte di un cubo:

```
<CalculatedMember name="Profit" dimension="Measures">
  <Formula>[Measures].[Store Sales] - [Measures].[Store Cost]</Formula>
  <CalculatedMemberProperty name="FORMAT_STRING" value="$#,##0.00"/>
</CalculatedMember>
```

Una possibilità aggiuntiva dell'ultima modalità è trasformare una *misura* o un membro calcolato in invisibile. Se si specifica visible="false", l'interfaccia utente viene avvisata che il membro o la *misura* è invisibile. Questo è utile quando si vogliono eseguire calcoli in un certo numero di passi e nascondere i risultati intermedi. Per esempio:

```
<Measure name="Store Cost" column="store_cost" aggregator="sum"
                                  formatString="#,###.00" visible="false"/>
<CalculatedMember name="Margin" dimension="Measures" visible="false">
  <Formula>([Measures].[Store Sales] - [Measures].[Store Cost]) /
                                    [Measures].[Store Cost]</Formula>
</CalculatedMember>
<CalculatedMember name="Store Sqft" dimension="Measures" visible="false">
  <Formula>[Store].Properties("Sqft")</Formula>
</CalculatedMember>
<CalculatedMember  name="MarginSqft"  dimension="Measures"  visible="true">
  <Formula>[Measures].[Margin] / [Measures].[Store Cost]</Formula>
  <CalculatedMemberProperty name="FORMAT_STRING" value="$#,##0.00"/>
</CalculatedMember>
```

Con la clausola WITH SET delle interrogazioni MDX permette di dichiarare degli insiemi che possono essere usate al loro interno. Per esempio:

```
WITH SET [Top Sellers] AS
    'TopCount([Warehouse].[Warehouse Name].MEMBERS, 5,
    [Measures].[Warehouse Sales])'
SELECT
    {[Measures].[Warehouse Sales]} ON COLUMNS,
    {[Top Sellers]} ON ROWS
FROM [Warehouse]
WHERE [Time].[Year].[1997]
```

L'elemento <NamedSet> permette di definire insiemi con nomi come parte di un cubo. Per esempio:

```
<Cube name="Warehouse">
  ...
  <NamedSet name="Top Sellers">
    <Formula>TopCount([Warehouse].[Warehouse Name].MEMBERS, 5,
```

```
    [Measures].[Warehouse Sales])</Formula>
  </NamedSet>
</Cube>
```

```
SELECT
    {[Measures].[Warehouse Sales]} ON COLUMNS,
    {[Top Sellers]} ON ROWS
FROM [Warehouse]
WHERE [Time].[Year].[1997]
```

Warehouse	*Warehouse Sales*
Treehouse Distribution	31,116.37
Jorge Garcia, Inc.	30,743.77
Artesia Warehousing, Inc	29,207.96
Jorgensen Service Storage	22,869.79
Destination, Inc.	22,187.42

Tabella 11 - Risultati di una interrogazione che utilizza un insieme definito nello schema

2.4.12 ESTENSIONI DI MONDRIAN

Alcune volte, il linguaggio per definire gli schemi in Mondrian, o il linguaggio MDX non sono abbastanza flessibili per risolvere tutti i tipi di problemi. Per risolvere questi ultimi è possibile scrivere un pezzo di codice java da aggiungere all'applicazione.

Tutte le estensioni di Mondrian sono tecnicamente delle Service Provider Interface (SPI), cioè delle classi java che devono implementare un'interfaccia e che Mondrian può chiamare a tempo di esecuzione. È necessario registrare l'estensione (generalmente all'interno del file catalogo.xml o nella stringa di connessione) e che la classe sia disponibile all'interno del classpath. Esistono diversi tipi di estensioni di Mondrian.

- Le funzioni definite dall'utente devono implementare l'interfaccia mondrian.spi.UserDefinedFunction. Per esempio:

```
import mondrian.olap.*;
import mondrian.olap.type.*;
import mondrian.spi.UserDefinedFunction;

public class PlusOneUdf implements UserDefinedFunction {
    // public constructor
    public PlusOneUdf() {
    }

    public String getName() { return "PlusOne"; }

    public String getDescription() {
        return "Returns its argument plus one";
    }

    public Syntax getSyntax() {
```

```
        return Syntax.Function;
    }

    public Type getReturnType(Type[] parameterTypes) {
        return new NumericType();
    }

    public Type[] getParameterTypes() {
        return new Type[] {new NumericType()};
    }

    public Object execute(Evaluator evaluator, Exp[] arguments) {
       final   Object   argValue   =arguments[0].evaluateScalar(evaluator);
       if (argValue instanceof Number) {
            return new Double(((Number) argValue).doubleValue() + 1);
       } else {
            return null;
       }
    }

    public String[] getReservedWords() {
        return null;
    }
}

<Schema>
  ...
  <UserDefinedFunction name="PlusOne" class="it.quix.PlusOneUdf">
</Schema>
```

- Un formattatore di cella che modifica il comportamento della funzione Cell.getFormattedValue(). La classe deve implementare l'interfaccia mondrian.olap.CellFormatter, e deve essere specificata come segue:

```
<Measure name="name" formatter="it.quix.MyCellFormatter"/>
```

- Un formattatore di membro che modifica il comportamento della funzione Member.getCaption(). La classe deve implementare l'interfaccia mondrian.olap.MemberFormatter e deve essere specificata come segue:

```
<Level column="column" name="name" formatter="it.quix.MyMemberFormatter"/>
```

- Un formattatore di proprietà che modifica il comportamento della funzione Property.getPropertyFormattedValue(). La classe deve implementare l'interfaccia mondrian.olap.PropertyFormatter e deve essere specificata come segue:

```
<Level name="MyLevel" column="LevelColumn" uniqueMembers="true"/>
  <Property name="MyProp" column="PropColumn"
                                formatter="it.quix.MyPropertyFormatter"/>
<Level/>
```

- Un processore di schemi che implementa l'interfaccia mondrian.rolap.DynamicSchemaProcessor. Il suo uso deve essere specificato come parte della stringa di connessione, come ad esempio:

```
Jdbc=jdbc:odbc:MondrianFoodMart; JdbcUser=ziggy;JdbcPassword=stardust;
                                DynamicSchemaProcessor=com.acme.MySchemaProcessor
```

L'effetto è quello di modificare la lettura dello schema. Questo può essere importante per implementare l'internazionalizzazione in certe applicazioni.

2.4.13 CONTROLLO DEGLI ACCESSI A UN CUBO

Il controllo degli accessi in Mondrian viene effettuato con il meccanismo dei ruoli. I ruoli sono definiti dagli elementi <Role> che sono riportati come figli dell'elemento <Schema> dopo l'ultimo elemento <Cube> (quindi i ruoli saranno validi per tutti i cubi di quel catalogo). Il seguente è un esempio di ruolo:

```
<Role name="California manager">
  <SchemaGrant access="none">
    <CubeGrant cube="Sales" access="all">
      <HierarchyGrant hierarchy="[Store]" access="custom"
                                        topLevel="[Store].[Store Country]">
        <MemberGrant member="[Store].[USA].[CA]" access="all"/>
        <MemberGrant member="[Store].[USA].[CA].[Los Angeles]"
                                                          access="none"/>
      </HierarchyGrant>
      <HierarchyGrant hierarchy="[Customers]" access="custom"
                                topLevel="[Customers].[State Province]"
                                   bottomLevel="[Customers].[City]">
        <MemberGrant member="[Customers].[USA].[CA]" access="all"/>
        <MemberGrant member="[Customers].[USA].[CA].[Los Angeles]"
                                                          access="none"/>
      </HierarchyGrant>
      <HierarchyGrant hierarchy="[Gender]" access="none"/>
    </CubeGrant>
  </SchemaGrant>
</Role>
```

Uno <SchemaGrant> definisce l'accesso di default agli oggetti dello schema. L'attributo access può avere il valore "all" o "none". Il valore può essere sovrascritto dal valore dei singoli oggetti. Nel caso riportato nell'esempio precedente, visto che access è settato a none, un utente con quel ruolo sarà abilitato solo a navigare sul cubo "Sales" perché esplicitamente riportato tra i diritti.

Un <CubeGrant> definisce gli accessi a un particolare cubo, e in modo simile al <SchemaGrant> l'attributo "access" può assumere i valori "all" o "none" e può essere soprascritto dai suoi sotto-oggetti.

Un <HierarchyGrant> definisce l'accesso a una *gerarchia*. L'attributo access può assumere i valori "all" per indicare che tutti i membri sono visibili, "none" per indicare che l'esistenza della *gerarchia* è nascosta all'utente oppure "custom". Con l'accesso di tipo "custom" è possibile

usare gli attributi “topLevel” e “bottomLevel” per limitare i livelli visibili dall’utente. Un’altra alternativa è quella di usare gli elementi di tipo <MemberGrant> per definire in modo personalizzato i membri visibili.

È possibile definire degli elementi <MemberGrant> solo racchiusi in elementi <HierarchyGrant> che hanno l’attributo access settato a "custom". Questi elementi danno o rimuovono diritti di accesso a un dato membro e a tutti i suoi figli. Il meccanismo dei <MemberGrant> segue le seguenti regole:

1. I membri ereditano l’accesso dai loro predecessori. Ad esempio, se si proibisce l’accesso a “California”, non si avrà accesso neanche a “San Francisco”.
2. I diritti sono ordine dipendenti. Ad esempio, se si garantisce l’accesso a “USA” e poi si revoca l’accesso a “Oregon” allora l’utente non sarà in grado di vedere “Oregon”. Se le regole vengono invertite, allora l’utente sarà in grado di vedere tutto.
3. Un membro è visibile se qualcuno dei suoi figli è visibile. Supponiamo di impedire l’accesso a “USA” e poi di darlo a “California”. Un utente potrà vedere “USA” e “California” ma non vedrà gli altri stati. Il comportamento meno intuitivo è che i totali su “USA” saranno comunque quelli che riflettono la somma di tutti gli stati.
4. I <MemberGrant> non possono comunque sovrascrivere i permessi attributi con gli attributi “topLevel” e “bottomLevel” dell’elemento <Hierarchy>. Se si setta topLevel=”[Store].[Store State]” e si da accesso a “California” non si sarà abilitati a vedere “USA”.

Nell’esempio, l’utente ha accesso a “California” e a tutte le città in essa contenute ad eccezione di “Los Angeles”. Sarà autorizzato a vedere “USA” perché “California” che è un suo figlio è visibile, ma non le altre nazioni. Non potrà vedere “All store” perché l’attributo “topLevel” è settato a “Store Country”.

Un ruolo ha effetto sono se è associato a una connessione. Per default, hanno un ruolo che garantisce accesso a tutti i cubi contenuti nello schema della connessione. Molti database associano ruoli agli utenti e automaticamente assegnano i loro diritti all’accesso. Comunque, Mondrian non ha la nozione di utenti, quindi è possibile stabilire un ruolo in due modalità diverse:

1. nella stringa di connessione;
2. da codice, nei parametri passati alla connessione.

2.5 LA PIATTAFORMA PENTAHO

Il progetto Pentaho BI è un’iniziativa della comunità open source di fornire alle organizzazioni una classe di applicazioni per le loro esigenze di *business intelligence*.

2.5.1 LA PIATTAFORMA PENTAHO BI

La piattaforma Pentaho BI è diversa dai prodotti tradizionali di *business intelligence*. È un framework con componenti per la *business intelligence* che permette alle compagnie di sviluppare soluzioni complete per i problemi di *business intelligence*.

La piattaforma è process-centric perché il controllore centrale è il Workflow engine. Il Workflow engine usa le definizioni dei processi per definire il processo di *business intelligence* che esegue nella piattaforma. Il processo può essere facilmente personalizzato ed è possibile aggiungere nuovi processi. La piattaforma include componenti e report per analizzare le prestazioni di questi processi.

La piattaforma è solution-oriented perché le operazioni della piattaforma sono specificate nel processo di definizione e nei documenti delle operazioni che specificano ogni attività.

Questi processi e queste operazioni insieme definiscono la soluzione al problema di *business intelligence*. La soluzione può essere facilmente integrata nelle procedure di business esterne alla piattaforma. La definizione della soluzione può contenere un certo numero di processi e operazioni.

La piattaforma è composta:

- dal Framework di BI;
- dai componenti di BI;
- dal Workbench di BI;
- dai desktop Inboxes.

Il framework fornisce procedure per logging, auditing, security, scheduling, ETL, web services, attribute repository e rules engine.

I componenti di BI includono componenti per il reporting, l'analisi, il workflow, il dashboards e per il data mining.

Il workbench è un insieme di strumenti di progettazione e di amministrazione che sono integrati nell'ambiente Eclipse. Questi strumenti permettono agli analisti e ai progettisti di creare report, dashboard, modelli di analisi e processi di *business intelligence*.

I desktop Inboxes possono essere lettori RSS di terze parti o Pentaho Inbox Alerter. Le Inboxes consegnano le notifiche generate da processi e report.

Il framework e i componenti per la *business intelligence* sono progettati per usare il workbench e sono forniti dal server Pentaho. Il server Pentaho è un motore guidato dal workflow engine, che coordina l'esecuzione e la comunicazione tra i componenti.

Il controllo centralizzato del workflow engine è essenziale nell'architettura della piattaforma.

- La piattaforma è costruita su processi e processa definizioni. La piattaforma è in grado di comprendere la natura dei processi perché tutto quello che esegue in essa è un processo.
- I processi sono definiti con un linguaggio standard di definizione dei processi e sono visibili, modificabili e personalizzabili dall'esterno.

- Logging, auditing e security sono assicurati dal cuore del framework e sono utilizzati automaticamente per essere sicuri di avere dati accurati di audit e per monitorare le prestazioni.

L'architettura è una combinazione di codice sorgente originale e di componenti open source che sono stati integrati per garantire la completezza e la scalabilità della piattaforma.

La piattaforma Pentaho BI è costruita su delle fondamenta formate da server, motori e componenti open source. Questi forniscono il server J2EE, la sicurezza, il rules engine, il content management e il data integration del sistema. Molti di questi componenti sono basati su standard e possono essere sostituiti con altri prodotti. Per creare una soluzione integrata Pentaho ha aggiunto:

- *metadati* comuni per la definizione di documenti;
- interfacce utente comuni;
- sicurezza;
- notificazione tramite email;
- installazione, integrazione e validazione di tutti i componenti;
- semplici soluzioni;
- connettori tra le applicazioni;
- strumenti per la diagnostica.

La piattaforma Pentaho BI è costruita usando un certo numero di componenti open source di terze parti. I componenti open source permettono a Pentaho di focalizzarsi sul valore aggiunto di nuove funzioni nel sistema.

2.5.2 ARCHITETTURA – IL SERVER PENTAHO

Il server Pentaho è costruito sopra al framework e ai componenti di BI. Il server esegue dentro a un web server J2EE compliant come Apache, Jboss AS, WebSphere, WebLogic e Oracle AS. Il framework e i componenti possono essere incorporati al suo interno o in altri server o applicazioni. Il diagramma che segue mostra le relazioni tra i maggiori componenti all'interno del server.

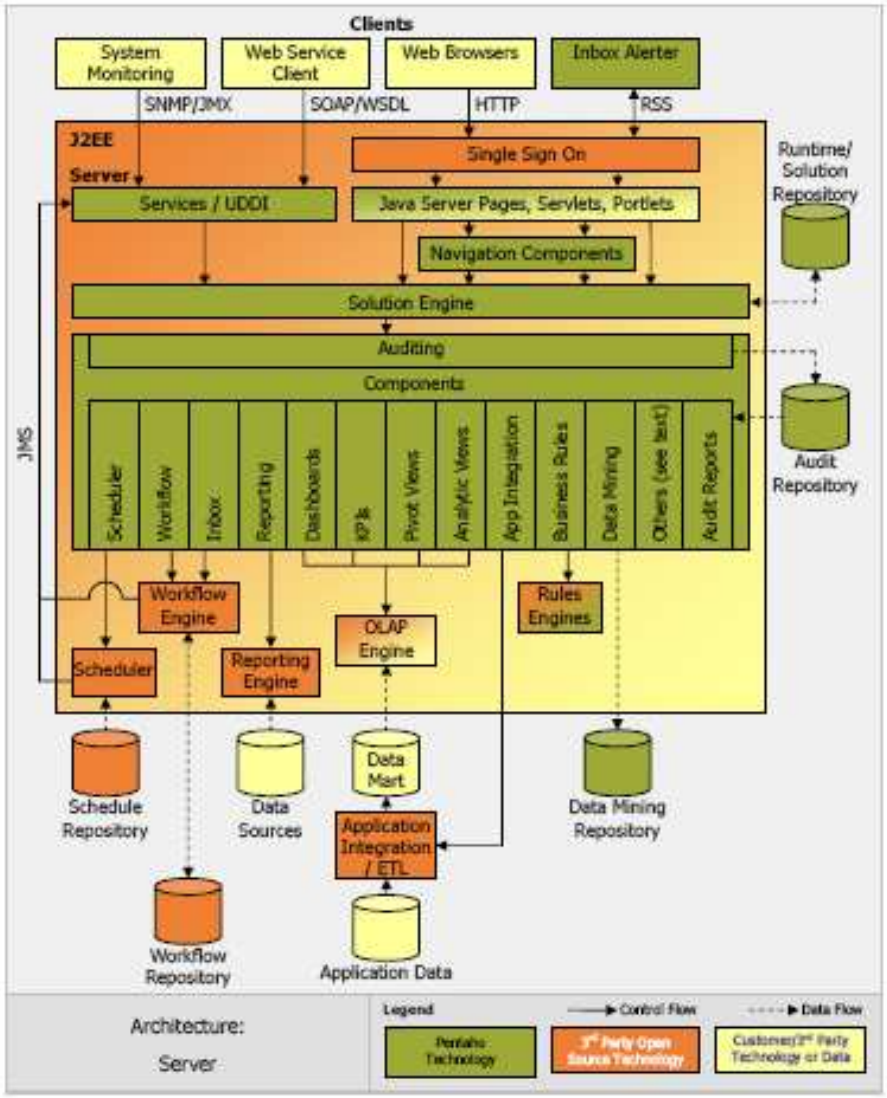


Figura 9 - Architettura della piattaforma Pentaho

Il server Pentaho fornisce numerose funzioni alla piattaforma per presentare agli utenti le informazioni come preferiscono. I contenuti dei componenti possono essere richiamati in XML, HTML o essere inclusi in portlet JSR 168.

Il server Pentaho contiene i motori e i componenti per creare report, fare analisi, definire business rules, mandare notifiche e fare workflow. Questi componenti sono integrati insieme quindi possono essere utilizzati per risolvere problemi di *business intelligence*.

I documenti che definiscono le soluzioni ai problemi di BI sono documenti XML che contengono:

- la definizione del processo di business;
- la definizione di attività che eseguono come parte di un processo. Queste attività includono definizioni per la sorgente dati, le query, i template per i report, le regole di notificazione, le regole di business,…;
- la relazione tra tutti gli elementi descritti sopra.

I componenti nel server contano su un Solution Engine per le informazioni che riguardano i documenti delle soluzioni, per la sicurezza, per le informazioni sui report e sugli elementi del workflow e per l'auditing. Più di una soluzione può essere eseguita nel server. I

documenti che descrivono le soluzioni possono essere copiati da un server ad un altro e possono essere distribuiti liberamente.

Il server Pentaho contiene una struttura che fornisce un sistema di amministrazione avanzato. Questo include servizi di monitoraggio, supporto per i Web service, strumenti di validazione delle configurazioni e strumenti di diagnostica.

Il server Pentaho contiene sistemi e componenti che forniscono strumenti di reporting e analisi delle prestazioni. Questi includono caratteristiche di slice-and-dice, what-if e data mining sugli attributi degli elementi del workflow, su processi singoli e servizi implicati nel processo di workflow.

Il server Pentaho include uno strumento per l'integrazione la trasformazione e il caricamento dei dati (ETL).

Il framework integra e amplia componenti open source di terze parti come:

- Mondrian OLAP server e JPivot Analysis front-end;
- Firebird RDBMS;
- Enhydra ETL, Shark e JaWE Workflow;
- JBoss application server, Hibernate e Portal;
- Weka Data Minig;
- Eclipse Workbench e BIRT reporting components;
- JOSSO single sign-on e LDAP integration;
- Mozilla Rhino Javascript Processor.

La piattaforma utilizza standard aperti e protocolli come: XML, JSR-94 (JCP's Rules Engine API), JSR-168 (JCP's Portlet Spec), SVG (Scalable Vector Graphics), XPDL (WFMC XML Process Definition Language), Xforms (W3C Web Forms), MDX (Microsoft OLAP Query Language), WSBPEL(Web Services Business Process Execution Language), WSDL (Web Services Description Language) e SOAP(Simple Object Access Protocol).

2.5.3 ALTRI COMPONENTI

La Inbox Alerter è una applicazione che necessita di essere installata sulla macchina dell'utente che vuole trarre vantaggio da questa funzionalità. La Inbox Alerter fornisce caratteristiche come:

- notifica di nuovi workflow;
- notifica del rilascio di nuovi report;
- gestione dei contenuti off-line.

Il Inbox Alerter usa comunicazioni RSS fornite dal server Pentaho. Tutti gli ascoltatori che supportano l'autenticazione possono ricevere le notificazioni dal server. Per la gestione dei contenuti off-line è richiesto Inbox Alerter.

Il server Pentaho include dei depositi integrati che memorizzano i dati necessari per definire e raggiungere delle soluzioni:

- Solution Repository contiene i *metadati* che definiscono le soluzioni;
- Runtime Repository contiene gli elementi di lavoro che il motore di workflow gestisce;
- Audit Repository: contiene le informazioni di audit;

2.5.4 ARCHITETTURE DEL WORKBENCH

Il Workbench è un ambiente di amministrazione e di progettazione Eclipse-based. Il Workbench genera le definizioni delle soluzioni che sono usate dal server Pentaho. Il Workbench esegue sotto diverse piattaforme. Il Workbench permette alle soluzioni, ai report, alle query, alle business rules, ai dashboard e ai workflow di essere visualizzati e editate graficamente. The Workbench è un'applicazione java che viene installata nei sistemi di amministrazione e di progettazione.

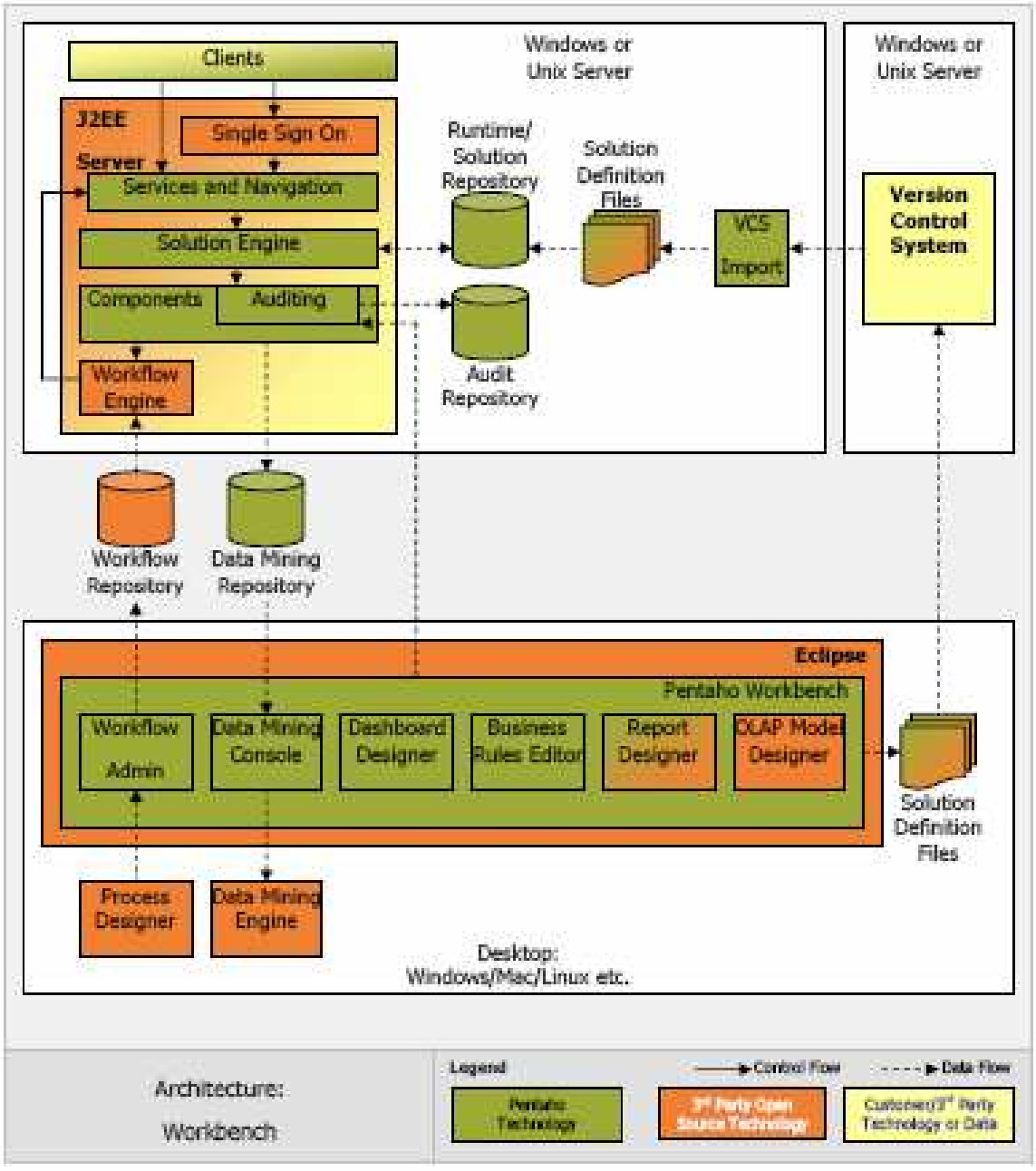


Figura 10 - Architettura del Workbench di Pentaho

2.5.5 EMBEDDED ARCHITECTURE

La tecnologia Pentaho può essere inclusa in un'applicazione Java standalone o in un'applicazione Java server-based. Il solution engine e i package dei componenti devono essere installati. Questi sono gli unici componenti e motori che sono richiesti e che necessitano di essere configurati. Più specificatamente i componenti opzionali sono: il Workflow engine, il Workflow repository, il runtime repository, l'auditing e audit repository, l'application integrator, l'ETL, i componenti per l'interfaccia utente, il solution repository e i file di definizione delle soluzioni.

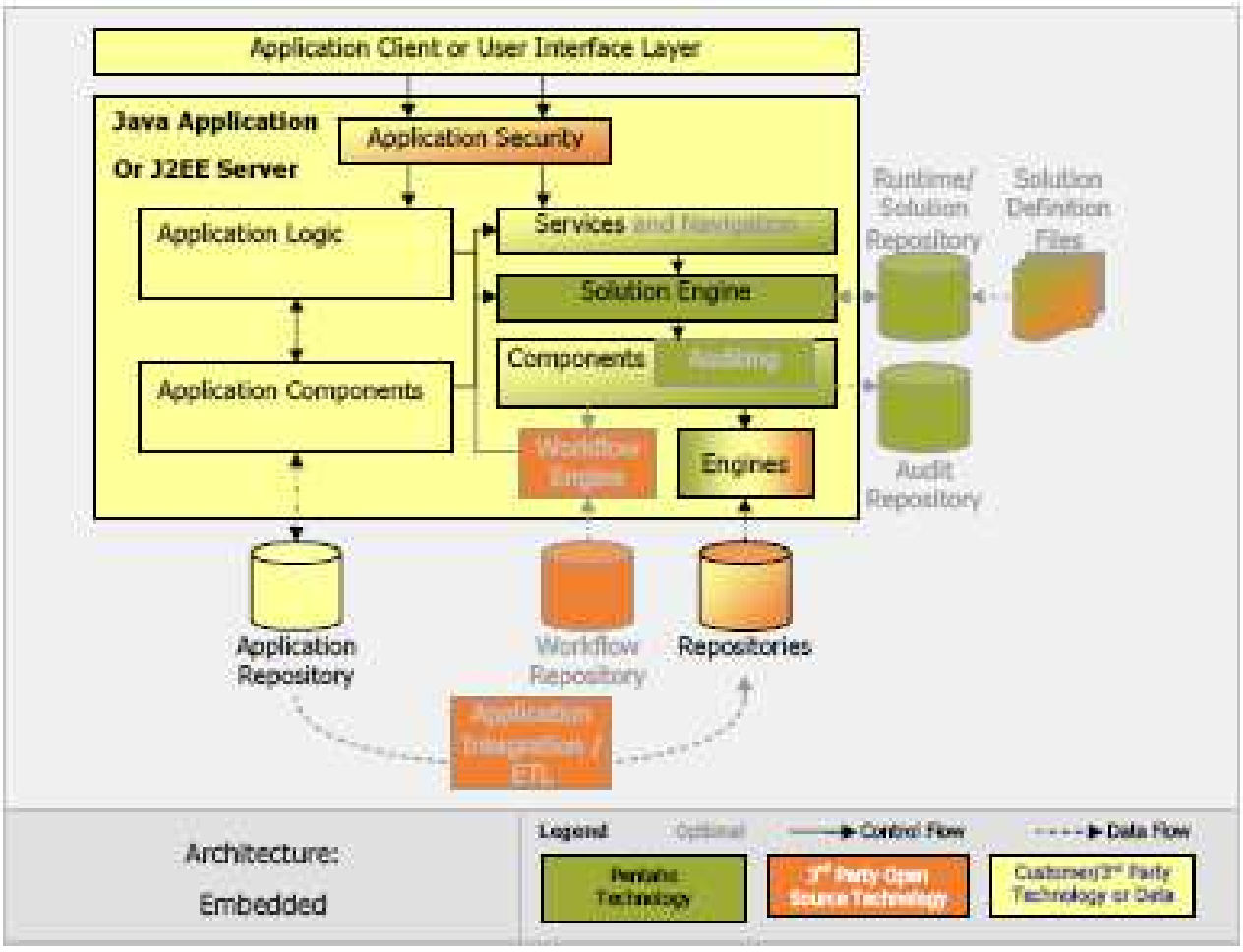


Figura 11 - Architettura per includere Pentaho in altre applicazioni

2.6 JPIVOT E WCF

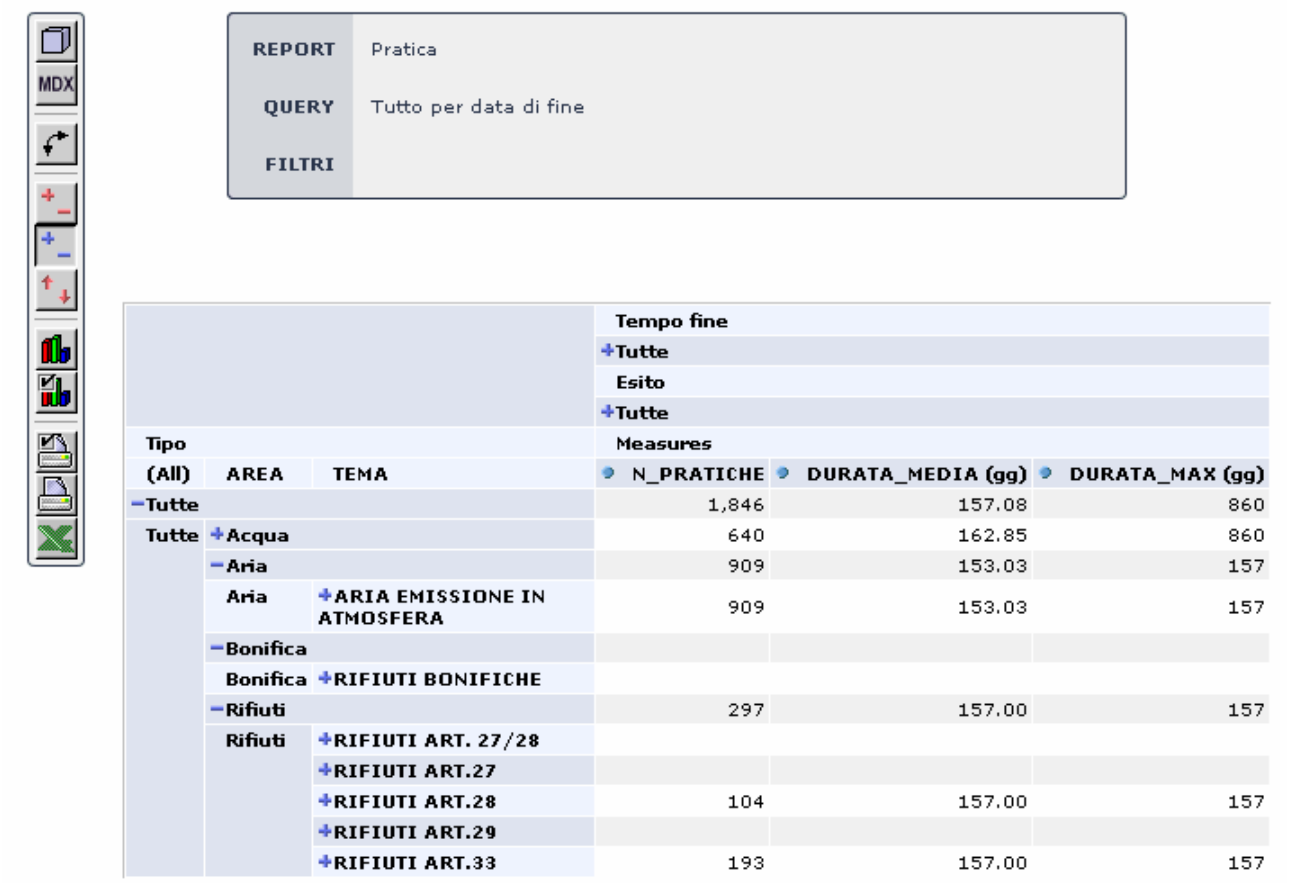
2.6.1 JPIVOT

JPivot è una libreria di tag JSP personalizzati che leggono una tabella OLAP e permette agli utenti di effettuare tipiche operazioni di navigazioni degli strumenti OLAP come slice, dice, drill down o roll up.

È in grado di utilizzare OLAP server come Mondrian o supporta lo standard XMLA per l'accesso a datasource.

JPivot è stato progettato per lavorare con diversi OLAP server, quindi JPivot non usa le API di Mondrian direttamente. Definisce al suo interno un modello del OLAP in termini di interfacce nei package olap.model e olap.navi. Per permettere a JPivot di lavorare con Mondrian, queste interfacce dovranno essere implementate usando le classi di Mondrian.

JPivot usa WCF (Web Component Framework) che supporta la creazione di componenti per l'interfaccia utente attraverso XML e XSLT.



			Tempo fine		
			+Tutte		
			Esito		
			+Tutte		
Tipo			Measures		
(All)	AREA	TEMA	N_PRATICHE	DURATA_MEDIA (gg)	DURATA_MAX (gg)
−Tutte			1,846	157.08	860
Tutte	+Acqua		640	162.85	860
	−Aria		909	153.03	157
	Aria	+ARIA EMISSIONE IN ATMOSFERA	909	153.03	157
	−Bonifica				
	Bonifica	+RIFIUTI BONIFICHE			
	−Rifiuti		297	157.00	157
	Rifiuti	+RIFIUTI ART. 27/28			
		+RIFIUTI ART.27			
		+RIFIUTI ART.28	104	157.00	157
		+RIFIUTI ART.29			
		+RIFIUTI ART.33	193	157.00	157

Figura 12 - Esempio dell'interfaccia di JPivot

2.6.2 WCF (WEB COMPONENT FRAMEWORK)

Le classi WCF aiutano a creare, presentare e validare form HTML attraverso documenti XML e XSLT. WCF incapsula componenti riusabili che sono stati sviluppati per JPivot. Il focus dei componenti è stato concentrato sull'interfaccia grafica. Sono stati sviluppati componenti come tabelle, alberi o form completi. Per i singoli elementi come bottoni o caselle di testo si possono usare le Java Server Faces. WCF si integra facilmente con JSF.

WCF è stato progettato per integrarsi facilmente con applicazioni web già esistenti come portali (ad esempio JetSpeed), JSF, Struts e altre. Per integrare la parte di view sono usati tag JSP personalizzati. Il controller lavora come un filtro servlet, quindi può essere utilizzato insieme a applicazioni esistenti.

WCF fornisce un controller, che identifica, per esempio, il pulsante che l'utente ha premuto e invoca l'ascoltatore che è stato registrato per quel pulsante e sincronizza le proprietà del java bean con gli elementi del form, formattando i dati per la presentazione, analizzandoli e validandoli.

I package presenti sono:

- Controller: permette di registrare un RequestListener a cui vengono notificate le azioni dell'utente. Per esempio, un RequestListener può essere registrato per un pulsante o per un tag ancora (<a href="…">);
- Format: un meccanismo estensibile di convertire oggetti java nelle rispettive rappresentazioni in stringhe e dare il supporto i18n per la conversione e i messaggi di errore;
- Convert: riempie i valori degli elementi di un form dalle proprietà del bean e aggiorna le proprietà del bean con gli input dell'utente;
- Ui: fornisce delle API per creare e manipolare alberi DOM che sono conformi alla DTD di xoplon;
- Form: gestione dei form, sincronizzando il form e il bean.

Il package Controller permette di registrare dei RegisterListener che vengono notificati delle azioni dell'utente. RequestListener sono delle classi che vogliono ricevere le notifiche di eventi di alcune richieste http. Per esempio, un RequestLister può essere registrato per uno specifico URL come <a href="…">. In questo caso l'ascoltatore sarà invocato tutte le volte in cui un utente clicca su un tag ancora. Un altro esempio è un pulsante in un form. Ci può essere un RequestListener per ogni bottone.

Un RequestListener può ricevere notifiche per richieste che contengono:

- una specifica coppia nome-valore;
- uno specifico nome con valore arbitrario (Ad esempio <INPUT type="button" ….>);
- uno specifico valore con un nome arbitrario.

Un Dispatcher determina il RequestListener dai parametri della richiesta (coppie nome-valore) e notifica il match.

I Dispatcher possono essere innestati. Un Dispatcher figlio, che è stato aggiunto a un Dispatcher attraverso il metodo addRequestListener(), risponderà a tutte le richieste che riceve dal Dispatcher padre e le inoltra ai sui ascoltatori. L'uso comune è quello di avere un singolo Dispatcher di root nella servlet Controller, che riceve tutte le richieste. Il successivo Dispatcher nella *gerarchia* gestisce gli eventi di una pagina, il successivo di un form e l'ultimo un singolo pulsante. Tutte le gerarchie possono essere scelte e codificate nei parametri dell'URL.

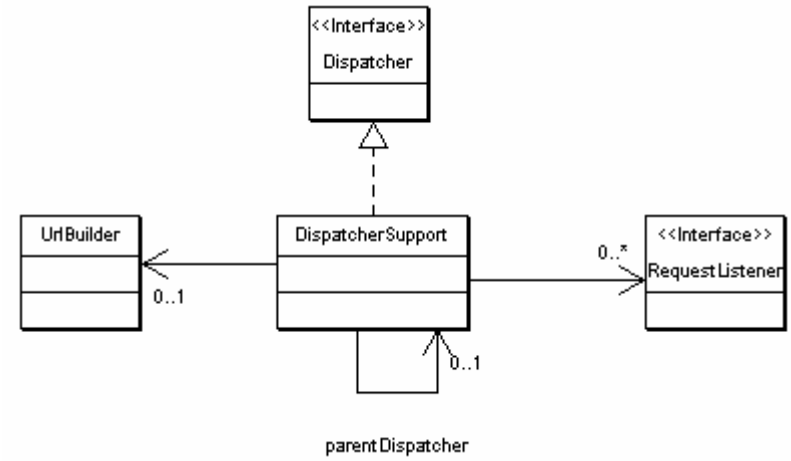


Figura 13 - Architettura WCF

Il Controller è implementato da RequestFilter e SessionListener. Il controller fornisce accesso al Dispatcher e opzionalmente può redirigere la richiesta su altre pagine. RequestFilter e SessionListener sono accessibili attraverso il controller ma non direttamente.

JPivot e WCF utilizzano "expression language" per accedere semplicemente a delle proprietà tramite il loro path. Ad esempio #{customer.address.name} viene tradotto in:

```
session.getAttribute("customer").getAddress().getName()
```

WCF utilizza i jakarta commons beanutils. Con JSP 2.0 si può usare la notazione <code>#{}</code> per definire espressioni per attributi WCF e la notazione <code>\${}</code> per definire espressioni JSP EL.

Le espressioni JSP attualmente non possono essere utilizzate insieme a tag WCF. Tutti gli attributi dei tag che contengono <code>rtexprvalue</code> vengono settate a falso.

2.7 IL PROBLEMA DELLA MATERIALIZZAZIONE DELLE VISTE

Il problema della materializzazione delle viste riguarda quel processo di selezione dell'insieme delle viste tra tutte quelle possibili la cui materializzazione produca vantaggi prestazionali per il sistema di data warehouse. La scelta di quali viste materializzare deve rispettare un insieme di obbiettivi di progetto che non riguarda solo l'aspetto prestazionale, ma anche vincoli di spazio e di tempo di aggiornamento.

La precedente definizione pone l'attenzione su due elementi importanti:

- la definizione degli obiettivi della materializzazione;
- la tecnica di selezione da utilizzare.

Secondo Theodoratos e Bouzeghoub (vedi bibliografia [6]), un obiettivo può essere sia la minimizzazione di una funzione di costo sia un vincolo; questi ultimi si suddividono in vincoli orientati al sistema e vincoli orientati agli utenti. Le funzioni che possono essere minimizzate si possono raggruppare nelle seguenti tipologie:

FUNZIONI COSTO	Costo di valutazione del carico di lavoro	Questo tipo di tecniche di selezione delle viste richiedono in input il carico di lavoro stimato a cui un *data mart* dovrà rispondere. Il costo totale del carico di lavoro può essere calcolato come somma pesata del costo delle diverse interrogazioni, dove il peso associato a ciascuna interrogazione risente della frequenza con cui viene sottoposta al sistema e della sua importanza per l'utente. In letteratura sono state proposte diverse funzioni di costo: alcune basate sul fatto che il costo di una stessa interrogazione varia proporzionalmente alla dimensione delle viste stesse; altre, più sofisticate, calcolano una stima dell'effettivo numero di pagine di dati a cui è necessario accedere per rispondere all'interrogazione.
	Costo di manutenzione delle viste	Le viste materializzate richiedono periodici aggiornamenti per riflettere le modifiche intervenute nelle sorgenti operazionali. Il costo di manutenzione è il costo delle interrogazioni necessarie a propagare gli aggiornamenti. Il calcolo di questo costo è complesso a causa delle diverse soluzioni possibili.
VINCOLI DI SISTEMA	Spazio di memorizzazione	Un vincolo che rappresenta normalmente uno dei principali vincoli alla materializzazione delle viste è lo spazio di memorizzazione su disco a disposizione del *data mart*. Per calcolare questo vincolo è necessario disporre di una funzione per la stima della dimensione delle viste. Bisogna considerare anche altri modi per migliorare le prestazioni del sistema che richiedono spazio su disco (ad esempio gli indici). La scelta di come distribuire lo spazio diventa quindi una scelta progettuale importante.
	Tempo di aggiornamento	L'aggiornamento del *data mart* viene eseguito normalmente quando il data warehouse e fuori linea (di notte) e comunque comporta sempre una forte diminuzione delle prestazioni del sistema. Inoltre, contemporaneamente all'aggiornamento delle viste, normalmente il sistema viene sottoposto ad altre operazioni periodiche (backup, sincronizzazione, …). Di conseguenza il tempo necessario dedicato a questo tipo di compito è limitato (se si aggiornano i dati di notte, il numero di ore che è possibile dedicare a questo compito sarà limitato dalle ore libere (6-8) e dal tempo occupato dalle altre attività). Questo tipo di vincoli richiede un numero di viste materializzate tale da permettere il loro aggiornamento nel tempo a disposizione.

VINCOLI UTENTE	Tempo di risposta delle interrogazioni	Si tratta di un tipo di vincolo con cui l'utente pone un limite massimo al suo tempo d'attesa, cioè tra il tempo intercorso tra il momento di sottomissione dell'interrogazione e la risposta del sistema. Per specificare questo tipo di vincolo dovrà essere specificato il valore del tempo massimo di risposta per le diverse interrogazioni (normalmente quelle caratteristiche per l'applicazione).
	Data di aggiornamento delle risposte	Si tratta di un tipo di vincolo con cui l'utente limita il tempo intercorso tra due aggiornamenti di una vista consecutivi. Questo tipo di vincolo serve per definire la "freschezza" con cui i dati sono proposti all'utente.

Tabella 12 - Tipi di costi della materializzazione delle viste

È facile notare che tutti gli obiettivi elencati in precedenza siano in contrasto tra loro ed è quindi facile intuire che la soluzione dovrà tenere conto di ognuno di essi cercando un compromesso.

È ovvio che se i vincoli espressi sono troppo restrittivi, può accadere che il problema della materializzazione delle viste non presenti soluzioni.

Una prima assunzione è che, nel nostro contesto, c'è una netta separazione tra il livello delle viste e il livello operazionale.

Ciò significa che le viste aggregate sono ottenute aggregando i record dalla fact table di base. Il potere informativo delle viste non sarà quindi mai superiore a quello della fact table.

Diversamente, altre architetture considerano un *data mart* come una collezione di viste aggregate indipendenti tra cui non è possibile identificare un sottoinsieme di viste di base che consentono di rispondere, sebbene in tempi più lunghi, a tutte le interrogazioni possibili.

Una seconda assunzione di rilievo consiste nel considerare sempre disponibile un carico di lavoro di riferimento, ossia che esista un insieme di interrogazioni (quelle caratteristiche) che essendo quelle eseguite più frequentemente dagli utenti generino questo tipo di carico di lavoro.

Questo contrasta con l'idea dei sistemi OLAP che presuppongono un carico di lavoro estemporaneo e in continua evoluzione. Tuttavia, la pratica ha dimostrato che è possibile individuare un insieme di interrogazioni che caratterizzano e rappresentano il carico totale.

Il problema della materializzazione delle viste viene normalmente formulato come il problema di minimizzare il tempo di risposta, cioè le prestazioni percepite dall'utente, nel rispetto dei vincoli di sistema (spazio su disco e tempo di aggiornamento).

La complessità di questo problema deriva dalla dimensione dello spazio di ricerca della soluzione, che cresce esponenzialmente con il numero degli attributi dimensionali. Per esempio, in assenza di gerarchie, un fatto con n dimensioni determina 2^n possibili viste. Le tecniche per la soluzione del problema operano in due fasi:

- nella prima vengono individuate, tra tutte le possibili viste materializzabili, quelle effettivamente utili per un dato carico di lavoro;
- successivamente, se ne determina il sottoinsieme che minimizza la funzione di costo nel rispetto dei vincoli di sistema.

Una vista di uno schema è determinata univocamente dal suo pattern di aggregazione a meno delle *misure* in esso contenute.

Tra le varie strutture dati per la codifica dei pattern validi per un certo schema, il reticolo multidimensionale è largamente utilizzato.

Un reticolo multidimensionale è il reticolo che modella l'ordinamento parziale di roll-up tra i pattern. I pattern inseriti nel reticolo sono tutti quelli validi per l'aggregazione, ossia tutti gli insiemi di attributi dimensionali in cui non esistono dipendenze funzionali tra gli elementi. Le linee nel grafo rappresentano l'ordinamento parziale tra i pattern.

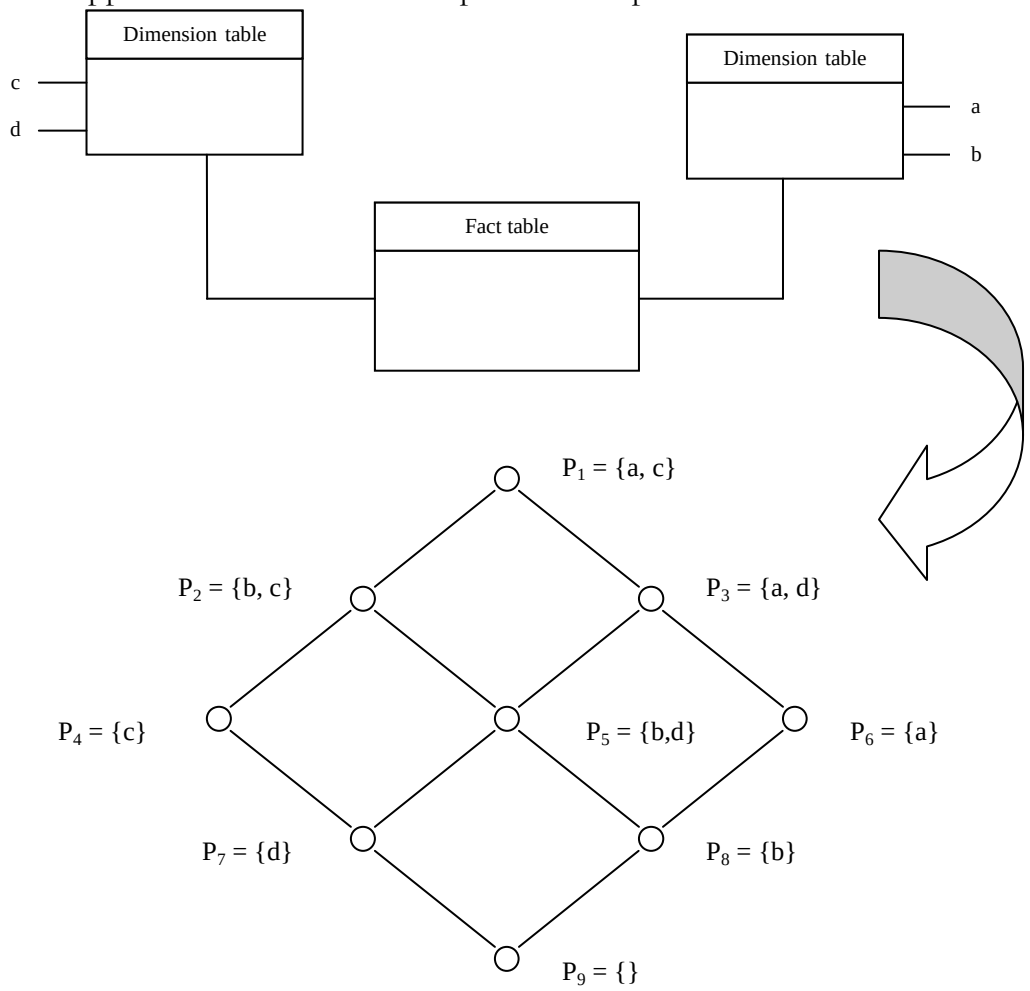


Figura 14 - Reticolo corrispondente al cubo multidimensionale con dimensioni a e c e gerarchie a → b e c → d

La dimensione del reticolo multidimensionale cresce esponenzialemente rispetto al numero di attributi presenti nel cubo ed è quindi difficilmente applicabile a problemi reali di una certa complessità.

Nella figura successiva sono rappresentate le viste candidate per un carico di lavoro composto da tre interrogazioni Q_1, Q_2 e Q_3 eseguibili rispettivamente sui pattern P_6, P_7 e P_8. Oltre ai pattern corrispondenti alle interrogazioni risultano candidati i pattern P_5 e P_3 (come ottenere queste ultime sarà spiegato approfonditamente nel paragrafo seguente), infatti:

- P_5 consente di risolvere sia Q_2 che Q_3 e non esiste un altro pattern P_x tale che $P_x \leq P_5$ e per il quale valga la stessa proprietà;

- P_3 consente di risolvere tutte e tre le interrogazioni e non esiste un altro pattern P_y tale che $P_y \leq P_3$ e per il quale valga la stessa proprietà.

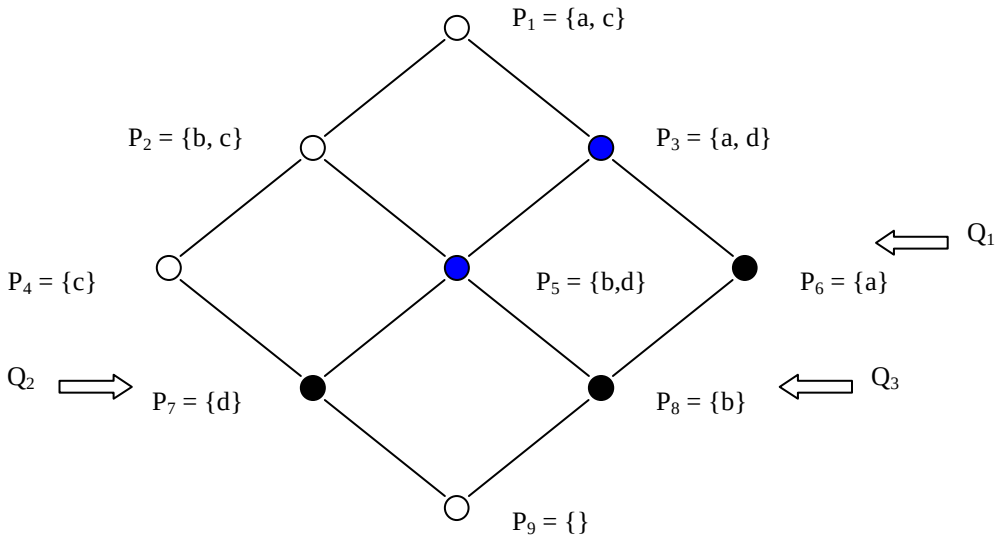


Figura 15 - In blu e nero le viste candidate

L'ottimizzazione assoluta del carico di lavoro si ottiene materializzando, tra le viste candidate, tutte quelle corrispondenti alle singole interrogazioni. Nei casi reali, però, i vincoli di sistema rendono inattuabile questa soluzione: sarà quindi necessario determinare un sottoinsieme delle viste candidate che garantisca buone prestazioni nel rispetto dei vincoli. Nell'esempio proposto sono state analizzate tre possibili soluzioni con una valutazione puramente qualitativa.

Soluzione	*Viste materializzate*	*Costo*	*Spazio*	*Tempo*
1	P_6, P_7 e P_8	basso	alto	alto
2	P_3	alto	basso	basso
3	P_6 e P_7	medio	medio	Medio

Tabella 13 - Vantaggi di varie soluzioni di materializzazione delle viste

In sintesi, è consigliabile materializzare una vista quando essa risolve direttamente una interrogazione molto frequente e/o permette di risolvere parecchie interrogazioni. Non è consigliabile materializzare una vista quando il suo pattern è simile a una vista già materializzata e/o il suo pattern è molto fine e/o la materializzazione non riduce di almeno un ordine di grandezza il costo del carico di lavoro (vedi bibliografia [15]).

2.7.1 SELEZIONE DELLE VISTE DA MATERIALIZZARE

In questo paragrafo sarà approfondito il problema della scelta delle viste da materializzare per migliorare le prestazioni di un server OLAP.

Un database multidimensionale è un magazzino che fornisce un ambiente integrato per interrogazioni che richiedono complesse aggregazioni di una grande quantità di dati storici. In

questi database le informazioni sono organizzate tipicamente con una struttura a stella o sue variazioni.

Questa semplice struttura può essere rappresentata dal seguente diagramma, in cui le entità D_i rappresentano le dimensioni che sono connesse dall'entità F che è la Fact Table.

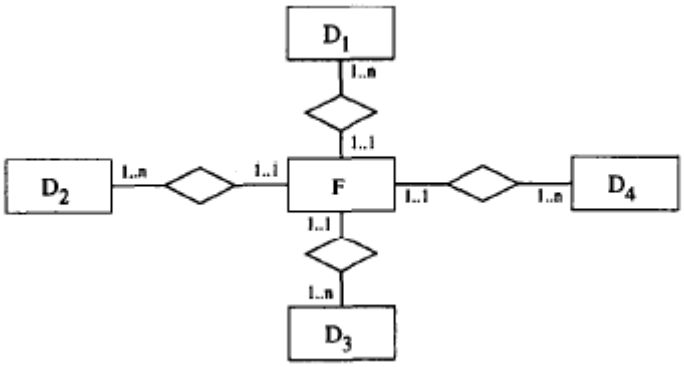


Figura 16 - Diagramma ER di una struttura a stella

Tutte le tabelle delle dimensioni contengono tutte le informazioni che le descrivono, mentre la fact table correla tutte le dimensioni e contiene le informazioni sugli attributi che riguardano l'intersezione delle dimensioni.

Siccome la computazione di tutti gli elementi del cubo è molto onerosa, per garantire tempi di risposta accettabili alle interrogazioni si creano delle pre-aggregazioni. Dal lato opposto, non è possibile materializzare tutte le possibili aggregazioni richieste perché sarebbero troppo onerose da aggiornare.

Sono state proposte alcune tecniche per selezionare le migliori pre-aggregazioni da materializzare. L'algoritmo proposto da [1] funziona molto bene per database di medie dimensioni. Nell'articolo citato è stato considerato un esempio pratico con gradi quantità di dati su un supermercato che vende una grande varietà di prodotti. Il database memorizza informazioni su ogni vendita in ogni negozio, per ogni giorno, considerando promozioni, di ogni prodotto venduto. Si possono identificare le seguenti dimensioni:

- Prodotti, che possono essere caratterizzati da più di 50 attributi;
- Store, che caratterizzano ogni punto vendita. Questa dimensione può essere caratterizzata da più di 20 attributi;
- Tempo, che fornisce il dettaglio appropriato per consentire le analisi sul database. La dimensione tempo può avere più di 15 attributi che la caratterizzano;
- Promozione, che descrive le caratteristiche di ogni promozione sui prodotti. Questa dimensione può essere caratterizzata da più di 10 attributi.

La fact table fornisce le informazioni sulle vendite che necessitano alle analisi finanziarie richieste. La tabella include anche gli identificatori di tutte le dimensioni.

Definiamo un database multidimensionale come una collezione di relazioni $D_1,\ldots,D_n,F$ dove ogni D_i è una tabella delle dimensioni e ognuna è caratterizzata da un identificatore univoco all'interno della dimensione d_i e dove F è la fact table che connette tutte le tabelle $D_1,\ldots,D_n$. L'identificatore della relazione F è data da tutte le chiavi esterne $d_1,\ldots,d_n$ di tutte le tabelle dimensionali che connette. Lo schema di F contiene un insieme di attributi addizionali V che rappresentano i valori a cui applicare le funzioni di aggregazione.

Le tabelle dimensionali possono contenere gerarchie. Sia D una tabella dimensionale con identificatore d. Definiamo una *gerarchia* di attributi come un insieme di dipendenze funzionali FDD=$\{fd_0, fd_1, \ldots, fd_n\}$ dove ogni fd_i è caratterizzata da due insiemi di attributi $A^l_i \subset Attr(D)$ e $A^r_i \subset Attr(D)$ (rispettivamente chiamate lato sinistro e lato destro della dipendenza funzionale). La dipendenza funzionale può essere rappresentata come fd_i: $A^l_i \rightarrow A^r_i$.

Ogni dipendenza funzionale fd_i è un vincolo sui contenuti della tabella dimensionale D: per ogni coppia di tuple $t_1,t_2 \in D$, $t_1[A^l_i] = t_2[A^l_i] \Rightarrow t_1[A^r_i] = t_2[A^r_i]$. Una dipendenza fd_0 con $A_l = \{d\}$ e con $A^r_0 = \{Attr(D) - d\}$ è sempre presente nell'insieme FDD. Le dipendenze funzionali devono essere acicliche.

Una *gerarchia* di attributi si definisce "tree-like" se non riporta tutte le dipendenze transitive, tutti gli attributi compaiono al massimo una volta nel lato destro delle dipendenze funzionali e nel lato sinistro solo attributi singoli. La chiusura transitiva di una *gerarchia* "like-tree" e una *gerarchia* di attributi "like-tree" completa (FD^{TC}).

Consideriamo, ad esempio, la dimensione Store del database descritto in precedenza, con chiave s e con un insieme di attributi {z, c, st, n} rappresentano rispettivamente il codice di avviamento postale, nazione, regione, e il numero di venditori. La dimensione ha la seguente *gerarchia* di attributi "like-tree":

$\{fd_0: s \rightarrow \{z, c, st, n\}, fd_1: z \rightarrow c, fd_2: c \rightarrow st\}$

Definiamo *gerarchia* di attributi di un database multidimensionale FDDB è l'unione delle gerarchie degli attributi di tutte le tabelle dimensionali Dj contenute nel database multidimensionale.

Interrogazioni di analisi, normalmente richiedono il calcolo di funzioni di aggregazione sui dati.

Ad esempio, consideriamo il database multidimensionale già considerato in precedenza:

```
MDDB = {Product, Store, Time, Promotion, F}
```

Dove ogni tabella dimensionale ha i propri attributi e la fact table F = {p, s, d, r, f} ha gli attributi p, s, d e r come chiavi esterne sulle tabelle dimensionali e f rappresenta la quantità di vendite. Le seguenti query possono essere sottoposte al database multidimensionale:

- Q_1 = totale delle vendite per prodotto;
- Q_2 = totale delle vendite per prodotto e Store;
- Q_3 = totale delle vendite per prodotto e giorno;
- Q_4 = totale delle vendite per prodotto, Store e giorno.

Le interrogazioni che consideriamo vengono trasformate in interrogazioni di tipo select-join-groupby, con alcune restrizioni nei predicati di selezione e di join. In particolare, i predicati di selezione sono semplici predicati di confronto tra gli attributi dimensionali e un valore costante. Questi predicati di selezione producono uno "slice" dei dati.

Consideriamo una query q che restituisce il totale delle vendite per particolari store, raggruppati per prodotto. La query q può trovare risposta accedendo ai risultati di una query che ritorna le vendite raggruppate per prodotto e Store, e selezionando in esso solo le tuple relative allo specifico store. In questo caso, possiamo concentrarci nella parte della query che calcola gli aggregati.

Le operazioni di join possono essere effettuate tra la fact table e tutte le sue dimensioni. Consideriamo la notazione standard di un'interrogazione SQL con clausola group by e con funzioni di aggregazione. Gli attributi su cui fare aggregazioni possono essere attributi delle tabelle dimensionali o della fact table.

Data una query q definiamo la query come caratterizzata dall'insieme degli attributi su cui effettuare il raggruppamento A. Possiamo rappresentare la query come q^A.

Una caratteristica fondamentale delle query in database multidimensionali è che spesso è possibile riutilizzare i risultati della query per rispondere a altre query. Nell'esempio precedente possiamo usare i risultati della query q_2 per rispondere alla query q_1, aggregando le vendite per tutti gli Store.

Il riuso di query è strettamente relazionato a un nuovo operatore. Questo operatore, riceve come input una tabella T, un insieme di attributi di aggregazione A e una funzione f, calcolare l'unione dei risultati delle query valutando f, avendo come attributi di aggregazione tutte le possibili combinazioni degli attributi di A.

Dato un database multidimensionale MDDB = $\{D_1, D_2, \ldots, D_n, F\}$ il lattice del cubo rappresenta l'insieme di tutte le possibile query con raggruppamenti che possono essere definite sulle chiave esterne di F. Questo lattice è caratterizzato dai seguenti elementi:

- una relazione d'ordine definita dalla comparazione tra gli insiemi degli attributi di aggregazione (ad esempio: $q^{A1} \leq q^{A2} \leftrightarrow A_1 \subseteq A_2$);
- l'operatore di "meet" è l'unione degli attributi di aggregazione;
- l'operatore di "join" è l'intersezione degli attributi di aggregazione;
- la query che raggruppa su tutte le chiavi esterne è l'elemento più altro;
- la query che calcola le funzioni di aggregazione su tutte le tuple di F è l'elemento più basso (quella con l'insieme degli attributi di aggregazione vuoto).

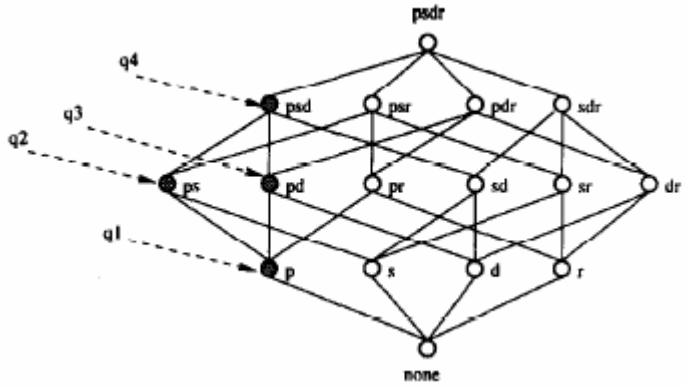


Figura 17 - Esempio di lattice di un cubo

La presenza di più dimensioni rende il problema più complesso. Il primo aspetto è che incrementare il numero di potenziali gruppi di attributi, che incrementa in modo esponenziale il numero degli elementi del lattice. Il secondo aspetto è la presenza di gerarchie, che permette di rimuovere alcuni elementi dal lattice. Consideriamo una query che raggruppa su una chiave dimensionale d_j e anche su un attributo a_j della stessa dimensione D_j. Visto che esiste una dipendenza funzionale da d_j a a_j, una query che raggruppa su $\{d_j, a_j\}$ produce gli stessi risultati di una query che raggruppa su $\{d_j\}$.

Siano q_x^{Ax} e q_y^{Ay} due query e sia FD_{DB} la *gerarchia* degli attributi del database multidimensionale. L'operatore antenato (rappresentato dal simbolo $\oplus$) è definito dal seguente algoritmo:

```
Operatore ⊕: q_x^Ax ⊕ q_y^Ay ⇒ q_z^Az

A_z := A_x ∪ A_y
Per ogni fd_i ∈ FD^TC_DB
      Per ogni a_j ∈ A_i^r
            If ({a_i^l} ∪ a_j) ⊆ A_z
                  A_z := A_z - a_j
Ritorna q^Az
```

L'operatore costruisce l'unione degli attributi caratterizzanti delle interrogazioni ed elimina tutti gli elementi per i quali esiste una dipendenza funzionale nella FD_{DB}.

Il risultato di applicare l'operatore $\oplus$ alle query q_x e q_y è la più piccola query che contiene tutte le informazioni necessarie per rispondere a entrambe le query. Se applicato nel modo riflessivo ($q_x^{Ax} \oplus q_x^{Ax}$) rimuove tutti gli attributi ridondanti.

Consideriamo la *gerarchia* sulla dimensione Store dell'esempio predente. Le query $q^{\{n\}}$, $q^{\{c\}}$ e $q^{\{st\}}$ possono essere calcolate a partire da $q^{\{c,n\}} = q^{\{n\}} \oplus q^{\{c\}} \oplus q^{\{st\}}$.

Definiamo l'operatore discendente (rappresentato dal simbolo Θ) è definito dal seguente algoritmo:

```
Operatore Θ: q_x^Ax Θ q_y^Ay ⇒ q_z^Az

Per ogni fd_i ∈ FD^TC_DB
      Se {a_i^l} ⊆ A_x
            A_x := A_x ∪ A_i^r
      Se {a_i^l} ⊆ A_y
            A_y := A_y ∪ A_i^r
A_z := A_x ∩ A_y
Ritorna q_z^Az ⊕ q_x^Ax
```

L'algoritmo, per prima cosa estende gli argomenti A_x e A_y con tutti gli attributi che possono essere derivati da essi. Considera l'intersezione dei due insiemi calcolati in precedenza e rimuove da questo ultimo gli attributi che sono alla destra di una dipendenza funzionale quando gli attributi alla sinistra sono contenuti nell'insieme.

L'operatore discendente calcola il più grande insieme di attributi caratterizzanti le query che possono essere calcolati sia da q_x e q_y.

Sia $\{D_1, \ldots, D_n, F\}$ un database multidimensionale e sia FD_{DB} la *gerarchia* di attributi del database. Consideriamo l'insieme delle query caratterizzate dalla combinazione di attributi di $\{D_1, \ldots, D_n, F\}$ eccetto la combinazioni che contendono entrambi i lati di una dipendenza funzionale $fd_i \in FD_{DB}$. Questo insieme di query identifica un lattice dove:

- la relazione d'ordine è data dalla seguente definizione:

$$q^{A1} \leq q^{A2} \leftrightarrow (A_1 \Theta A_2) = A_1 \leftrightarrow (A_1 \oplus A_2) = A_2$$

- $\oplus$ è l'operatore di "meet";
- Θ è l'operatore di "join";

- la query che raggruppa su tutte le chiave esterne di F $\{d_1, \ldots, d_n\}$ è l'elemento più alto;
- la query che aggrega su tutte le tuple di F è l'elemento più basso (insieme vuoto di attributi di raggruppamento).

Siano q_i e q_j due query in un database multidimensionale. Definiamo l'ultimo discendente comune come una query (la più specializzata) che può essere usata per rispondere alla query. Definiamo una query q_i dipendente da q_j se la query q_j può essere utilizzata per rispondere alla query q_i.

Il numero di aggregazioni possibili in un lattice dipende dal numero di attributi delle dimensioni del database multidimensionale e il numero e la struttura degli attributi delle gerarchie. In assenza di gerarchie nelle tabelle dimensionali il numero di possibili aggregazioni può essere calcolato con la seguente formula:

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

dove i è il numero di dimensioni del database multidimensionale e n_i è il numero di attributi non chiave della dimensione D_i. In presenza di gerarchie, il numero di possibili aggregazioni diminuisce e la formula precedente costituisce un limite superiore. Prendiamo come esempio, il caso considerato in precedenza con $n_{product} = 12289$, $n_{store} = 8193$, $n_{time} = 129$ e $n_{promotion} = 1025$. Il numero di possibili aggregazioni risulta essere $1{,}3313*10^{13}$.

Il problema da risolvere è trovare l'insieme delle viste da materializzare per massimizzare le prestazioni del database multidimensionale in risposta a un insieme di interrogazioni rappresentativo. Il problema consiste nel trovare il miglior compromesso tra il vantaggio di avere una vista aggregata per migliorare le prestazioni e il costo di mantenere aggiornate le viste materializzate.

Dato un database multidimensionale, un insieme di query Q e un insieme di frequenze delle query F, il problema della materializzazione delle viste viene rappresentato con Θ(DB,Q,F). La soluzione del problema Θ(DB,Q,F) è un insieme di viste M da materializzare. Una soluzione $M = \varnothing$ è sempre possibile e rappresenta la situazione in cui nessuna vista addizionale viene materializzata. In questo caso tutte le query vengono risolte utilizzando direttamente la fact table.

L'identità dell'insieme ottimale M, deve essere definito da una funzione di costo. La funzione di costo è composta da due parti: il costo della query e il costo degli aggiornamenti. Sia F l'insieme delle frequenze di ogni query contenuta in Q. Sia $Cq_i(M)$ il costo per rispondere alla query q_i dall'insieme delle viste materializzate M. Allora il costo totale delle interrogazioni sarà:

$$C_Q(Q,M,F) = \sum_{q_i \in Q} f_{q_i} C_{q_i}(M)$$

Consideriamo lo stesso insieme di viste materializzate M. Sia f_{mi} la frequenza con cui la vista $m_i \in M$ viene modificata e $C_u(m_i)$ è il costo di aggiornamento. Il costo totale di aggiornamento è:

$$C_M(M,F) = \sum_{m_i \in M} f_{m_i} C_u(m_i)$$

dato il problema di materializzazione delle viste descritto in precedenza Θ(DB,Q,F), il costo di una soluzione M è la somma del costi delle query e dei costi degli aggiornamenti.

$$C(Q,M,F) = C_Q(Q,M,F) + C_M(M,F)$$

Esistono molte tecniche per descrivere il costo di una query (vedi, ad esempio, bibliografia [1]), alcune molto semplici altre molto complesse. La funzione $Cq_i(M)$ ritorna il costo di calcolo della query q_i dato un insieme di viste materializzate M. Ipotizziamo che il costo di ogni query dipenda da un solo elemento contenuto in M e che la funzione costo sia monotona al crescere della dimensione della vista materializzata da cui dipende la query.

Definiamo la funzione costo di una query $Cq_i(M)$ come restringibile se il costo che restituisce è sempre uguale al minimo valore dei $Cq_i(m_j)$ per ogni $m_j \in M$. Tutte le query introdotte nell'esempio precedente sono riducibili.

Data una query q_i, una funzione di costo riducibile $Cq_i(M)$ e un insieme di viste materializzate M, definiamo la materializzazione meno costosa m_j come quella per cui $Cq_i(M)=Cq_i(m_j)$.

Una funzione di costo Cq_i si definisce monotona se per ogni $v_j,v_k \in$ MD-lattice dalle quali q_i dipende, $|v_j| < |v_k| \rightarrow Cq_i(v_j) < Cq_i(v_k)$ dove $|v|$ rappresenta la cardinalità di v e la freccia denota l'implicazione logica. Dall'osservazione delle cardinalità delle viste nel lattice ne segue che una funzione monotona garantisce anche che $v_j \leq v_k \rightarrow Cq_i(v_j) < Cq_i(v_k)$.

Consideriamo l'inserimento di tuple nella fact table o nelle tabelle dimensionali come il tipo prevalete di modifiche che avvengono in un database multidimensionale. Assumiamo che tutti gli inserimenti avvenuti non violino le regole di integrità referenziale tra le tabelle dimensionali e la fact table.

Possiamo semplificare il modello di costo di aggiornamento assumendo un'unica frequenza di aggiornamento f_u valida per tutte le viste materializzate, dove f_u rappresenta la frequenza di inserimento nella fact table. La funzione costo di aggiornamento può quindi essere riscritta come:

$$C_M = f_u \sum_{m_j \in M} c_u(m_j)$$

Una funzione di costo di aggiornamento si definisce monotona se per ogni $m_j, m_k \in M$, $|m_j| < |m_k| \rightarrow c_u(m_j) \leq c_u(m_k)$ dove $|m_j|$ rappresenta la cardinalità di m_j. Consideriamo di avere a disposizione una funzione di costo di aggiornamento monotona.

In confronto al numero di nodi di un lattice multidimensionale, il numero delle query rappresentative è estremamente piccolo. Questa considerazione suggerisce che solo alcune viste siano rilevanti quando si decide come minimizzare il costo totale. L'idea di questa tecnica di riduzione è quella di considerare solo le viste del lattice che, quando vengono materializzate, possono fornire qualche contributo per ridurre il costo totale. Possiamo chiamare esse viste candidate.

Definiamo una vista v_i contenuta in un lattice multidimensionale come una vista candidata se verifica una delle seguenti due condizioni:

- la vista v_i è associata a qualche query q_i;
- esistono due viste candidate v_j e v_k, e v_i è un antenato di v_j e v_k.

Sia v_i una vista candidate di un lattice multidimensionali. Quindi, scegliendo v_i per essere materializzata otterremo qualche beneficio per ridurre il costo totale. Esistono due casi.

- v_i è associata alla query q_i. Partendo dalla definizione di costo di una soluzione, possiamo scrivere la seguente formula, che identifica la frequenza f_{qi} che rende conveniente la materializzazione della vista v_i quando un insieme di viste M è già materializzato.

$$f_{q_i} < f_u \frac{c_u(v_i)}{c_{q_t}(v_t) + c_{q_i}(v_i)}$$

  dove $v_t \in M$ rappresenta la vista materializzata meno costosa che può essere usata per rispondere alla query q_i.

- Esistono almeno due viste candidate, v_j e v_k per cui v_i è un loro antenato. Questa condizione è sufficiente per mostrare che esiste almeno un caso in cui materializzare v_i fornisce alcuni benefici. Assumiamo che esistano due query q_j e q_k associate alle viste v_j e v_k. Il contributo delle query q_j e q_k e delle viste v_j e v_k alla funzione costo c(Q,M,F), quando v_j e v_k sono materializzata e v_i non lo è, diventa:

$$C_1 = f_u \cdot c_u(v_j) + f_{q_j} \cdot c(v_j) + f_u \cdot c_u(v_k) + f_{q_k} \cdot c(v_k)$$

  Il contributo al costo C(Q,M,F) se v_i è materializzata e v_j e v_k no, con v_i è la materializzazione più conveniente per entrambe le query q_j e q_k, è:

$$C_2 = f_u \cdot c_u(v_i) + f_{q_j} \cdot c_{q_j}(v_i) + f_{q_k} \cdot c_{q_k}(v_i)$$

  Scegliere vi per la materializzazione diminuisce il costo totale se C1 > C2, quindi:

$$f_u > \frac{f_{q_j} \cdot (c_{q_j}(v_i) - c_{q_j}(v_j)) + f_{q_k} \cdot (c_{q_k}(v_i) - c_{q_k}(v_j))}{c_u(v_j) + c_u(v_k) - c_u(v_i)}$$

  con l'ipotesi che $c_u(v_j) + c_u(v_k) - c_u(v_j) > 0$. L'intuizione è che se il costo di aggiornare le viste v_j e v_k è maggiore che il costo di aggiornamento di v_i per una frequenza di aggiornamento abbastanza elevata, conviene materializzare la vista v_i invece di v_j e v_k.

In più, per essere sicuri che le viste candidare sono solo quelle rilevanti per il processo di decisione delle materializzazioni, bisogna provare che le viste non candidate non diminuiranno in nessun caso il costo totale.

Definiamo una vista non candidata v_i con almeno una vista candidata dipendente da essa come direttamente dipendente l'unica vista candidata v_j quella per cui le altre viste dipendenti di v_i sono dipendenti anche da v_j.

Possiamo determinare la vista candidata direttamente dipendete di v_j prendendo l'insieme V' di tutte le viste candidate dipendenti da v_j e calcolando il discendente di tutte le viste in V'. Questa vista è unica (perché il discendente è sempre identificato), è una vista candidata (perché discendente di un insieme di viste candidate) e appartiene a V'.

La scelta se materializzare o meno una vista è dominata dalla scelta se una vista è candidata o no.

Assumiamo che esista una vista non candidata v_i che appartiene all'insieme ottimo delle viste da materializzare M, e verifichiamo di giungere a una contraddizione. Possiamo distinguere due casi.

- Supponiamo che non esistono viste che dipendono da v_i. Il contributo della vista v_i al costo $C_1 = C(Q, M, F)$ della soluzione M per $\Theta(DB, Q, F)$ è rappresentato solo dal contributo del costo di aggiornamento $f_u \cdot c_u(v_i)$ visto che nessuna query può usare v_i (altrimenti ci sarebbe una vista candidata che dipende da v_i). Il costo della soluzione $M\text{-}v_i$ è uguale a $C_2 = C(Q, M\text{-}v_i, F)$, dove $C_1 = C_2 + f_u \cdot c_u(v_i)$. Per far si che materializzare v_i comporti qualche beneficio deve essere verificata l'espressione $C_1<C_2$. Questo comporta che $f_u \cdot c_u(v_i) < 0$, che è un assurdo perché la frequenza di aggiornamento e il costo di aggiornamento sono due grandezze entrambe positive.
- Supponiamo che esista almeno una vista candidata dipendente da v_i. Identifichiamo con v_j la vista candidata direttamente dipendente di v_i. Consideriamo i due casi raffigurati nella seguente figura:

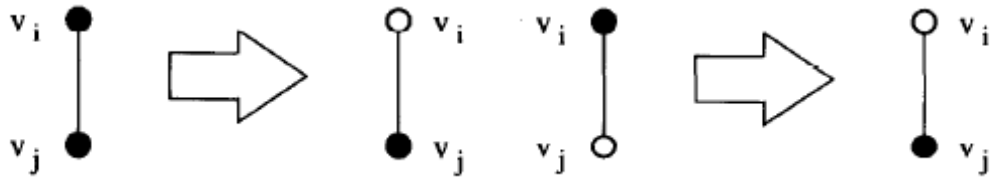


Figura 18 - Configurazioni di materializzazione

1. Entrambe le viste v_i e v_j vengono materializzate (figura a sinistra). Sia $M'=M\text{-}v_i\text{-}v_j$. Il costo di questa soluzione:

$$C_3 = \sum_{q_i \in Q} f_{q_i} \cdot c_{q_i}(M' \cup v_i \cup v_j) + f_u \cdot \sum_{v_k \in M' \cup v_j \cup v_i} c_u(v_k)$$

   Osserviamo che la vista v_i non viene usata per nessuna query $q_i \in Q$ perché tutte le query che dipendono da v_i dipendono anche da v_j, e visto che $v_i \leq v_j$ ne segue che $Cq_i(v_i)>Cq_i(v_j)$. Possiamo quindi rimuovere questo termine dalla prima parte della formula.

$$C_3 = \sum_{q_i \in Q} f_{q_i} \cdot c_{q_i}(M' \cup v_j) + f_u \cdot \sum_{v_k \in M' \cup v_j \cup v_i} c_u(v_k)$$

   Il costo della soluzione $M' \cup v_j$ è:

$$C_4 = \sum_{q_i \in Q} f_{q_i} \cdot c_{q_i}(M' \cup v_j) + f_u \cdot \sum_{v_k \in M' \cup v_j} c_u(v_k)$$

   La differenza tra C_3 e C_4 è rappresentato dal termine dal termine $f_u \cdot c_u(v_i)$. Per far si che la soluzione comporti vantaggi deve essere vera la condizione $C_3<C_4$ da cui ricaviamo che $f_u \cdot c_u(v_i) < 0$ che è un assurdo.
2. v_i viene materializzata mentre v_j no (situazione rappresentata nella figura a destra). Sia $M' = M - v_i$. Il costo di questa soluzione è:

$$C_5 = \sum_{q_i \in Q} f_{q_i} \cdot c_{q_i}(M' \cup v_i) + f_u \cdot \sum_{v_k \in M' \cup v_i} c_u(v_k)$$

   Ma tutti i termini $Cq_i(M' \cup v_j)$ definiti in C_4 sono più piccoli dei $Cq_i(M' \cup v_i)$ di C_5 perché tutte le query che dipendono da v_i dipendono anche da v_j, ma v_j è più piccola di

v_i e la funzione costo è monotona. I termini $c_u(v_j)$ sono più piccoli di $c_u(v_i)$, per la proprietà di monotonia della funzione costo di aggiornamento al variare della dimensione delle viste. Quindi è impossibile per il costo C_5 di essere minore del costo C_4.

Quindi abbiamo provato che le viste candidate sono le sole che sono rilevanti per decidere quali materializzazioni minimizzano il costo totale. Possiamo definire un sottolattice formato considerando solo le viste candidate.

Dato un lattice multidimensionale e un insieme di query Q, l'insieme delle viste candidate identifica un nuovo lattice, in cui l'operatore di join e di meet rimangono gli stessi. Questo lattice viene chiamato MDred-lattice.

Dato un insieme di query Q e una *gerarchia* di attributi di un database multidimensionale, possiamo identificare tutti gli elementi del MDred-lattice. L'identificazione può essere fatta attraverso il seguente algoritmo:

```
L := Q
lastViews := L
newViews := ∅

finché lastViews ≠ ∅
      per ogni v_i ∈ lastViews
            per ogni v_j ∈ L con v_j ≠ v_i
                  se v_i ⊕ v_j ∉ L
                        newViews := newViews ∪ (v_i ⊕ v_j)
      L := L ∪ newViews
      lastViews := newViews
      newViews := ∅
ritorna L
```

L'algoritmo estende iterativamente l'insieme L. L è inizialmente posto uguale all'insieme delle query Q. L'ultimo antenato comune di tutte le coppie di elementi viene aggiunto a L, e il processo viene iterato considerando gli ultimi antenati comuni come nuovi elementi di L e combinati con gli altri elementi. Il procedimento termina quando un punto fisso viene raggiunto.

2.7.2 ESEMPIO DI UTILIZZO DELL' ALGORITMO DI SELEZIONE DELLE VISTE

Consideriamo l'esempio proposto nel paragrafo precedente, e applichiamogli l'algoritmo illustrato qui sopra.

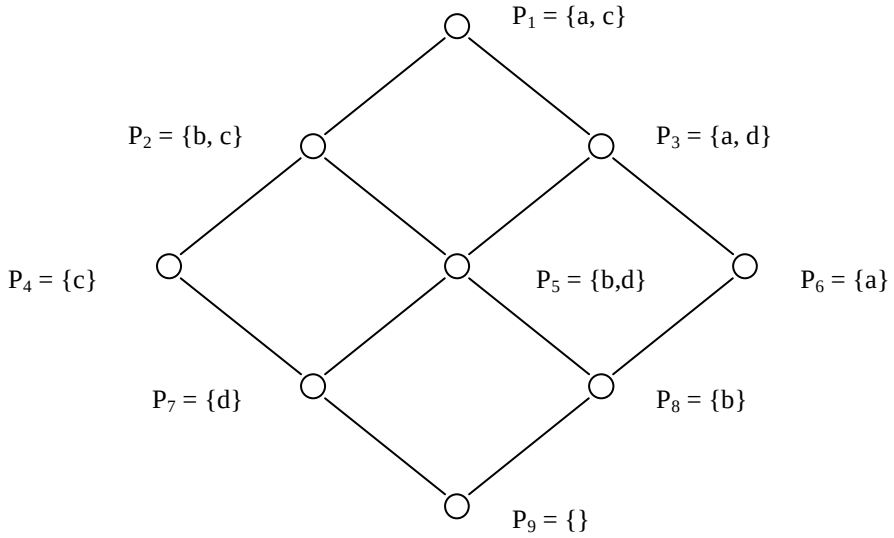


Figura 19 - Esempio di lattice

Nell'esempio riportato, ricordiamo che esistono le seguenti dipendenze funzionali: a→b, c→d.

```
L := {P6, P7, P8}          lastViews := {P6, P7, P8}    newViews := {∅}

Passo 1: vi=P6 vj=P7 vi+vj={a,d}   vi⊕vj={a,d} {a,d}=P3∉L newViews={P3}
         vi=P6 vj=P8 vi+vj={a,b}   vi⊕vj={a}   {a}=P6∈L
         vi=P7 vj=P8 vi+vj={b,d}   vi⊕vj={b,d} {b,d}=P5∉L newViews={P3,P5}

L := {P3, P5, P6, P7, P8}  lastViews := {P3, P5}       newViews := {∅}

Passo 2: vi=P3 vj=P5 vi+vj={a,b,d} vi⊕vj={a,d} {a,d}=P3∈L
         vi=P3 vj=P6 vi+vj={a,d}   vi⊕vj={a,d} {a,d}=P3∈L
         vi=P3 vj=P7 vi+vj={a,d}   vi⊕vj={a,d} {a,d}=P3∈L
         vi=P3 vj=P8 vi+vj={a,b,d} vi⊕vj={a,d} {a,d}=P3∈L
         vi=P5 vj=P6 vi+vj={a,b,d} vi⊕vj={a,d} {a,d}=P3∈L
         vi=P5 vj=P7 vi+vj={b,d}   vi⊕vj={b,d} {b,d}=P5∈L
         vi=P5 vj=P8 vi+vj={b,d}   vi⊕vj={b,d} {b,d}=P5∈L

L := {P3, P5, P6, P7, P8}  lastViews := {∅}           newViews := {∅}

Ritorna {P3, P5, P6, P7, P8}
```

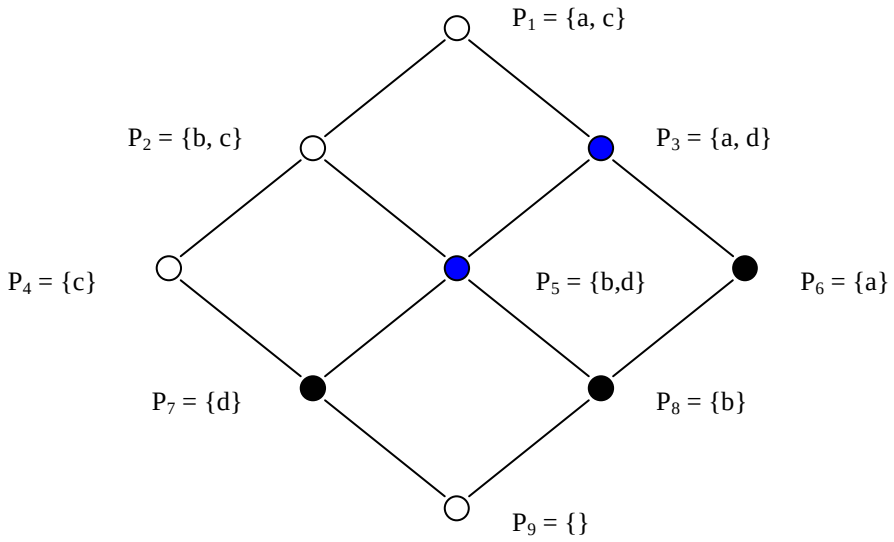


Figura 20 - Risultati dell'algoritmo di selezione delle viste da materializzare

Come si può notare i risultati ottenuti sono gli stessi presentati nel paragrafo precedente. Naturalmente, questo è un caso molto semplice, composto da due sole dimensioni e ognuna di esse è composta da una *gerarchia* di due attributi. Nella realtà, le situazioni sono molto più complicate e le viste selezionate per la materializzazione possono essere molto numerose.

2.7.3 APPLICAZIONE DI UNA TECNICA EURISTICA PER DIMINUIRE LA CARDINALITÀ DEI RISULTATI

Dopo aver costruito il MDred-lattice, una semplice tecnica euristica può essere usata per ridurre la dimensione del lattice, rimuovendo le viste che non danno il contributo aspettato alla soluzione ottimale. Questa tecnica si basa sulla stima della dimensione delle materializzazioni. Una volta stimate le dimensioni delle viste, è possibile determinare quando l'uso di un livello di aggregazione è troppo dettagliato e la sua materializzazione offre un aiuto limitato per la risposta delle query rispetto alla materializzazione di una vista di più alto livello.

Consideriamo una dimensione A con 1000 tuple. In una vista che contiene una aggregazione sulla coppia di attributi $\{A_1, A_2\}$ dove ogni attributo ha 100 valori distinti. Ci sono 10000 possibili valori della coppia di attributi ma solo 1000 tuple sono presenti nella dimensione e quindi al massimo 1000 tuple possono essere presenti nella vista. Se i dati sono uniformemente distribuiti, la dimensione stimata della vista è di 1000. Invece di materializzare questa vista, risulta più conveniente usare la vista che ha attributo di aggregazione la chiave della dimensione A. Il vantaggio consiste nella facilità di riduzione di questa vista per calcolare altre aggregazioni e nella riduzione del numero di elementi del MDred-lattice. Possiamo modificare l'algoritmo di costruzione del MDred-lattice stimando ad ogni passo la dimensione di una vista e sostituire a quelle con dimensioni elevate, una vista di più alto livello.

Definiamo una funzione di stima della dimensione size(v_A) come una funzione che applicata a una vista caratterizzata da un insieme di attributi A di un database multidimensionale, restituisce il numero delle tuple della vista.

```
Ripeti
      Stop := True
      Per ogni fd_i ∈ FD_DB
            Se (A ∩ A_i^r ≠ ∅) ∧ (ρ · size(v^Ail) < size(v^(A ∩ Air))
                  A := A - A_i^r
                  A := A ∪ A_i^l
                  Stop := False
Finché Stop = False
Ritorna A
```

L'algoritmo riportato qui sopra considera tutte le dipendenze funzionali per identificare quando gli attributi di A che appaiono nel lato destro di una dipendenza funzionale (o un sottoinsieme di essi) producono una dimensione stimata che non differisce di più di un fattore ρ della dimensione stimata degli attributi del lato sinistro. Quando questo avviene, gli attributi del lato destro vengono sostituiti dagli attributi del lato sinistro. In pratica si rimpiazza la vista v_j con la vista v_i dove $v_i \leq v_j$.

Il parametro ρ rappresenta la soglia dell'incremento della dimensione per sostituire il lato sinistro con il dato destro. È possibile usare questo valore per valutare il tradeoff tra l'accuratezza della soluzione e la riduzione della dimensione della soluzione scelta.

Numero di query	*Viste nel Mdred-lattice*	*Viste dopo la riduzione euristica*
20	795	85
25	1417	93
30	2735	114
35	3648	129
40	12378	145
45	21559	157

Tabella 14 - Riduzione del numero di viste con tecniche eurisitche

2.7.4 RISOLVIBILITÀ DELLE INTERROGAZIONI SULLE VISTE

Prima di procedere alla selezione delle viste da materializzare è necessario verificare che sia possibile risolvere un'interrogazione su una specifica vista.

Un primo esempio si presenta in caso di *misure* con funzione di aggregazione distinct count. In questo caso un'eventuale interrogazione che coinvolga questo tipo di *misura* può essere risolta solo dalla vista aggregata con lo stesso pattern di aggregazione della query o dalla fact table. In fatti, nel caso considerassi una vista con un pattern di aggregazione diverso, non sarebbe possibile utilizzarla perché non è possibile riaggregare i dati.

In generale, possiamo affermare che una vista può essere utilizzata per risolvere un'interrogazione se e solo se il pattern di aggregazione della vista è più fine del pattern di aggregazione dell'interrogazione e se le *misure* richieste sono ricavabili da quelle presenti nella vista.

Nel caso in cui le interrogazioni formulate includano più pattern di aggregazione, il pattern da considerare sarà sempre quello più fine presente.

È importante che la vista che si utilizza contenga anche i valori su cui sono effettuati dei filtri. Consideriamo il seguente esempio:

```
VENDITA[regione,tipo;anno='2000' AND quantità=50].prezzo unitario
```

Il pattern di aggregazione è {regione,tipo}, ma per risolvere l'interrogazione è necessario anche l'attributo anno su cui vengono filtrate le *misure*.

Per quello che riguarda la disponibilità delle *misure* all'interno di una vista, si sottolinea che non è strettamente necessario che tutte le *misure* richieste siano fisicamente disponibili, ma è sufficiente che siano ricavabili a partire da quelle presenti.

Occorre, in questo caso, distinguere vari casi a seconda dalle caratteristiche delle *misure* coinvolte.

- *Misure* derivate: consideriamo un'interrogazione in cui il valore dell'incasso è calcolabile avendo a disposizione una vista primaria che contiene i prezzi unitari e le quantità vendute. Purtroppo operando su dati aggregati si perde questa possibilità, poiché eseguendo la moltiplicazione dopo l'aggregazione si otterrebbe un risultato errato (la somma dei prodotti è diversa dal prodotto delle somme). Per questo motivo l'attributo derivato deve essere memorizzato esplicitamente nelle viste materializzate per poter utilizzare la vista.
- *Misure* non additive: nel caso si calcoli il valore di una *misura* non-additiva a partire da dati parzialmente aggregati è indispensabile che nella vista siano presenti anche le *misure* di supporto correlate. Per esempio, nell'esempio precedente, la vista aggregata corrispondente al pattern {regione, tipo, anno} sarà utilizzabile solo se, oltre a prezzo unitario, essa conterrà anche la *misura* di supporto count che consente di pesare diversamente l'apporto dei singoli pezzi in base alle quantità vendute (media pesata).

2.7.5 FRAMMENTAZIONE DELLE VISTE

Per migliorare le prestazioni di un sistema di data warehouse si possono, in generale, applicare tutte le tecniche utilizzate per velocizzare l'esecuzione di interrogazioni in un database. Nell'applicare queste tecniche bisogna tenere presente tre importanti differenze:

- in un data warehouse le operazioni più comuni sono quelle di lettura (si possono eseguire altre operazioni solo durante la fase di alimentazione dei dati) al contrario dei sistemi operazionali in cui, in generale, si possono avere tutti i tipi di operazioni;
- la natura dei dati all'interno di un data warehouse ha una struttura molto più semplice che tende a uniformarsi a quella di uno schema a stella, al contrario di un sistema operazionale in cui lo schema dei dati può essere anche molto complesso (pensiamo alle gerarchie, alle relazioni fra più di due tabelle, ecc… del modello relazionale);
- le operazioni di lettura all'interno di un sistema di data warehouse coinvolgono normalmente gran parte delle tuple presenti nello schema. Pensiamo, ad esempio, alle operazioni di aggregazione, che coinvolgono tutti i record. Viceversa, in un sistema operazionale, molte operazioni coinvolgono una piccola parte delle tuple presenti.

Tenendo conto di queste differenze, l'uso di indici e altre tecniche migliora sensibilmente le prestazioni di un sistema di datawarehouse. Una tecnica che sicuramente può essere applicata in questo ambito è quella nota con il nome di frammentazione o partizionamento.

In ambito relazione, con il termine partizionamento (o frammentazione) si intende la suddivisione di una tabella in più tabelle detti frammenti al fine di aumentare le prestazioni del sistema. I benefici derivanti dall'uso di questa tecnica sono particolarmente evidenti se il data warehouse è implementato su una architettura parallela con un array di dischi e un opportuno algoritmo di allocazione dei frammenti. In questo modo è possibile operare in modo parallelo prevedendo da dischi diversi i frammenti utilizzati in una stessa interrogazione.

Esistono due modalità principali per partizionare una tabella.

- Con il partizionamento orizzontale, una relazione viene suddivisa in più parti, ognuna delle quali contiene tutti gli attributi ma solo un sottoinsieme delle tuple di quella di origine.
- Con il partizionamento verticale, una relazione viene suddivisa in più parti ognuna contenenti tutte le tuple e un sottoinsieme degli attributi di quella di origine. Nel caso di partizionamento verticale gli attributi chiave della tabella di partenza dovranno essere replicati in ogni frammento al fine di permettere la ricostruzione della tabella di partenza.

2.8 TECNICHE DI ALIMENTAZIONE DEI DATAWAREHOUSE

2.8.1 PROGETTAZIONE DELL'ALIMENTAZIONE

Questa fase si occupa di progettare le procedure che definiscono come caricare i dati provenienti dai sistemi operazionali all'interno degli schemi a stella definiti in precedenza. Normalmente questa procedura può essere divisa nelle seguenti due fasi:

- dalle sorgenti operazionali al livello riconciliato. Questa prima fase è la più complessa perché prevede delle procedure che manipolano i dati per trasformali dalla loro struttura iniziale a una loro versione integrata e normalizzata. In questa fase vengono anche effettuate operazioni di pulizia dei dati;
- dal livello riconciliato al livello del *data mart*. Questa seconda fase si occupa di conformare i dati memorizzati nel livello riconciliato agli schemi a stella utilizzati. In questa fase vengono definite procedure per:
    - la denormalizzazione dei dati;
    - l'inserimento delle chiavi surrogate;
    - il calcolo dei dati derivati;

Dato che il livello riconciliato per definizione rappresenta il punto di arrivo della fase di integrazione e di pulizia dei dati, ne segue che sia lo schema riconciliato sia i dati in esso contenuti dovrebbero essere consistenti e privi di errori.

Durante la fase di analisi, si dovrebbe definire le corrispondenze tra il livello delle sorgenti e il livello degli schemi riconciliati. La progettazione dell'alimentazione dello schema riconciliato permette di rendere operativa questa corrispondenza definendo, per ogni concetto presente nello schema riconciliato, le procedure necessarie a svolgere le funzioni tipiche di un ETL.

- Estrazione: indica le operazioni che permettono di acquisire i dati dalle sorgenti.
- Trasformazione: indica le operazioni che conformano i dati delle sorgenti allo schema riconciliato.
- Caricamento: indica le operazioni necessarie a inserire i dati trasformati nel database riconciliato aggiornando eventualmente quelli già presenti.

Esiste poi un'altra operazione altrettanto importante da affiancare a queste tre ed è quella di pulizia dei dati, che serve ad eliminare eventuali incongruenze dovute ad errori e omissioni.

Come si può notare nella figura successiva l'alimentazione del livello riconciliato utilizza in genere un'area di memorizzazione temporanea (staging area) che contiene i dati grezzi a cui applicare le procedure di trasformazione e pulizia.

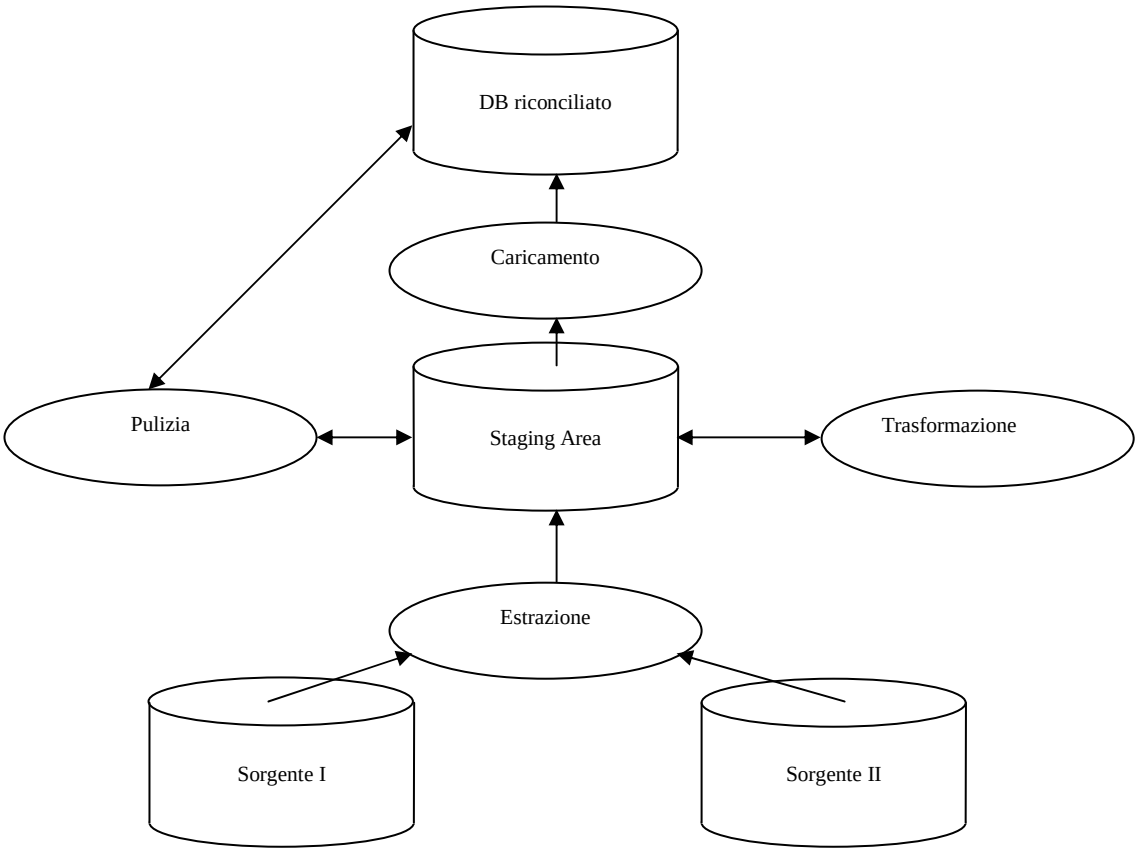


Figura 21 - Il processo di alimentazione del livello riconciliato

Nonostante l'approccio GAV (Global As View) preveda che ogni singolo concetto dello schema riconciliato sia definito come una vista sugli schemi sorgenti e che quindi possa essere alimentato singolarmente, normalmente le operazioni di alimentazione vengono in realtà svolte contemporaneamente per più concetti con lo scopo di:

- sfruttare la staging area come buffer temporaneo;
- minimizzare il costo degli accessi alle sorgenti.

Per maggiori informazioni sull'argomento vedi bibliografia [15].

2.8.2 L'ESTRAZIONE DEI DATI

Questo tipo di operazioni, indispensabili per alimentare lo schemi, vengono eseguite, oltre che all'atto dell'inizializzazione, periodicamente, in base all'intervallo di aggiornamento stabilito dal progettista per acquisire le nuove informazioni relative agli eventi verificatisi durante la vita del sistema.

Durante la definizione delle procedure di estrazione è importante individuare la natura dei dati presenti nelle sorgenti, che può essere tipizzata come segue:

- transitoria se il sistema operazionale ne mantiene solo l'immagine corrente sovrascrivendo i dati che non sono più validi;
- semi-storicizzata se il sistema mantiene un limitato numero degli stati precedenti e non è possibile determinare per quanto tempo ciascun dato verrà conservato nel sistema;
- storicizzate quando tengono traccia, per un intervallo di tempo ben definito, di tutte le modifiche intervenute sui dati.

Per esempio, sono normalmente transitori i dati di inventario come i livelli delle scorte di magazzino, che vengono soprascritti all'arrivo della nuova merce; sono invece semi-storicizzati i dati riguardanti gli stati di un sistema di cui vengono mantenute le ultime n versioni. Il momento in cui lo stato meno recente viene soprascritto del n+1-esimo evento che varia lo stato del sistema, e non può essere previsto a priori. Infine, sono normalmente storicizzati i dati in ambito bancario e assicurativo, in cui le informazioni relative ai conti correnti o alle polizze assicurative devono essere mantenute per un lasso di tempo determinato da precisi obblighi di legge.

Ovviamente i problemi nascono in presenza di sorgenti transitorie o semi-storicizzate poiché per esse dovranno essere definiti dei meccanismi in grado di estrarre i dati indipendentemente dalla brevità del loro intervallo di vita.

Sicuramente l'approccio più semplice, ma anche non adatto in caso di rilevanti quantità di dati, consiste nell'estrazione statica che prevede la scansione completa di tutti i dati presenti nelle sorgenti operazionali.

Questa soluzione risulta obbligatoria all'atto dell'inizializzazione.

Considerando che i dati modificati in uno specifico intervallo di tempo sono una piccola parte di tutti quelli contenuti all'interno del sistema operazionale, in caso di grandi moli di dati, risultano molto più efficaci le tecniche che permettono di determinare le variazioni limitando così la porzione di dati da leggere.

Gli approcci basati su questo principio sono detti di estrazione incrementale e possono inizialmente essere classificati come immediati e ritardati. I primi registrano la modifica nei dati nel momento in cui essa viene registrata dal sistema operazionale, i secondi invece posticipano tale operazione.

Devlin (Bibliografia [7]) ha definito cinque tecniche diverse:

- Estrazione assistita dall'applicazione. È una tecnica di estrazione immediata e consiste nel realizzare all'interno delle applicazioni operazionali, senza peraltro modificare il comportamento esterno, un insieme di funzioni che consentano di memorizzare stabilmente nella staging area le modifiche intervenute sui dati. L'estrazione assistita dall'applicazione fornisce una soluzione molto potente, ma è difficile da realizzare, poiché richiede di modificare applicazioni esistenti, spesso non progettate allo scopo e poco documentate, d'altro canto, soprattutto nei casi di sistemi legacy che non supportano trigger, giornali e marche temporali, può essere l'unica tecnica incrementale utilizzabile. L'estrazione assistita dall'applicazione trova giustificazione anche nei sistemi operazionali di nuova generazione in cui sia presente un substrato di primitive utilizzato uniformemente da tutte le applicazioni per accedere ai dati in modo indipendente dal DBMS. La presenza di un livello API fa si che l'estrazione dei dati avvenga in modo centralizzato e trasparente sia ai progettisti delle applicazioni, che possono essere modificate, sia ai gestori del DBMS, che può essere sostituito.
- Estrazione basata su trigger. È una tecnica di estrazione immediata, in cui la responsabilità dell'estrazione dei dati è spostata dalle applicazioni al DBMS. Un trigger è una procedura attivata direttamente dal DBMS al verificarsi di particolari eventi; ai fini dell'estrazione un trigger viene associato a ogni evento che causa cambiamenti nei dati che si vogliono monitorare. Il trigger salverà il dato modificato in un apposito file o tabella della staging area, da cui poi esso sarà prelevato per le opportune trasformazioni. Per motivi di prestazioni non è possibile adottare in modo estensivo questa tecnica, che richiede al DBMS di monitorare continuamente le transazioni che potrebbero innescare i trigger.
- Estrazione basata su log. È una tecnica di estrazione immediata in cui le operazioni di estrazione dei cambiamenti sono basate sui file di log prodotti dal DBMS. I file di log sono generati da tutti i DBMS moderni e rappresentano il principale strumento per le operazioni di back-up e ripristino dei dati; il limite principale dell'estrazione basata su log riguarda la corretta interpretazione dei file di log, il cui formato è normalmente proprietario dello specifico DBMS. Vista la complessità dell'analisi richiesta, è consigliabile utilizzare l'estrazione basata su log solo quando il modulo di estrazione è sviluppato direttamente dal produttore del DBMS.
- Estrazione basata su marche temporali. È una tecnica di estrazione ritardata e richiede che lo schema relativo ai record da estrarre preveda uno o più campi utilizzati per contrassegnare i record modificati rispetto all'ultima esecuzione del processo di estrazione dei dati. Il sistema operazionale stesso è preposto all'aggiornamento delle marche temporali, mentre il modulo per l'estrazione dei dati opera in secondo tempo direttamente sui dati operazionali producendo l'insieme delle modifiche avvenute sul sistema. Rispetto alle tecniche precedenti, l'efficacia di quella basata su marche dipende dalla struttura stessa del sistema operazionale: se i dati del sistema operazionale sono transitori o semi-storicizzati l'estrazione basata su marche non può identificare gli stati intermedi di quei record che vengono modificati più volte nell'intervallo di aggiornamento. Un ulteriore problema di questo approccio si verifica

in caso di cancellazione: il sistema dovrà accorgersi che il record non è più presente nel database.

- Comparazione di file. È una tecnica di estrazione ritardata che prevede di confrontare due versioni successive dei dati al fine di evidenziarne le differenze e richiede pertanto di mantenere la versione dei dati antecedente all'inizio delle modifiche. Come nel caso dell'estrazione basata su marche, l'efficacia dipende dalla natura dei dati: nel caso di dati temporanei o semi-storicizzati alcune modifiche avvenute tra due catture successive possono venire perse. Questa tecnica deve essere considerata solo quando nessuna delle precedenti tecniche è applicabile, poiché richiede di mantenere due versioni complete dei dati e comporta un elevato costo di esecuzione a causa delle operazioni di comparazione. Le tecniche di estrazione ritardata sono utili in presenza di sistemi operazionali non storicizzati o parzialmente storicizzati solo se gli utenti non richiedono di valutare ogni modifica dello stato delle informazioni.

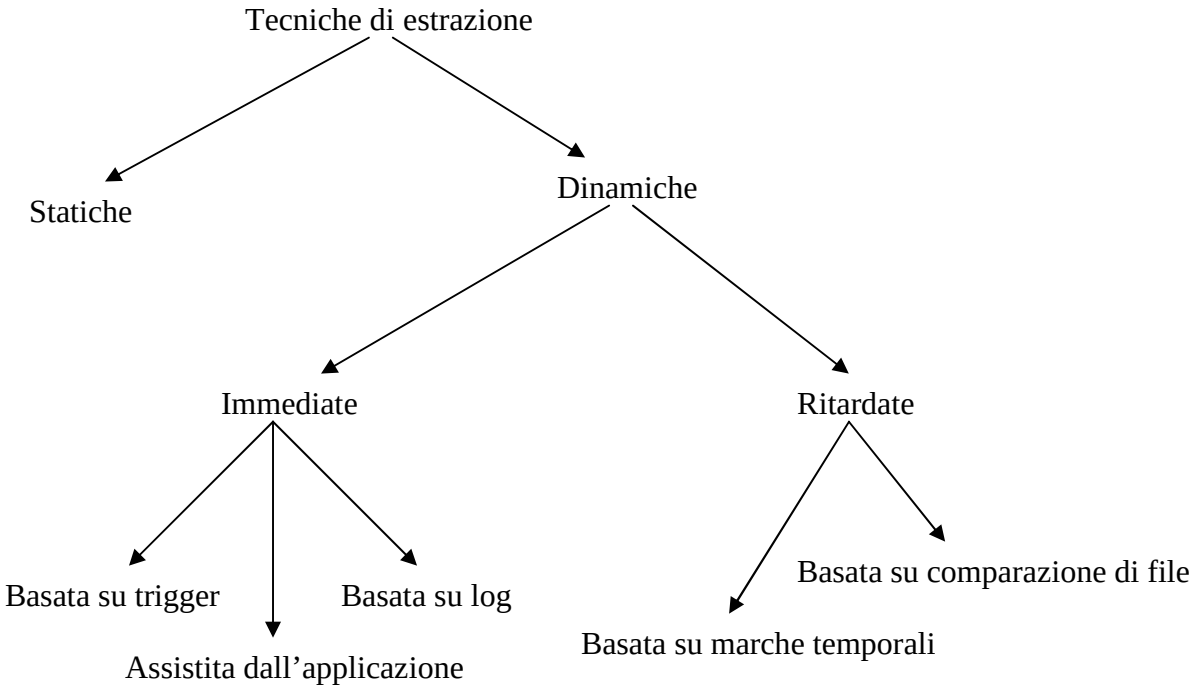


Figura 22 - Tecniche di estrazione dei dati

	Statica	*Marche temporali*	*Comparazione di file*	*Assistita dall'applicazione*	*Trigger*	*Log*
Gestione dei dati transitori o semi-storicizzati	No	Incompleta	Incompleta	Completa	Completa	Completa
Supporto per sistemi basati su file	Si	Si	Si	Si	No	Raro

	Statica	*Marche temporali*	*Comparazione di file*	*Assistita dall'applicazione*	*Trigger*	*Log*
Tecnica di realizzazione	Prodotti	Prodotti o sviluppo interno	Prodotti	Sviluppo interno	Prodotti	Prodotti
Costi di sviluppo interno	Nessuno	Medi	Nessuno	Alti	Nessuno	Nessuno
Utilizzo in sistemi legacy	Si	Difficile	Si	Difficile	Difficile	Si
Modifiche ad applicazioni	Nessuna	Probabile	Nessuna	Probabile	Nessuna	Nessuna
Dipendenza delle procedure dal DBMS	Limitata	Limitata	Limitata	Variabile	Alta	Limitata
Impatto sulle prestazioni del sistema operazionale	Nessuna	Nessuna	Nessuna	Medio	Medio	Nessuna
Complessità delle procedure di estrazione	Bassa	Bassa	Bassa	Alta	Media	Bassa

Tabella 15 - Comparazione delle caratteristiche delle diverse tecniche di estrazione incrementale

Qualunque tecnica incrementale si utilizzi, il risultato della fase di estrazione consiste nell'insieme di record della sorgente modificati, aggiunti o cancellati rispetto alla precedente esecuzione della procedura di estrazione, temporaneamente memorizzati nella staging area.

Al fine di semplificare la successiva fase di caricamento dei dati nel *data mart* conviene associare a ogni record il tipo di operazione che ne ha determinato la variazione: in questo modo il processo di caricamento potrà stabilire a priori come trattare ciascun record.

Per maggiori informazioni sull'argomento vedi bibliografia [15].

2.8.3 LA TRASFORMAZIONE DEI DATI

Durante questa fase vengono eseguite le trasformazioni necessarie a conformare i dati delle sorgenti alla struttura dello schema riconciliato.

Le categorie di operazioni più comuni sono:

- Conversione: è applicata a tutti i singoli campi che negli schemi sorgenti hanno un formato diverso da quello dello schema riconciliato. Tipiche conversioni sono quelle di tipo (ad esempio da intero a decimale, …), quelle legate al sistema di *misura* (per esempio da britannico a metrico decimale, …) o infine quelle legate al formato (per esempio da minuscolo a maiuscolo, …).
- Arricchimento: permette di combinare le informazioni presenti in uno o più campi per creare nuove informazioni o meglio rendere più facilmente fruibili quelle esistenti. Esempio classico di questo tipo di trasformazione è il calcolo di dati derivati.

- Separazione/Concatenamento: permette di ricombinare i campi letti dagli schemi sorgenti separando informazioni che erano memorizzate assieme in questi ultimi e che invece sono mantenute separate nello schema riconciliato e viceversa.

2.8.4 IL CARICAMENTO DEI DATI

In quest'ultima fase i dati estratti dalle sorgenti operazionali e opportunamente trasformati vengono caricati dalla staging area nel database riconciliato.

È ovvio che la modalità di caricamento dipende dalla tecnica utilizzata in fase di estrazione ed inoltre può variare a seconda che lo schema che si sta alimentando sia storicizzato o meno.

Se nelle fasi precedenti è stata utilizzata una tecnica di estrazione statica, i dati presenti saranno completamente cancellati e sostituiti con i nuovi.

Al contrario se nella fase precedente è stata utilizzata una tecnica incrementale, la politica di aggiornamento dipenderà dal livello di storicizzazione.

In questo caso l'inserimento di nuovi record non pone particolari problemi, ma la modifica di quelli già presenti va valutata con attenzione.

Se consideriamo dati non storicizzati, durante la procedura di caricamento è indispensabile individuare i record che hanno subito delle variazioni ed effettuarne la sostituzione.

Questa operazione, che in genere non è particolarmente complessa, può diventarlo qualora le operazioni di estrazione e trasformazioni abbiano inciso profondamente sulla struttura dei record (per esempio modificandone la chiave).

Per poter propagare la modalità incrementale anche all'alimentazione degli schemi è necessario che le procedure preposte a questo scopo siano in grado di determinare quali record del database riconciliato sono stati modificati.

Per raggiungere questo scopo ad ogni relazione dovrà essere aggiunto un flag che memorizzi quale operazione ha causato la modifica.

2.8.5 ALIMENTAZIONE DELLE DIMENSION TABLE

Le prime tabelle da alimentare nel corso della procedura sono le tabelle dimensionali. Questo deriva dalla struttura degli schemi a stella, che presuppone un vincolo di integrità referenziale tra la tabella centrale e le tabelle esterne. In caso di schemi più complicati, come quelli a fiocco di neve, il principio non cambia: le prime tabelle ad essere alimentate sono quelle più esterne.

Se a livello riconciliato gli aggiornamenti sono stati eseguiti in modo incrementale, anche il caricamento nei *data mart* potrà sfruttare questa tecnica.

In caso contrario, sarà probabilmente più efficiente ricreare l'intera dimension table. Quest'ultima soluzione può però comportare tempi elevati, anche se normalmente la grandezza delle tabelle dimensionali è minore delle tabelle dei fatti.

Le due operazioni principali da svolgere durante questa fase sono:

- identificare l'insieme dei dati da caricare;

- sostituire le chiavi presenti nelle tabelle del sistema riconciliato con quelle surrogate utilizzate negli schemi a stella.

La parte più complessa è stabilire se le tuple di una dimensional table hanno o meno effettivamente subito una modifica. Infatti, a causa della denormalizzazione che caratterizza le tuple delle tabelle dimensionali, queste ultime contengono attributi provenienti da più relazioni dello schema riconciliato e non tutti gli attributi di queste relazioni saranno inclusi nella dimensional table.

In più le procedure ETL che alimentano lo schema riconciliato non rappresentano le modifiche nel dettaglio dei singoli attributi, ma a livello di interi record.

Questo implica che se si considera una tupla modificata non è detto che i valori da caricare all'interno della dimensional table abbiano, in realtà, subito aggiornamenti.

Questa situazione può essere critica quando si considerano dimensioni dinamiche che portano alla proliferazione di record della dimensional table che contengono gli stessi valori.

L'unica soluzione esistente a questo problema è quella di comparare il contenuto dei record, che si suppongono modificati, con i corrispondenti record della dimension table.

Nella maggior parte dei casi il modo più efficiente consiste nel duplicare temporaneamente la dimension table nella staging area ed eseguire lì la comparazione o, in alternativa, se lo schema riconciliato è materializzato, la comparazione può essere fatta sul quest'ultimo a partire dai record estratti dal database operazionale.

Un'altra operazione importante da svolgere in questa fase è l'introduzione delle chiavi surrogate. Questa operazione è di per sé semplice per i nuovi record, visto che si tratta di generare un nuovo valore per la chiave che non sia già utilizzato nella dimension table.

Al contrario, invece, per rendere possibili le modifiche ai record è indispensabile mantenere un collegamento tra la vecchia e la nuova chiave delle tuple della dimension table. Questa operazione viene di solito realizzata memorizzando in modo permanente nella staging area una tabella che include la chiave dello schema riconciliato e la chiave della relazione surrogata della dimension table.

Con questa modalità, per identificare una tupla della dimension table avendo a disposizione il record del database riconciliato, è sufficiente ricercare nella tabella sopra citata la tupla che contiene la stessa chiave del record del database riconciliato e leggere la corrispondente chiave surrogata.

Al contrario del caso di inserimento di una nuova coordinata per una dimensione che determina semplicemente l'aggiunta di una tupla alla dimension table, la modifica del valore di un attributo viene realizzata in modo diverso a seconda delle caratteristiche di dinamicità dichiarate.

- Gerarchie statiche: non essendo possibile nessuna modifica dei record già presenti, l'unica operazione consentita è l'aggiunta di nuovi record. Non è allora strettamente necessario mantenere la tabella di corrispondenza tra chiavi descritta in precedenza.
- Gerarchie di tipo 1: prevedono la sostituzione del dato modificato senza tener traccia del valore precedente. In questo caso è sufficiente sovrascrivere la tupla della dimension table corrispondente alla tupla modificata nel sistema relazionale, identificabile tramite la tabella di corrispondenza tra chiavi.

- Gerarchie di tipo 2: prevedono il mantenimento dei record modificati e l'inserimento di nuovi record per le versioni aggiornate. In questo caso una modifica dà luogo all'inserimento di una nuova tupla con un nuovo valore di chiave surrogata e all'aggiornamento della tabella in corrispondenza della chiave.
- Gerarchie di tipo 3: prevedono la completa storicizzazione della dimension table. A fronte di una modifica sarà necessario modificare la vecchia tupla della dimension table inserendo la marca temporale di fine validità del record, aggiungere un nuovo record contenente i dati aggiornati e aggiornare la tabella di corrispondenza tra chiavi.

Per maggiori informazioni sull'argomento vedi bibliografia [15].

2.8.6 ALIMENTAZIONE DELLE FACT TABLE

La procedura di alimentazione della fact table è molto simile a quella delle dimension table descritta nel paragrafo precedente.

L'aggiornamento della fact table, come già ricordato in precedenza, segue sempre l'aggiornamento delle dimension table correlate per rispettare i vincoli di integrità referenziale dello schema a stella.

In modo analogo alle tabelle dimensionali, per identificare la corrispondenza tra gli attributi dello schema riconciliato e quelli della fact table è necessario fare riferimento alla documentazione del progetto logico, mentre per i valori delle chiavi surrogate da inserire nelle tuple della fact table possono invece essere determinati utilizzando le tabelle di corrispondenza predisposte durante l'alimentazione delle dimension table.

In questo tipo di operazione la maggior parte delle operazioni sono inserimenti, ma modifiche e cancellazioni di tuple già caricate nel database sono sempre possibili a causa di errori rilevati e corretti nei dati operazionali o a causa degli aggiornamenti retrospettivi.

Le modalità di gestione delle modifiche sono possibili due scenari:

- sostituzione: se un evento già registrato viene successivamente modificato, i nuovi valori che lo caratterizzano vengono sostituiti dai precedenti; nel secondo, i nuovi valori vengono invece accumulati (per esempio, come somma delle inscrizioni in una certa data);
- accumulazione: si presta all'utilizzo nel caso in cui la modifica sia stata generata dalla presenza di eventi registrati in momenti diversi ma relativi allo stesso tempo di validità.

La scelta tra una e l'altra soluzione o di entrambe dipende per prima cosa dalla possibilità di discriminare le cause che hanno determinato la modifica dell'evento, ma è anche influenzata da altri fattori che coinvolgono l'intera struttura del processo di alimentazione e richiede un'analisi dettagliata.

Per esempio, se c'è una differenza tra la grana temporale del livello riconciliato e del fatto, ossia se un evento del fatto è determinato da più eventi del livello riconciliato, la modifica non potrà concretizzarsi con una semplice sostituzione di *misure* dell'evento.

Per maggiori informazioni sull'argomento vedi bibliografia [15].

2.8.9 ALIMENTAZIONE DELLE VISTE MATERIALIZZATE

Come già accennato in precedenza gli aggiornamenti della fact table primaria devono essere propagati anche alle relative viste materializzate. Questa operazione è, dal punto di vista logico, molto semplice visto che una vista materializzata può essere alimentata tramite un'aggregazione della fact table originaria.

Ricordiamo, che nel caso di un sistema basato sul server OLAP Mondrian (vedi paragrafo 2.3.6), quest'ultimo non si occupa di mantenere aggiornate le viste secondarie in caso di variazioni della fact table e che non mette a disposizione nessuno strumento per questo scopo. Spetta al progettista, definire le opportune operazioni, ad esempio con procedure schedulate, necessarie per alimentare le viste secondarie, implementando la tecnica che ritiene più opportuna a seconda del contesto applicativo.

La soluzione più intuitiva prevede l'eliminazione e la ricostruzione ex-novo della vista e, essendo la più semplice, viene effettivamente impiegata solo quando la quantità di dati su cui operare lo consente. Al contrario, in molti casi occorre utilizzare tecniche più sofisticate per contenere i temi di aggiornamento.

- Per abbattere i costi di aggiornamento delle viste, considerare l'ordinamento di roll-up tra i pattern di aggregazione, che rende possibile alimentare una vista secondaria a partire da altre viste secondarie. Questa soluzione risulta conveniente in quanto una tabella più aggregata comporta una riduzione dei costi.
- In alcuni casi si rende necessario adottare anche per le viste una tecnica di aggiornamento incrementale. In questo caso le stesse informazioni utilizzate per l'aggiornamento incrementale della fact table primaria possono essere utilizzate anche per le viste esaminando le informazioni codificate nelle gerarchie.

Un discorso a parte meritano le viste aggregate su una *gerarchia* temporale. In questo caso risulta inevitabile apportare modifiche a tutte le tuple delle aggregazioni che sono relative al periodo corrente. Consideriamo, ad esempio, uno schema alimentato quotidianamente, un'eventuale vista aggregata per mese dovrà essere aggiornata in media 30 volte ogni mese. Per evitare questa ripetizione di aggiornamenti è possibile non includere i dati del mese corrente nella vista aggregata. In questo caso si evitano i problemi legati alle modifiche ed è sufficiente predisporre un processo mensile che effettui l'aggiornamento della vista.

Il problema dell'aggiornamento delle viste aggregate diventa particolarmente oneroso in presenza di eventi particolari che comportino la modifica di una larga parte dei dati (per esempio una ristrutturazione delle zone di vendita che implichi la ridristribuzione dei fatturati). In questi casi è consigliabile eliminare completamente le viste coinvolte e ricalcolarle ex-novo.

2.9 STRUTS

2.9.1 COS'È UN FRAMEWORK?

È molto frequente imbattersi nel termine 'framework' nella letteratura riguardante lo sviluppo di applicazioni. Molto spesso però non si ha un'idea chiara di cosa si intenda con

questo termine. Un framework è un'architettura generica che costituisce l'infrastruttura per lo sviluppo di applicazioni in una determinata area tecnologica. Detto in maniera molto semplice è un insieme di classi ed interfacce di base, che costituiscono l'infrastruttura di un'applicazione. In base a questa definizione è facile pensare erroneamente che utilizzare un framework equivalga ad usare una libreria di classi, mentre in realtà vi è una sostanziale differenza tra le due cose. Una libreria di classi, quali ad esempio le classi di base del linguaggio Java, viene utilizzata dallo sviluppatore per svolgere determinate funzionalità; in questo caso il codice che noi scriviamo invoca il codice esistente per svolgere una certa funzione, ma il controllo del flusso applicativo rimane a nostro carico.

Adottare un framework significa invece attenersi ad una specifica architettura ovvero nella pratica estendere le classi del framework e/o implementarne le interfacce. In tal caso sono i componenti del framework che hanno la responsabilità di controllare il flusso elaborativo. Il nostro codice applicativo non è direttamente invocato dall'intervento dell'utente sul sistema, ma il flusso elaborativo che passa attraverso il codice del framework: sono le classi del framework che invocano il nostro codice applicativo e non viceversa come nel caso delle librerie di classi.

Utilizzare un framework significa implicitamente adottare una specifica architettura per la propria applicazione. Anche se questo può sembrare vincolante è invece, nel caso di un framework valido, uno dei maggiori vantaggi. All'inizio di un progetto, infatti, la scelta dell'architettura è uno dei momenti fondamentali che può determinare il successo o l'insuccesso del progetto stesso. A volte è una scelta che viene trascurata o sottovalutata, principalmente per un errato approccio allo sviluppo applicativo considerato esclusivamente come una attività di scrittura di codice, ma che produce effetti disastrosi se non ponderata attentamente. Utilizzare un framework maturo e già ampiamente testato significa attenersi ad un'architettura che funziona e quindi iniziare un progetto da una base solida. Ciò porta inoltre ad un significativo risparmio di tempo e risorse in quanto lo sviluppatore non deve più preoccuparsi di realizzare componenti infrastrutturali, ma può concentrarsi esclusivamente sullo sviluppo della logica di business che poi è il valore aggiunto della applicazione che si scrive. È bene precisare che un framework non va confuso con un design-pattern. Un design-pattern è una strategia di soluzione di un problema comune, è qualcosa di concettuale che prescinde dall'implementazione tecnologica. Un framework è invece qualcosa di concreto, è un insieme di componenti che può essere usato per realizzare un'applicazione; componenti che, quando il framework è ben strutturato, sono sviluppati secondo i design-pattern più diffusi nell'ambito specifico.

In genere i vantaggi dell'utilizzo di un framework vanno ben oltre gli svantaggi, anzi si può affermare che quanto più il progetto sia di grosse dimensioni tanto più l'utilizzo di un framework è altamente consigliabile. È anche possibile sviluppare un proprio framework, anche se, a meno di casi del tutto particolari, è difficile pensare di scrivere in casa un framework che risolva problematiche diverse da quelle risolte da quelli già esistenti. Se questa fosse però la propria scelta conviene comunque studiare almeno l'architettura ed il codice, ove disponibile, dei framework più diffusi per conoscere le soluzioni adottate per i vari problemi e confrontarle con le proprie.

Di seguito vengono schematicamente riassunti alcuni dei principali vantaggi che si ottengono nell'adozione di un framework nello sviluppo di applicazioni J2EE.

- Disegno architetturale: un buon framework è fondato su un disegno architetturale valido, in quanto il suo codice è scritto in base alle best-practices della tecnologia in uso. Ciò conferisce al proprio progetto fondamenta solide dalle quali partire.
- Riduzione dei tempi di progetto: lo sviluppatore deve implementare esclusivamente la logica applicativa potendo risparmiare le energie e il tempo necessari alla scrittura di componenti infrastrutturali.
- Semplificazione dello sviluppo: un buon framework semplifica lo sviluppo applicativo perché fornisce tutta una serie di componenti che risolvono la gran parte dei compiti comuni a tutte le applicazioni web J2EE (controllo del flusso, logging, gestione messaggi di errore, custom tags per la presentation logic, internazionalizzazione, validazione dei dati, ecc...).

Esistono molti framework per lo sviluppo di applicazioni web J2EE, sia open-source che prodotti commerciali. I criteri per la scelta sono molteplici ed è bene chiarire che non esiste il framework 'ideale'. Di seguito sono elencate alcune caratteristiche che devono essere considerate valutazione.

- Maturità del progetto: è sconsigliabile adottare un framework che sia in una fase iniziale di sviluppo e che sia poco adottato nella comunità degli sviluppatori e quindi poco testato sul campo in progetti reali. Meglio indirizzarsi verso progetti già stabili e sperimentati.
- Documentazione: va sempre verificato che la documentazione sia ricca e ben fatta. Questo facilita la risoluzione dei problemi che si incontrano nella realizzazione dell'applicazione e la comprensione del suo funzionamento.
- Validità del disegno architetturale: proprio perché la scelta di un framework influisce sull'architettura applicativa è bene verificare che sia disegnato correttamente e quindi che siano adottati i design-pattern e le best-practices della tecnologia di riferimento.
- Adozione degli standard: un framework deve essere fondato sui componenti standard della tecnologia di riferimento. Nel nostro caso sulle API che costituiscono la J2EE. Quanto più un framework impone soluzioni proprietarie, l'uso di specifici tool di sviluppo o un modello troppo indirizzato ad uno specifico caso applicativo tanto più va evitato.
- Estensibilità: deve essere possibile estenderne le funzionalità per adattarlo alle proprie esigenze.

2.9.2 IL PATTERN MVC (MODEL-VIEW-CONTROLLER)

Jakarta Struts è un MVC web application framework, ovvero è un framework per lo sviluppo di applicazioni web J2EE basato sul pattern Model-View-Controller. Uno dei principali requisiti di qualsiasi applicazione web è quello di definire un modello applicativo che consenta di disaccoppiare i diversi componenti dell'applicazione in base al loro ruolo nell'architettura per ottenere vantaggi in termini di riusabilità e manuntenibilità.

Esempio tipico di questo problema è l'utilizzo nello sviluppo di una applicazione web J2EE del modello applicativo che nella letteratura è spesso indicato come "JSP Model 1". In base a questo modello l'applicazione è costruita secondo una logica "JSP centric" in base alla quale presentation , control e business logic dell'applicazione sono tutti a carico delle pagine JSP. Il web browser accede direttamente alle pagine JSP dell'applicazione che al loro interno contengono logica applicativa e logica di controllo del flusso; all'interno delle pagine JSP sono cablati i riferimenti alle viste successive in base alla logica di flusso dell'applicazione che è codificata all'interno della pagina stessa. In questo modello non esiste un controllo centralizzato del flusso ma ogni vista si fa carico della selezione delle viste ad essa collegate. Un modello di questo tipo, come suggerito dalla stessa Sun, va evitato se non per lo sviluppo di piccoli prototipi o applicazioni molto semplici e dal flusso elaborativo praticamente statico, in quanto porta a scrivere applicazioni difficilmente gestibili al crescere della complessità e non riusabili nei suoi componenti.

Quando l'applicazione cresce in complessità non è pensabile svilupparla seguendo un simile approccio. Il pattern MVC è una implementazione di quello che va sotto il nome di "Model 2"; il Model 2 introduce il concetto di controllo centralizzato dell'applicazione, implementato da una servlet di controllo che gestisce tutte le richieste e le soddisfa delegando l'elaborazione a opportune classi Java.

In questo modello i ruoli di presentation, control e business logic vengono affidati a componenti diversi e sono tra di loro disaccoppiati, con evidenti vantaggi in termini di riusabilità, manuntenibilità, estensibilità e modularità.

In un'applicazione costruita secondo il pattern MVC si possono quindi individuare tre livelli logici ben distinti che molto schematicamente svolgono i seguenti compiti:

1 *Controller*: determina il modo in cui l'applicazione risponde agli input dell'utente. Esamina le richieste dei client, estrae i parametri della richiesta e li convalida, si interfaccia con lo strato di business logic dell'applicazione. Sceglie la successiva vista da fornire all'utente al termine dell'elaborazione.
2 *Model*: contiene i dati visualizzati dalle viste; è ciò che viene elaborato e successivamente presentato all'utente.
3 *View*: visualizza all'utente i dati contenuti nel model. È la rappresentazione dello stato corrente del Model.

2.9.3 L'IMPLEMENTAZIONE DI STRUTS DEL DESIGN PATTERN MVC

Struts è un MVC web application framework, ovvero è un insieme di classi ed interfacce che costituiscono l'infrastruttura per costruire web application J2EE conformi al design pattern MVC. I componenti fondamentali di Struts sono:

- ActionServlet: è la servlet di controllo centralizzata che gestisce tutte le richieste dell'applicazione;
- struts-config.xml: è il file XML di configurazione di tutta l'applicazione. In questo file vengono definiti gli elementi dell'applicazione e le loro associazioni;

- Action: le Action sono le classi alle quali la ActionServlet delega l'elaborazione della richiesta;
- ActionMapping: contiene gli oggetti associati ad una Action nello struts-config.xml come ad esempio gli ActionForward;
- ActionForm: gli ActionForm sono classi contenitori di dati. Vengono popolati automaticamente dal framework con i dati contenuti nelle request HTTP;
- ActionForward: contengono i path ai quali la servlet di Struts inoltra il flusso elaborativo in base alla logica dell'applicazione;
- Custom-tags: Struts fornisce una serie di librerie di tag per assolvere a molti dei più comuni compiti delle pagine JSP.

La ActionServlet è la servlet di controllo di Struts. Gestisce tutte le richieste client e smista il flusso applicativo in base alla logica configurata. Si potrebbe definire come la 'spina dorsale' di un'applicazione costruita su Struts. Tutta la configurazione dell'applicazione è contenuta nello struts-config.xml. Questo file XML viene letto in fase di start-up dell'applicazione dalla ActionServlet e definisce le associazioni tra i vari elementi di Struts. Nello struts-config.xml sono, ad esempio, definite le associazioni tra i path delle richieste http e le classi Action associate alle richieste stesse.

Le associazioni tra le Action e gli ActionForm, che vengono automaticamente popolati dal framework con i dati della richiesta ad essi associata e passati in input alla Action. Contiene inoltre l'associazione tra la Action e le ActionForward , ovvero i path configurati nello struts-config.xml ai quali la ActionServlet redirigerà il flusso applicativo al termine della elaborazione della Action.

Schematicamente il flusso elaborativo nella logica di Struts può essere riassunto come segue:

1 Il client invia una richiesta HTTP.
2 La richiesta viene ricevuta dalla servlet di Struts che provvede a popolare l'ActionForm associato alla richiesta con i dati della request e l'ActionMapping con gli oggetti relativi alla Action associata alla richiesta. Tutti i dati di configurazione sono stati letti in fase di start-up dell'applicazione dal file XML struts-config.xml.
3 La ActionServlet delega l'elaborazione della richiesta alla Action associata al path della richiesta passandole in input request e response HTTP e l'ActionForm e l'ActionMapping precedentemente valorizzati.
4 La Action si interfaccia con lo strato di business che implementa la logica applicativa. Al termine dell'elaborazione restituisce alla ActionServlet un ActionForward contenente l'informazione del path della vista da fornire all'utente.
5 La ActionServlet esegue il forward alla vista specificata nell'ActionForward.

È possibile individuare alcune caratteristiche peculiari di Struts, che sono comuni anche ad altri MVC framework.

- Esiste una sola servlet di controllo centralizzata. Tutte le richieste sono mappate sulla ActionServlet nel web.xml dell'applicazione. Ciò consente di avere un unico punto di gestione del flusso applicativo e quindi permette di implementare, in modo univoco e centralizzato, funzioni quali sicurezza, logging, filtri etc.

- Le viste dell'applicazione non contengono al loro interno il riferimento al flusso dell'applicazione e non contengono logica applicativa. I livelli logici dell'applicazione sono disaccoppiati.
- Le viste sono identificate con nomi logici definiti nel file di configurazione struts-config.xml. Nel codice Java non è presente alcun riferimento a nomi di pagine JSP il che rende molto più semplice variare il flusso applicativo.
- Tutta la configurazione dell'applicazione è scritta esternamente in un file XML il che consente di modificare le associazioni tra le richieste HTTP e le classi ad essa associate in modo molto semplice.

Si possono già evidenziare alcune delle caratteristiche di un'applicazione sviluppata con Jakarta Struts e alcuni vantaggi conseguenti al suo utilizzo.

- Modularità e Ricusabilità: i diversi ruoli dell'applicazione sono affidati a diversi componenti. Ciò consente di sviluppare codice modulare e più facilmente riutilizzabile.
- Manuntenibilità: l'applicazione è costituita da livelli logici ben distinti. Una modifica in uno dei livelli non comporta modifiche negli altri. Ad esempio una modifica ad una pagina JSP non ha impatto sulla logica di controllo o sulla logica di business, cosa che avveniva nel JSP Model 1.
- Rapidità di sviluppo: a differenza di quanto avveniva utilizzando il JSP Model 1, è possibile sviluppare in parallelo le varie parti dell'applicazione, view (JSP/HTML) e logica di business (Java) sfruttando al meglio le conoscenze dei componenti del team di sviluppo. Si possono utilizzare sviluppatori meno esperti e anche con poche conoscenze di Java per la realizzazione delle view, permettendo agli sviluppatori Java più esperti di concentrarsi sulla realizzazione della business logic.

2.9.4 LA ACTIONSERVLET

La org.apache.struts.action.ActionServlet è la servlet di controllo di Struts. Come già accennato nel precedente articolo è la servlet che gestisce tutte le richieste http che provengono dai client e indirizza il flusso applicativo in base alla configurazione presente nel file XML struts-config.xml.

Com'è ovvio la ActionServlet estende la javax.servlet.http.HttpServlet; i suoi metodi doGet() e doPost() chiamano entrambi un metodo process() che esegue quindi l'elaborazione sia in caso di richieste di tipo GET che di tipo POST. Di seguito è riportato il metodo doGet(), il doPost() è identico:

```
public void doGet(HttpServletRequest request, HttpServletresponse response)
                                        throws IOException,ServletException {
    process(request,response);
}
```

La ActionServlet esegue l'elaborazione che schematicamente comprende i seguenti step:

1 I metodi doGet() e doPost() invocano il metodo process() della ActionServlet.

2 Nel metodo process() la ActionServlet ottiene l'istanza del RequestProcessor, configurato per l'applicazione nel tag <controller> dello struts.config.xml, e ne esegue il metodo process().
3 Nel metodo process() del RequestProcessor viene eseguita l'elaborazione vera e propria, ed in output al metodo viene fornito un oggetto ActionForward che consente alla ActionServlet di inoltrare l'elaborazione in base alla configurazione presente nello struts-config.xml.

La ActionServlet viene configurata, come ogni servlet, nel web.xml. Di seguito è riportato un blocco <servlet> di configurazione standard della ActionServlet nel quale è settato ad 1 il parametro <load-on-startup> in base al quale il container istanzia la ActionServlet allo start-up della web-application e ne invoca il metodo init(). Inoltre mediante il parametro config è specificata la posizione del file XML di configurazione dell'applicazione. Il parametro debug abilita il debugging dell'applicazione.

Mediante il blocco <servlet-mapping> si specifica che tutte le richieste con path terminante in .do vengono mappate sulla servlet di controllo di Struts:

```
<!-Configurazione standard della Action Servlet -->
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet
    </servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
        <param-name>debug</param-name>
        <param-value>2</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<!- Mapping della Action Servlet -->
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Con questa configurazione di base è già possibile utilizzare la ActionServlet come servlet di controllo della propria applicazione.

2.9.5 IL REQUEST PROCESSOR

La org.apache.struts.action.RequestProcessor è la classe alla quale la ActionServlet delega l'elaborazione delle richieste. Il RequestProcessor viene configurato mediante il tag <controller> dello struts-config.xml, ed è possibile utilizzarne uno proprio scrivendo una classe che estende la org.apache.struts.action.RequestProcessor e ne implementa i metodi. In particolare è di uso comune fare l'override del metodo processPreprocess() che viene eseguito dal RequestProcessor prima dell'elaborazione di ogni richiesta. Questo metodo è il punto

ottimale per inserire controlli di validità della sessione, dell'utente o simili.
Il RequestProcessor esegue i seguenti step:

1. Legge il file struts-config.xml per trovate un elemento XML <action> corrispondente al path della richiesta.
2. Una volta trovato l'elemento <action> corrispondente verifica se è presente l'attributo name che corrisponde al nome dell'ActionForm configurato per la richiesta in elaborazione. In tal caso provvede a reperire una istanza dell'ActionForm e a popolarne gli attributi con i valori presenti nella request HTTP, facendo una corrispondenza tra nome parametro e nome attributo.
3. Se nell'elemento <action> è presente l'attributo validate al valore true chiama il metodo validate() dell'ActionForm per il controllo dei dati forniti dalla request.
4. Se il controllo è ok a questo punto il RequestProcessor chiama il metodo execute() dell'Action configurata nell'elemento <action> delegandole l'elaborazione della richiesta.
5. Il metodo execute() dell'Action al termine dell'elaborazione restituisce un oggetto ActionForward che consente al RequestProcessor di inoltrare il flusso elaborativo.

Come visto quindi la ActionServlet delega l'elaborazione della richiesta al RequestProcessor che a sua volta , dopo aver popolato con i dati della request l'ActionForm configurato nell'elemento <action> corrispondente al path della richiesta, delega l'elaborazione della singola richiesta alla classe Action corrispondente. Un esempio di configurazione di una Action nel file struts-config.xml è il seguente:

```
<!-definizione del ActionForm -->
<form-beans>
    <form-bean name="startForm" type="it.prove..MenuForm" />
</form-beans>
<!-definizione del Action -->
<action-mappings>
    <action path="/start" name="startForm" scope="request"
                                type="it.prove.StartAction" validate="true">
        <forward name="ok" path="/pagina1.jsp"/>
        <forward name="ko" path="/errorpage.jsp"/>
    </action>
</action-mappings>
```

In questo esempio al path /start.do viene associato il form it.prove.StartForm e la Action it.prove.StartAction.

Ciò significa che quando il RequestProcessor riceverà una richiesta con path /start.do valorizzerà gli attributi di un oggetto della classe it.prove.StartForm con i parametri della request e dopo averne validato i valori la passerà al metodo execute() della classe it.prove.StartAction che esegue l'elaborazione prevista.

2.9.6 LA CLASSE ACTION

Dopo aver esaminato la ActionServlet ed il RequestProcessor finalmente arriviamo a parlare della classe org.apache.struts.action.Action. La classe Action è l'elemento fondamentale

del controller di Struts in quanto per ogni funzione realizzata con Struts bisogna creare una propria classe che la estende e ne implementa il metodo execute() che è fatto come segue:

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response) throws Exception{
    //codice di esempio acquisizione form
    MyForm myForm = (MyForm)form;
    //acquisizione parametri dal form
    String param1 = myForm.getParam1();
    //business logic
    ...
    //inoltro dell'elaborazione
    return mapping.findForward("ok");
}
```

Il metodo execute() riceve in input request e response HTTP, un'istanza dell'oggetto ActionForm prima descritto, e un oggetto ActionMapping che contiene le informazioni configurate nell'elemento <action> tra le quali i forward, ovvero i percorsi a cui inoltrare in uscita l'elaborazione. Restituisce un oggetto ActionForward che contiene il path di inoltro dell'elaborazione.

È nel metodo execute() della propria Action che lo sviluppatore inserisce il proprio codice di elaborazione della richiesta per la funzione specifica.

Le Action costituiscono quindi il 'ponte' applicativo tra lo strato di Controller e di Model di un applicazione scritta con Struts ed hanno un ruolo fondamentale perché sono le classi che lo sviluppatore scrive continuamente nello sviluppo di una applicazione Struts.

2.9.7 LA CLASSE ACTIONFORM

In ogni applicazione web la view ha due compiti fondamentali: presentare all'utente i dati frutto dell'elaborazione eseguita e consentire all'utente l'immissione di dati da elaborare.

Normalmente in una applicazione J2EE tradizionale i dati da visualizzare sono contenuti negli attributi di un JavaBean memorizzato nell'appropriato scope al quale la pagina fa riferimento.

L'immissione di dati è realizzata mediante un form HTML contenuto nella pagina JSP e i vari input type che consentono l'invio nella request di dati che saranno reperiti dai componenti del controller e passati allo strato di business logic.

Il reperimento dei dati dalla request, la loro validazione e il popolamento con essi degli oggetti del model è a carico dello sviluppatore che dovrà scrivere codice per reperire i dati dalla request , istanziare un oggetto di una classe opportuna per memorizzarli , validarli e fornirli allo strato di business-logic.

Gli ActionForm di Struts consentono di automatizzare in parte questo compito che è uno dei più frequenti e ripetitivi in un'applicazione web J2EE.

Un ActionForm è una classe che estende la org.apache.struts.actions.ActionForm e che viene utilizzata per acquisire i dati da un form HTML e fornirli ad una classe Action.

In pratica il controller di Struts provvede a popolare in automatico gli attributi di un ActionForm associato ad una determinata Action con i dati inviati nella request, associandoli in

base alla corrispondenza nome-parametro nome-attributo, e a passare l'istanza dell'ActionForm così valorizzata al metodo execute() della Action stessa.

Gli ActionForm costituiscono quindi una sorta di buffer nel quale vengono posti i dati digitati in un form HTML, che possono così essere ripresentati facilmente all'utente in caso di errori nella validazione. Allo stesso tempo costituisce per così dire un 'firewall' per l'applicazione in quanto facilita il controllo dei dati prima che questi vengano passati allo strato di logica.

Gli ActionForm sono del tutto equivalenti a dei JavaBean e possono essere quindi utilizzati per contenere i dati restituiti dallo strato di business logic e da presentare all'utente oltre che a essere usati come classi corrispondenti ad un form HTML come il loro nome suggerisce.

Di seguito è riportato il codice di esempio di un ActionForm corrispondente ad un classico form di immissione di username e password:

```
public class LoginForm extends ActionForm {
    private String password = null;
    private String username = null;
    public String getPassword() {
       return this.password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getUsername() {
        return this.username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
}
```

Come si vede la struttura è esattamente quella di un JavaBean. In più hanno due metodi particolari: reset() e validate() che ne caratterizzano il comportamento.

Il metodo reset() viene chiamato dal controller dopo che questo ha creato o reperito dallo scope opportuno l'istanza dell'ActionForm. Può quindi essere usato per inizializzare gli attributi del form ad un valore stabilito.

Il metodo validate() viene chiamato dal controller dopo la valorizzazione degli attributi dell'ActionForm qualora nello struts-config.xml sia stato valorizzato a true l'attributo validate del tag <action> nel quale si fa riferimento all'ActionForm in questione.

Nel metodo validate() va inserito il codice per la validazione formale dei dati del form. Ciò garantisce di avere un punto standard nel codice nel quale questa validazione viene effettuata e che i dati che arrivano alla Action siano già stati formalmente validati.

2.9.8 GLI ACTIONERRORS

Il metodo validate() di un ActionForm è il punto nel quale viene inserito il codice di validazione formale dei dati immessi dall'utente in un form HTML.

La signature del metodo è la seguente:

```
public ActionErrors validate(ActionMappings mapping,
                                                HttpServletRequest request)
```

Il tipo di ritorno del metodo è un oggetto della classe ActionErrors che è un contenitore di oggetti della classe org.apache.struts.action.ActionError (o rispettivamente ActionMessages e ActionMessage nelle nuove versioni).

Ogni oggetto della classe ActionError rappresenta un errore verificatosi nella validazione dei dati. Qualora durante la validazione si verifichino degli errori, per ciascuno di essi viene creata una istanza di un oggetto ActionError e aggiunta all'oggetto ActionErrors restituito dal metodo. Se il controller verifica che l'oggetto ActionErrors in uscita al metodo validate() non è nullo, non trasferisce il controllo al metodo execute() della classe Action associata alla richiesta in elaborazione ma bensì alla pagina JSP il cui path è configurato nell'attributo input del tag <action> corrispondente.

Con un opportuno custom tag (<html:errors>) posto nella pagina stessa sarà possibile visualizzare i messaggi di errore associati agli errori verificatisi senza scrittura di codice aggiuntivo.

Il messaggio d'errore viene reperito automaticamente dal framework dal resouce bundle dell'applicazione; la chiave del messaggio è fornita nel costruttore dell'oggetto ActionError associato all'errore in questione. Di seguito è riportato l'esempio di un metodo che esegue la validazione dell'username e della password immessi nel form HTML associato all'ActionForm visto in precedenza:

```
public ActionErrors validate(ActionMapping mapping,
                                                HttpServletRequest request) {
    ActionErrors errors = new ActionErrors();
    if ((username == null) || (username.length() < 1))
           errors.add ("username",
                       new ActionError("errore.username.obbligatorio"));
    if ((password == null) || (password.length() < 1))
            errors.add("password",
                       new ActionError("errore.password.obbligatoria"));
  return errors;
}
```

Le label errore.username.obbligatorio e errore.password.obbligatorio sono le chiavi alle quali sono associati i messaggi di errore nel resource bundle dell'applicazione.

Con il metodo validate, le classi ActionErrors ed ActionError ed il tag <html:errors> il framework fornisce quindi un automatismo standard per la gestione della validazione dei dati immessi nella view dell'applicazione e per la visualizzazione dei messaggi di errore.

2.9.9 I COMPONENTI DI STRUTS PER LA GESTIONE DELL'INTERNAZIONALIZZAZIONE

Struts fornisce supporto all'internazionalizzazione essenzialmente per ciò che riguarda il reperimento di testo e immagini localizzate. Per gli altri aspetti, quali formattazione di date, importi etc. bisogna fare ricorso alle classi Java standard.

Struts gestisce l'internazionalizzazione fornendo gli strumenti per reperire risorse localizzate da opportuni resource bundle in base al locale corrente.

In una web-application, e quindi anche in quelle costruite con Struts, è possibile reperire l'informazione relativa al locale dell'utente mediante il metodo public java.util.Locale getLocale() dell'oggetto HttpServletRequest. Infatti l'informazione del locale utilizzato dall'utente è inviata al container in ogni request; Struts come default memorizza questa informazione nella sessione, ma è possibile variare questo comportamento impostando il valore dell'attributo locale del tag <controller…/> nello struts-config.xml. Il valore di default è false.

Con l'informazione del locale presente in sessione l'applicazione è quindi in grado di reperire dal resource bundle appropriato la risorsa localizzata.

Per la gestione dei resource bundle in Struts viene usata la classe org.apache.struts.util.MessageResources che segue la stessa logica della java.util.ResourceBundle arricchendola con alcune funzioni di utilità.

Anche la classe org.apache.struts.util.MessageResources è una classe astratta, e la sua concreta implementazione è fornita dalla classe org.apache.struts.util.PropertyMessageResources che consente di leggere stringhe localizzate alle quali è associata una chiave reperendole da file di properties, esattamente come fa la java.util.PropertyResourceBundle.

Il primo elemento da definire quindi per localizzare un'applicazione Struts è proprio il resource bundle ovvero il file di properties contenente la lista in formato nome/valore di tutte le label dell'applicazione. Esisterà un file di properties per la lingua di default della propria applicazione chiamato ad esempio ApplicationResources.properties che avrà una serie d'elementi del tipo:

```
...
button.aggiorna=Aggiorna
button.conferma=Conferma
button.elimina=Elimina
button.inserisci=Inserisci
button.salva=Salva
...
```

Dovranno poi essere definiti tanti altri file di properties per tutte le combinazioni lingua/regione per le quali si vuole che l'applicazione sia predisposta. Ad esempio

```
ApplicationResource_en_En.properties per inglese/regnoUnito
ApplicationResource_en_US.properties per inglese/Stati Uniti
ApplicationResources_fr_FR.properties per francese/Francia
```

Nello struts-config.xml va indicato qual è il resource bundle utilizzato dall'applicazione con il tag:

```
<message-resources parameter="it.prova.ApplicationResources"/>
```

indicando il nome radice della famiglia di resource bundle che si riferiscono alle stesse risorse localizzate.

I file di properties così definiti vanno quindi installati nella cartella /WEB-INF/classes dell'applicazione rispettando la struttura di package dichiarata nella definizione precedente.

Per reperire le stringhe localizzate nelle pagine JSP si utilizza invece un custom-tag della libreria struts-bean, precisamente il tag <bean:message>. L'utilizzo è banale , basta fornire come attributo la chiave corrispondente alla label che si vuole acquisire come nell'esempio seguente:

```
<bean:message key="label.username" />
```

In questo modo è molto semplice scrivere pagine JSP nelle quali non sono presenti label direttamente scritte nel codice, e quindi utilizzare lo stesso sorgente della pagina per visualizzare informazioni in lingue differenti.

2.9.10 IL VALIDATOR

Il Validator è un framework che fornisce gli strumenti per effettuare in modo automatico e dichiarativo i controlli formali sui campi digitati in un form HTML.

Usando il Validator non è necessario scrivere alcun codice di validazione nel metodo validate() degli ActionForm, ma è il Validator stesso che fornisce questa funzione purché i form bean dell'applicazione estendano uno degli ActionForm del Validator stesso.

Il Validator è costituito da un insieme di classi predisposte per eseguire tutti i più comuni controlli di validazione in genere usati nelle applicazioni, ma esiste anche la possibilità di creare routine di validazione non fornite dal Validator. Il Validator inoltre supporta sia la validazione server-side che quella client-side mediante opportune funzioni JavaScript, cosa non fornita dal meccanismo standard degli ActionForm di Struts.

La configurazione delle routine di validazione da applicare ad un campo di un form è fatta mediante un file di configurazione XML, quindi esternamente all'applicazione ed è facilmente modificabile al mutare delle esigenze applicative. Nel file validator-rules.xml vengono dichiarate tutte le routine di validazione disponibili, i loro nomi logici e il codice JavScript corrispondente a ciascuna routine di validazione per l'esecuzione dei controlli client-side.

Nel file validation.xml si specifica come queste routine vengano applicate ai vari campi di input dei form dell'applicazione, ai quali si fa riferimento mediante i nomi dei form beans dichiarati nello struts-config.xml.

Utilizzare il Validator con Struts significa eseguire i seguenti step:

1 Abilitare il Validator plug-in.
2 Configurare i due file XML appena citati, validator-rules.xml e validation.xml.
3 Creare form bean che estendano gli ActionForm del Validator.

Il Validator viene configurato come un plug-in di Struts. Per abilitare il Validator bisogna aggiungere nello struts-config.xml le seguenti righe:

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
<set-property property="pathnames"
            value="/WEB-INF/validator-rules.xml, /WEB-INF/validation.xml"/>
</plug-in>
```

In questo modo Struts all'avvio dell'applicazione carica e inizializza il Validator; si può notare come vengano definiti i percorsi dei due file XML di configurazione del Validator.

Il validator-rules.xml dichiara le routine di validazione messe a disposizione dal Validator. Esistono una serie di routine fornite dalla versione standard del Validator che coprono la gran

parte delle comuni esigenze per ciò che riguarda i controlli formali da eseguire sui campi di input di un form. Il file in genere non va quindi modificato a meno che non si vogliano definire delle proprie routine.

Per utilizzare il Validator i form bean dell'applicazione non devono estendere la classe ActionForm standard di Struts ma la classe org.apache.struts.validator.ValidatorForm che fornisce l'implementazione del metodo validate(). In questo caso non è più necessario scrivere il codice di validazione perché è il Validator che lo fa per noi. La configurazione dei form bean all'interno dello struts-config.xml è identica a quella fatta in precedenza utilizzando la classe ActionForm standard di Struts.

2.9.11 ESTENSIONI DEI CONCETTI DI STRUTS

Struts fornisce alcune Action base che arricchiscono il framework di alcune funzionalità rispetto alla Action base standard e che sono utili a diversi scopi . Di seguito forniamo una descrizione di quelle più utilizzate nella pratica.

Abbiamo già visto precedentemente che nel funzionamento standard di Struts, il framework esegue il metodo execute() della Action che corrisponde al URL della richiesta effettuata dal client. Il metodo execute() costituisce quindi il punto di ingresso nel framework a fronte di una richiesta dal client. Questo funzionamento si adatta poco alle situazioni nelle quali è necessario eseguire una serie di elaborazioni tra loro logicamente collegate. Tipico è l'esempio di operazioni di inserimento, cancellazione , lettura e aggiornamento su una stessa tabella di un database. Sarebbe abbastanza poco efficiente dover definire una Action per ciascuna singola operazione , in quanto questa tecnica porterebbe ad un proliferare di Action nell'applicazione e quindi ad una difficile gestione della stessa. Struts ci viene incontro fornendo org.apache.struts.actions.DispatchAction.

La DispatchAction è assolutamente analoga ad una Acion base ma fornisce la possibilità di invocare diversi metodi della stessa purché il client specifichi il metodo da chiamare. In pratica è come una Action che non ha un solo metodo execute() ma ne ha n con nomi diversi. Ognuno di questi metodi deve avere la stessa signature del metodo execute().

Ad esempio una DispatchAction potrebbe avere i seguenti metodi:

```
public ActionForward inserisci(ActionMapping mapping,
    ActionForm form, HttpServletRequest req, HttpServletResponse res)
                                        throws IOException,ServletException;
public ActionForward aggiorna(ActionMapping mapping, ActionForm form,
                      HttpServletRequest req, HttpServletResponse res)
                                        throws IOException,ServletException;
public ActionForward cancella(ActionMapping mapping, ActionForm form,
                       HttpServletRequest req, HttpServletResponse res)
                                        throws IOException,ServletException;
public ActionForward leggi(ActionMapping mapping, ActionForm form,
                        HttpServletRequest req, HttpServletResponse res)
                                        throws IOException,ServletException;
```

Affinché il framework sappia a quale metodo delegare l'elaborazione della richiesta, il client deve fornire nella request un parametro contenente il nome del metodo corrispondente.

Questo parametro va ovviamente specificato nella definizione della Action nello struts-config.xml nel seguente modo:

```
<action
path="/gestioneTabella"
type="it.esempio.GestioneTabellaAction"
name="gestioneTabellaForm"
scope="request"
input="/tabella.jsp"
parameter="metodo"/>
```

Se quindi da una pagina JSP si vuole invocare il metodo inserisci() della Action GestioneTabellaAction sarà sufficiente specificare:

```
http://servername/context-root/gestioneTabella?metodo=inserisci
```

Chiaramente il nome del metodo può essere specificato in diversi modi, come un hidden contenuto in un form HTML oppure può essere impostato da una funzione JavaScript prima di eseguire la submit() del form.

L'importante è che è possibile raggruppare logicamente azioni tra di loro correlate in un'unica Action il che porta ad una migliore strutturazione dell'applicazione ed evita duplicazioni inutili di codice. Nella pratica comune la DispatchAction è effettivamente molto utile.

La org.apache.struts.actions.LookupDiapatchAction è utile quando la scelta del metodo da eseguire in una Action di tipo 'dispatch' è effettuata mediante i button di un form ma si ha la necessità di avere le label dei button localizzate e non ci si vuole affidare a codice JavaScript per la selezione del metodo da attivare.

La LookupDiapatchAction è, quindi, del tutto analoga alla DispatchAction per quel che riguarda la sua struttura, quello che cambia è il modo con il quale viene selezionato il metodo da mandare in esecuzione.

In questo caso le label dei button vengono associate alle loro key contenute nel resource bundle dell'applicazione, e queste key, che molto probabilmente non sono nomi di metodo validi, vengono mappate dallo sviluppatore ai metodi della Action mediante la definizione di un metodo così fatto:

```
protected Map getKeyMethodMap(ActionMapping mapping, ActionForm form,
                                                HttpServletRequest request) {
  Map map = new HashMap();
  map.put("bottone.leggi", "leggi");
  map.put("bottone.inserisci", "inserisci");
  map.put("bottone.cacnella", "cancella");
  map.put("bottone.modifica", "modifica");
  return map;
}
```

Nel metodo viene definita una HashMap nella quale ad ogni key è associato il nome di un metodo della Action. Questo metodo è usato dal framework per determinare la corrispondenza tra la label localizzata del bottone cliccato ed il metodo della LookupDispatchAction da eseguire.

In base a questo codice la definizione dei bottoni nella pagina JSP sarà del seguente tipo:

```
<html:form action="/gestioneTabella">
    <html:submit property="method">
        <bean:message key=" bottone.leggi ">
    </html:submit>
    <html:submit property="method">
        <bean:message key=" bottone.inserisci ">
    </html:submit>
    <html:submit property="method">
        <bean:message key=" bottone.cacnella ">
    </html:submit>
    <html:submit property="method">
        <bean:message key=" bottone.modifica ">
    </html:submit>
</html:form>
```

In questo modo è possibile avere una Action di tipo 'dispatch' i cui metodi sono attivabili da button di un form di una applicazione localizzata semplicemente scrivendo un metodo getKeyMethodMap() come descritto.

La org.apache.scaffold.http.BaseAction è una Action fornita nei package opzionali denominati Scaffold. È una Action che fornisce un metodo execute() base fatto come segue:

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
  HttpServletRequest req, HttpServletResponse res) throws Exception {
  // Controlla le pre-condizioni all'elaborazione
  preProcess(mapping,form,request,response);
  // ci sono errori va in failure
  if (isErrors(request)) {
      return findFailure(mapping,form,request,response);
  }
  // Esecuzione della logica
  try {
      executeLogic(mapping,form,request,response);
  }catch (Exception e) {
      // Gestione dell'eccezione
      setException(request,e);
      catchException(mapping,form,request,response);
  }
  finally {
      // Elaborazioni finali
      postProcess(mapping,form,request,response);
  }
  // Se ci sono errori va in failure
  if (isErrors(request)) {
      return findFailure(mapping,form,request,response);
  }
  // Otherwise, check for messages and succeed (only 1_0)
  if ((isStruts_1_0()) && (isMessages(request))) {
      saveErrors(request,getMessages(request,false));
  }
}
```

La BaseAction può essere usata come classe base di tutte le Action dell'applicazione allo scopo di standardizzare la scrittura dei vari metodi execute(). In pratica in questo caso lo sviluppatore dovrà implementare nella propria Action il metodo executeLogic() ed

eventualmente uno tra i metodi preProcess() o postProcess(). Il metodo execute() della ActionBase rispecchia le operazioni effettuate in genere da un comune metodo execute() di una Action. Questo approccio può essere comunque usato come spunto per costruirsi una propria Action base se non si vuole utilizzare la BaseAction dei package Scaffold.

La ForwardAction consente di inoltrare il flusso dell'applicazione ad un'altra risorsa individuata mediante un URI valido, risorsa che può essere una pagina JSP una servlet o altro.

La ForwardAction quindi non fa altro che creare un RequestDispatcher ed effettuare il forward alla risorsa individuata dall'attributo parameter specificato nella definizione della Action nello struts-config.xml.

```
<action path="/vaiAllaPagina1" type="org.apache.struts.actions.ForwardAction"
name="pagina1Form" scope="request" input="/pagina0.jsp" parameter="/pagina1.jsp "/>
```

La ForwardAction è molto utile quando nell'applicazione si hanno pagine JSP che non richiedono alcuna elaborazione a monte prima di essere visualizzate. Affinché si eviti di effettuare un inoltro alla pagina in questione direttamente da un'altra pagina JSP dell'applicazione si può usare la ForwardAction. In questo modo si resta aderenti al modello di Struts che prevede un controllo centralizzato di tutte le richieste e si predispone l'applicazione a modifiche future. Se un domani, infatti, la pagina in questione richiederà qualche elaborazione basterà sostituire una Action opportuna al mapping corrispondente alla ForwardAction.

Un altro utilizzo della ForwardAction è come elemento d'integrazione con altre applicazioni data la sua caratteristica di effettuare un inoltro ad un URI generico.

La IncludeAction è la corrispettiva della ForwardAction per l'operazione di include della risposta generata da un'altra risorsa. Consente di aggiungere alla propria, l'elaborazione effettuata da un'altra risorsa quale ad esempio una servlet. Non è molto utilizzata, ma è in ogni caso fornita da Struts qualora possa servire.

Per un'analisi più approfondita del framework Struts vedi bibliografia [19].

3. PROTOTIPO REALIZZATO

3.1 SCOPO DELL' APPLICAZIONE

Il prototipo realizzato durante il periodo di stage si propone come strumento di data warehouse per servizi di reportistica per piccole e medie quantità di dati. Si tratta di un'applicazione Web scritta in Java che utilizza il framework Jakarta Struts (vedi capitolo 2.9). L'applicazione si basa su un OLAP server open source chiamato Mondrian (vedi capitolo 2.3) e su un componente grafico per la navigazione di cubi di nome JPivot (vedi paragrafo 2.6.1).

L'applicazione realizzata implementa le seguenti funzionalità:

- definire schemi che descrivono più cubi con diverse strutture dati memorizzati all'interno di diversi DBMS;
- permettere all'amministratore del sistema di definire dei report, delle tabelle e dei grafici su ognuno degli schemi definiti in precedenza;
- organizzare gli elementi creati nel punto precedente con una struttura Categoria / Report per permettere una maggiore navigabilità all'utente;
- fornire un "Modeller" all'amministratore per costruire interrogazioni sui cubi con un meccanismo di Query-by-example, cioè di permettere, a partire da un'interrogazione predefinita, all'amministratore modificando quest'ultima per creare nuovi report e salvarli nel sistema;
- permettere all'amministratore del sistema di creare cruscotti che contengono più oggetti tra quelli definiti in precedenza in un'unica pagina. La disposizione degli oggetti all'interno della pagina viene specificata da template personalizzabili basati su tabelle HTML;
- permettere agli utenti di visualizzare gli elementi creati dall'amministratore del sistema e che quest'ultimo ha reso visibili. La navigazione degli utenti non si limita alla semplice visualizzazione, ma all'interazione con gli elementi con operazioni di "drill-down", "roll-up" o "drill-member".
- permettere di definire procedure notturne che si occupino dell'aggiornamento dei dati contenuti all'interno degli schemi a stella;
- permettere all'amministratore di sistema di definire diverse procedure composte di un numero qualsiasi di regole per analizzare e estrarre informazioni utili dai file di log di accesso generati da Tomcat. Per una descrizione più dettagliata di questa funzione si veda il paragrafo 3.3.6 più avanti.

La scelta di sviluppare l'applicazione basandosi sui due componenti open-source Mondrian e JPivot è stata fatta considerando i benefici che ciascuno dei due componenti avrebbe apportato all'interno dell'applicazione.

In particolare:

- Mondrian, almeno dalla versione 2.0.5 in poi, supporta un gran numero di funzioni MDX che permettono di eseguire calcoli molto complessi che non si limitano solo all'aggregazione di dati;
- Mondrian permette di configurare diverse sorgenti dati, in particolare attraverso una connessione JDBC può interrogare la maggior parte dei DBMS;
- Lo schema dei dati non è vincolato al solo modello a stella, ma è possibile definire diverse strutture dati attraverso un catalogo scritto in XML;
- Mondrian si occupa di aspetti critici come il caching dei dati richiesti al fine di migliorare le prestazioni del sistema;
- è possibile definire funzioni definite dall'utente, scritte in Java. Una volta che sono state registrate in Mondrian è possibile utilizzarle all'interno di interrogazioni MDX;
- JPivot permette agli utenti una navigazione semplice, almeno per quello che riguarda le funzioni di "drill-down" e di "roll-up" (che sono le più usate) dei cubi;
- JPivot si occupa di richiedere al server OLAP solo i dati strettamente necessari per la visualizzazione e per l'utente, riscrivendo ad ogni azione dell'utente la query da eseguire e interrogando il server OLAP solo per i dati non ancora visualizzati in quella sessione; mette a disposizione operazioni come il "drill-member" e il "drill-position" che permettono all'utente di focalizzarsi solo sui dati che ritiene importanti e al server OLAP di perdere tempo a fornire altri dati non necessari.

Altre caratteristiche del prototipo realizzato sono:

- possibilità di esportare in formato pdf e excel i report ottenuti;
- possibilità di definire nuovi template basati su una tabella HTML di dimensioni massime di 5 x 5 e di definire per ogni posizione l'occupazione in termini di righe e colonne;
- possibilità di definire nuovi schemi e le relative connessioni attraverso un'apposita sezione del prototipo specificando tutti i parametri necessari per una connessione JDBC, il file di catalogo da utilizzare e la query MDX di default da utilizzare per il "Modeller";
- possibilità di inserire una descrizione per ogni elemento del programma;
- possibilità di visualizzare una pagina in cui vengono raccolti i risultati delle ultime procedure di alimentazione;
- possibilità di resettare l'applicazione, invalidando tutti i dati contenuti nella cache di Mondrian (quelli di tutti gli utenti, non solo i propri) per rendere disponibili modifiche ai cataloghi o ai dati contenuti negli schemi senza riavviare Tomcat;
- la funzione di "OLAP Navigator" per permettere agli utenti di modificare le dimensioni coinvolte in un report in modo grafico;
- possibilità di effettuare operazioni di pivoting sui report per ruotare le dimensioni presenti.

3.2 DESCRIZIONE DEL PROTOTIPO REALIZZATO

In questo paragrafo procederemo analizzando da più vicino il prototipo realizzato durante il periodo di stage. Ognuno dei paragrafi successivi descrive una differente sezione dell'applicazione realizzata. Le sezioni descritte sono:

- gestione degli schemi e delle sorgenti dati (Paragrafo 3.2.1);
- gestione delle categorie e dei gruppi di report (Paragrafo 3.2.2);
- gestione delle liste di elementi (Paragrafo 3.2.3);
- gestione degli elementi di tipo "Report" (Paragrafo 3.2.4);
- gestione degli elementi di tipo "Tabella" (Paragrafo 3.2.5);
- gestione degli elementi di tipo "Grafico" (Paragrafo 3.2.6);
- gestione degli elementi di tipo "Pagina" (Paragrafo 3.2.7);
- gestione dei template (Paragrafo 3.2.8);
- gestione delle procedure schedulate eseguite (Paragrafo 3.2.9);
- gestione delle regole di analisi dei log di accesso (Paragrafo 3.2.10);
- funzioni messe a disposizione per la navigazione dei cubi (Paragrafo 3.2.11).

Una nota importate, riguarda la presenza o meno di un determinato ruolo da parte dell'utente al momento dell'autenticazione. Se l'utente che entra nel sistema, ha associato un ruolo di nome "bart-admin", avrà diritti d'amministratore, altrimenti sarà considerato come un utente non amministratore. Questo tipo di discriminazione è una prima distinzione tra gli utenti del sistema. Come vedremo in seguito, i diritti d'amministrazione abiliteranno sezioni e funzioni dell'applicazione non disponibili per gli utenti senza questo diritto.

Per la parte di visualizzazione, l'applicazione è predisposta a una verifica di livello superiore sull'identità degli utenti. Per una descrizione di questa funzione vedi il capitolo 4.

Ogni schermata dell'applicazione che permette l'inserimento di dati, effettua una validazione di questi ultimi utilizzando le funzionalità messe a disposizione dal framework Struts (vedi Paragrafo 2.9.8).

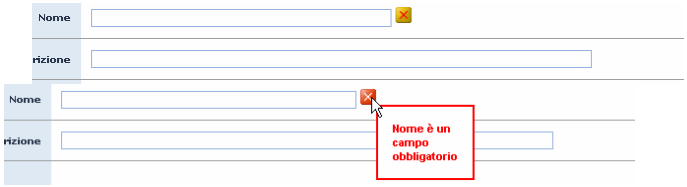


Figura 23 - Esempio di validazione

Di seguito riportiamo lo schema ER del database sottostante l'applicazione.

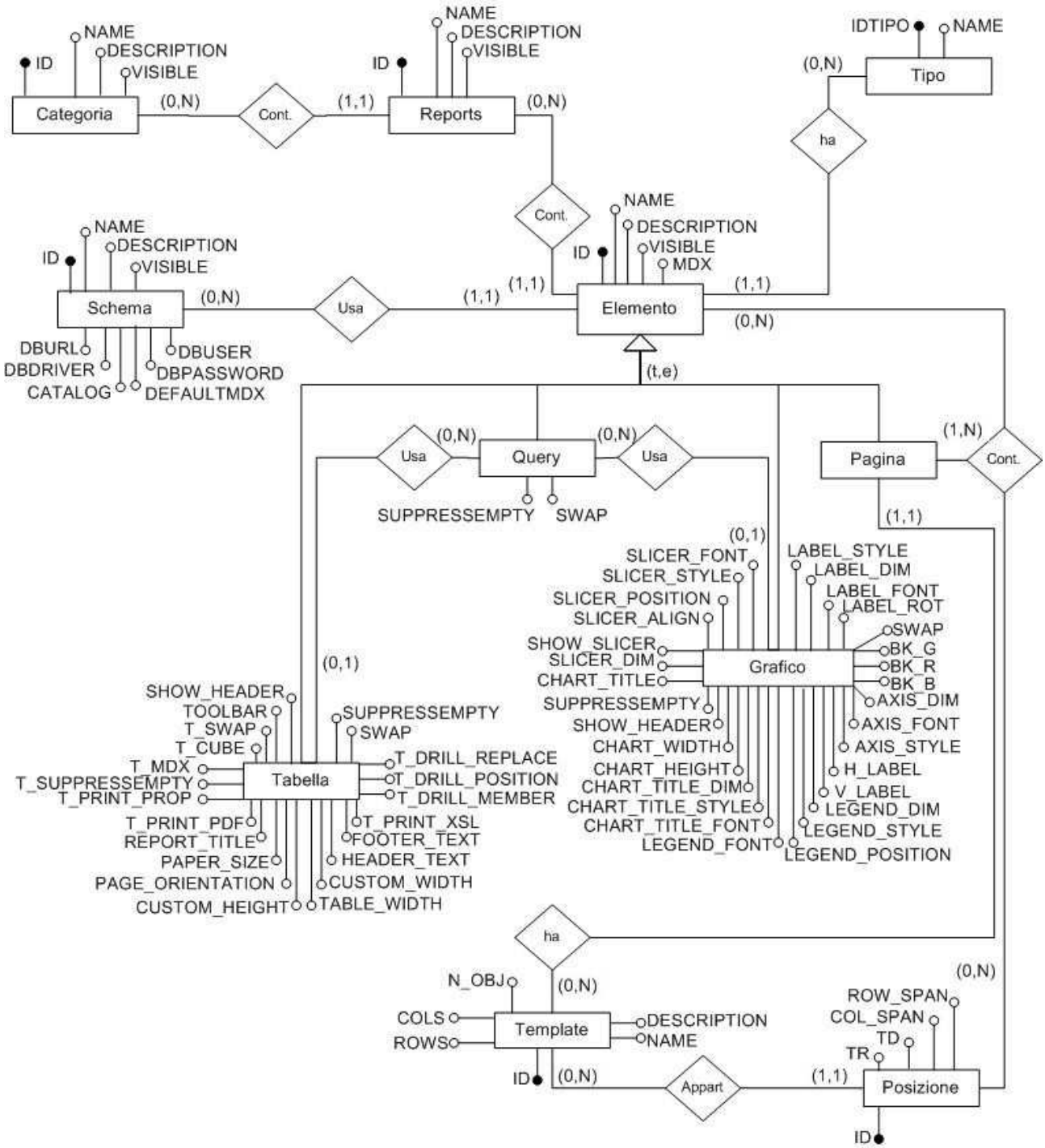


Figura 24 - Modello ER del database dell'applicazione

3.2.1 GESTIONE DEGLI SCHEMI E DELLE SORGENTI DATI

In questa sezione dell'applicazione vengono gestite tutte le possibili connessioni alle sorgenti dati disponibili nell'applicazione. L'accesso a questa sezione è consentita solo agli utenti con diritti di amministrazione. Non è detto che il rapporto tra sorgenti dati e cubi sia di 1:1, ad esempio è possibile definire più cubi all'interno di un'unica sorgente dati a patto che:

- tutte le tabelle dimensionali e dei fatti di tutti i cubi siano contenute all'interno dello stesso schema della stessa istanza di DBMS;

- all'interno del file di catalogo siano definiti entrambi i cubi.

Nella figura seguente possiamo vedere la prima schermata di questa sezione che mostra l'elenco degli schemi già definiti all'interno del sistema.

Figura 25 - Lista degli schemi

Da questa schermata è possibile accedere a tutte le operazioni effettuabili sugli schemi. Cliccando sull'immagine del cubo, a sinistra della lista è possibile vedere in dettaglio le informazioni inserite, tramite l'immagine con la matita a destra, modificare i valori immessi, mentre con l'ultima immagine a destra è possibile cancellare uno schema. Tramite il pulsante nuovo, posto sopra la lista, è possibile creare un nuovo schema. Ogni schema è caratterizzato dai seguenti attributi:

- un identificativo numerico;
- un nome per lo schema;
- una descrizione;
- utente e password per l'accesso al database;
- URL di connessione al database;
- nome del driver da utilizzare;
- URL del file di catalogo;
- query di default dello schema per abilitare la funzione di "Query-by-example" del "Modeller" in caso di nuove interrogazioni.

Le figure successive mostrano le schermate di modifica e di dettaglio di uno schema.

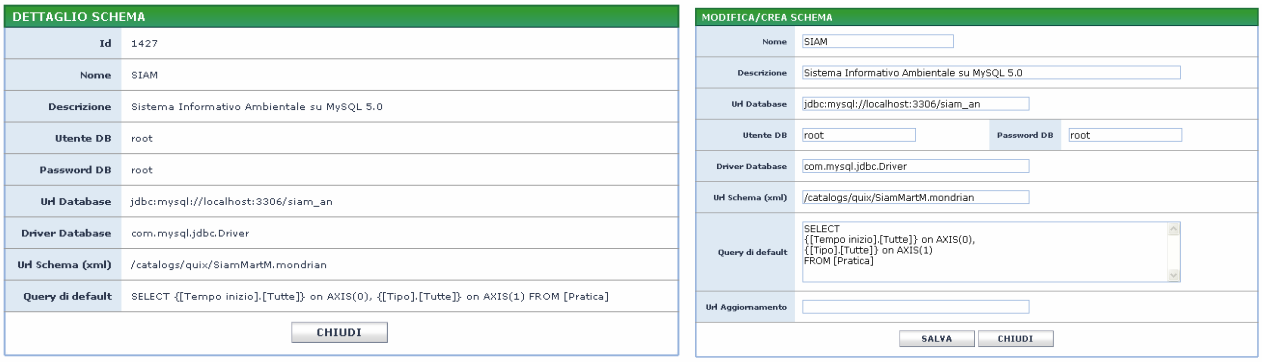


Figura 26 - Dettaglio e modifica di uno schema

Naturalmente non è possibile inserire due schemi con lo stesso nome, altrimenti non sarebbe più possibile per gli utenti identificarli quando richiesto all'interno degli attributi di un report, di una tabella o di un grafico.

3.2.2 GESTIONE DELLE CATEGORIE E DEI GRUPPI DI REPORT

Questa sezione dell'applicazione divide tutti gli elementi contenuti e salvati all'interno dell'applicazione all'interno di una struttura categoria / report. Per semplificare e fare un paragone con i file system, possiamo pensare a una struttura composta da cartelle di primo e secondo livello, con gli elementi contenuti solo all'interno di cartelle di secondo livello.

L'accesso a questa sezione è consentito a tutti gli utenti, tuttavia solo agli utenti con diritti d'amministrazione potranno modificare, inserire o cancellare categorie o gruppi di report.

Nella figura seguente possiamo vedere la prima schermata di questa sezione che mostra l'elenco delle categorie già definite all'interno del sistema.

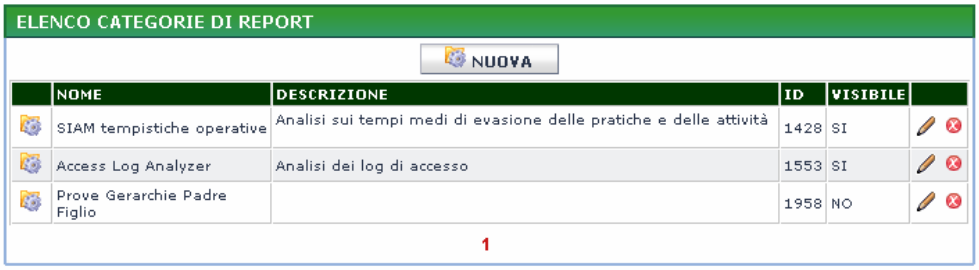


Figura 27 - Elenco delle categorie

Com'è possibile notare dalla figura, le categorie sono solitamente utilizzate per distinguere gli ambiti applicativi da cui provengono i dati dei report. Per ogni categoria è possibile definire:

- un identificativo numerico;
- un nome;
- una descrizione;
- se è visibile o meno agli utenti senza diritto di amministrazione.

La possibilità indicata nell'ultimo punto dell'elenco precedente, presente anche nei singoli elementi, è pensata per nascondere un elemento, un gruppo di elementi o una categoria finché è ancora in progettazione.

La figura successiva mostra la schermata di modifica di una categoria.

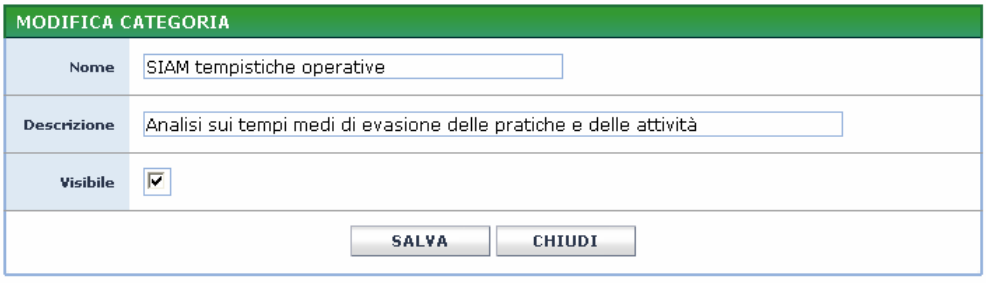


Figura 28 - Modifica di una categoria

Analogamente alle categorie, si comportano anche i gruppi di report. Nella figura seguente possiamo vedere la lista dei gruppi di report presenti all'interno della lista "SIAM Tempistiche Operative".

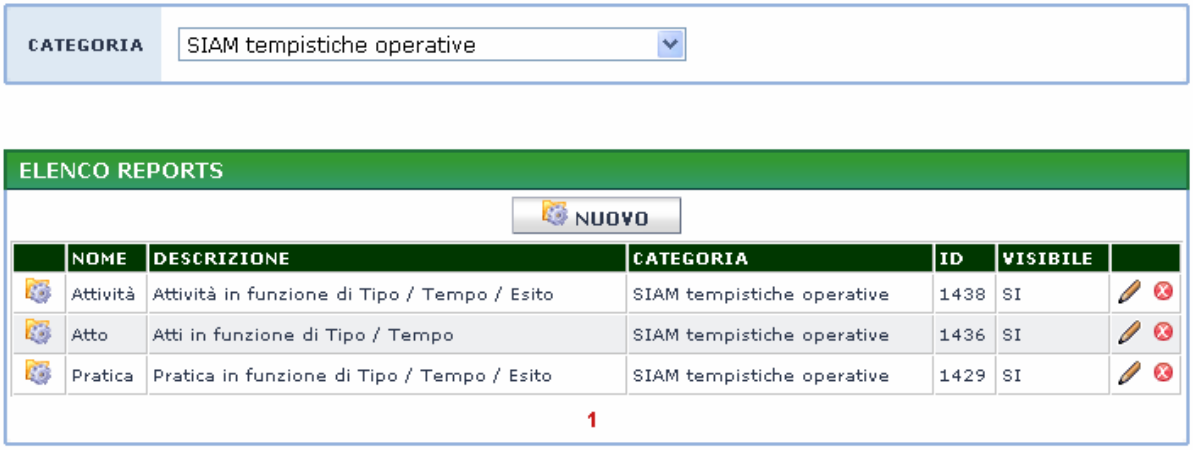


Figura 29 - Elenco di gruppi di report

Nella figura successiva possiamo vedere la modifica e i dettagli di un gruppo di report.

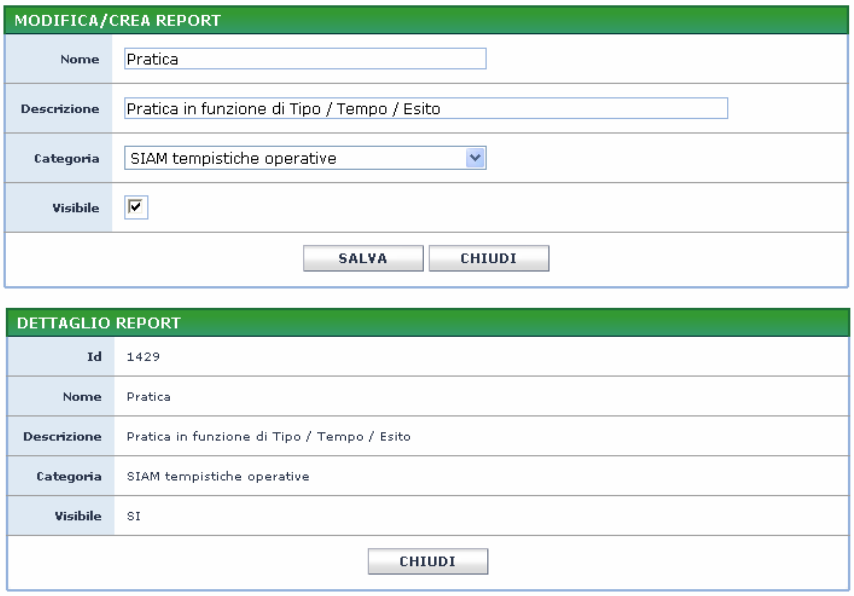


Figura 30 - Dettaglio e modifica di un gruppo di report

3.2.3 GESTIONE DELLE LISTE DEGLI ELEMENTI

Questa sezione dell'applicazione gestisce le liste di elementi contenute all'interno dei gruppi di report. L'accesso a questa sezione è consentito a tutti gli utenti, tuttavia solo agli utenti con diritti di amministrazione potranno modificare, inserire o cancellare elementi, gli altri potranno solo avviarne l'esecuzione e visualizzarne i risultati.

Nella figura seguente viene mostrata la pagina che contiene la lista degli elementi del gruppo di report "Pratica".

Figura 31 - Esempio di elenco di elementi

Notiamo, in alto nella figura, i quattro pulsanti per creare i quattro tipi di elementi possibili e descritti nei paragrafi successivi. Notiamo inoltre, la sezione in alto che permette di filtrare la lista sottostante a seconda del tipo di elemento cercato.

La funzione dell'attributo visibile contenuto in tutti i tipi di elementi funziona come precedentemente descritto per le categorie.

Di seguito riportiamo l'immagine di una lista di elementi che contiene più tipi di elementi.

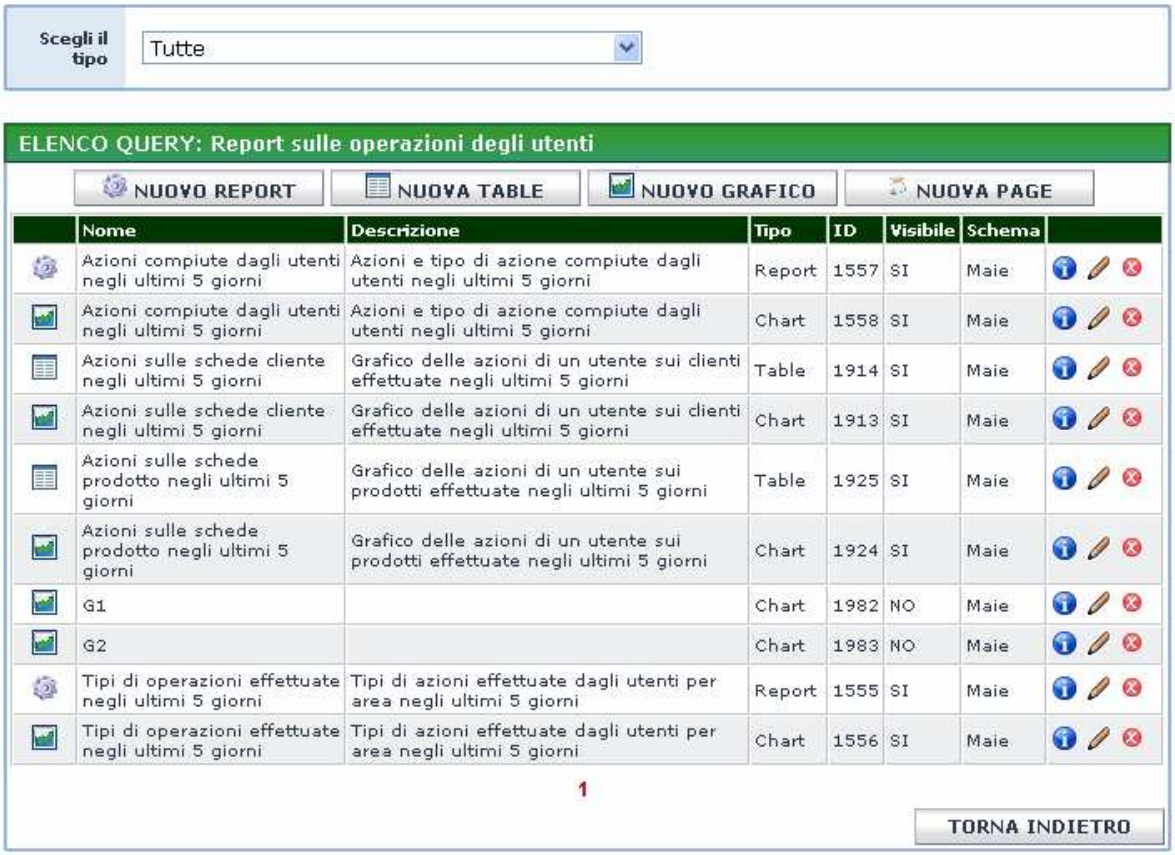


Figura 32 - Lista di elementi con più tipi

3.2.4 GESTIONE DEGLI ELEMENTI DI TIPO "REPORT"

In questa sezione dell'applicazione vengono gestiti gli elementi di tipo report. L'accesso a questa sezione è consentito solo agli utenti con diritti di amministrazione.

Questi tipi di elementi sono pensati per gli utenti più esperti, che a partire da un report di partenza sono in grado di avere accesso a tutte le opzioni possibili sui dati visualizzati. Purtroppo nella versione attuale non è possibile salvare le modifiche effettuate come un nuovo report di uso esclusivo di quell'utente.

Per ognuno di questi elementi, gli attributi che possono essere specificati sono:

- lo schema su cui eseguire l'interrogazione;
- il nome del report;
- una descrizione del report;
- la query MDX da eseguire;
- se il report è visibile;
- se il report è drillthrough enable, cioè se è possibile ottenere l'elenco dei record che hanno generato un determinato valore;
- se il report deve nascondere le righe e le colonne vuote.

Di seguito riportiamo le immagini delle schermate di modifica e di dettaglio di un report.

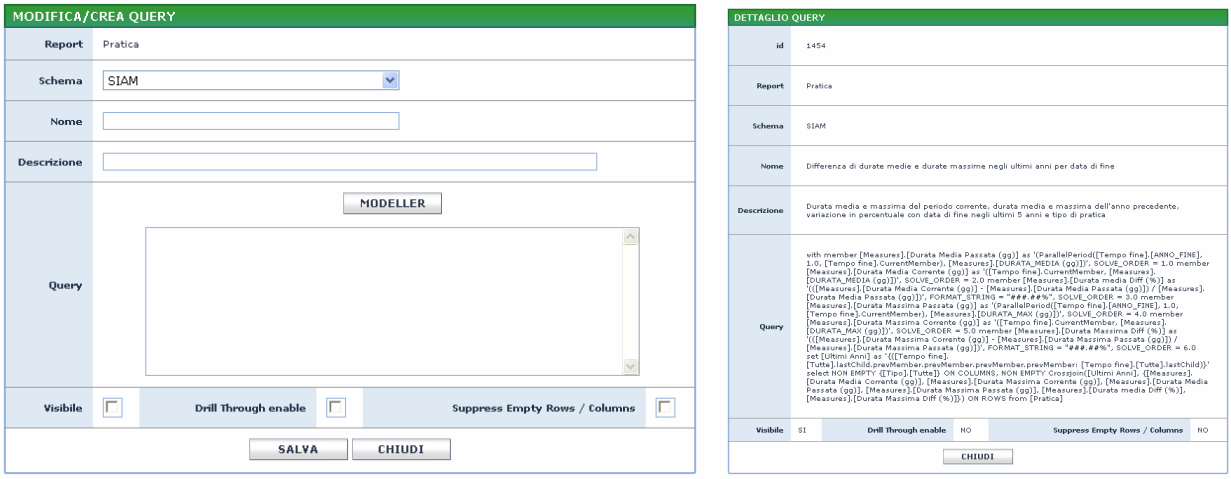


Figura 33 - Dettaglio e modifica di un report

Come si può notare al centro della schermata di modifica, è possibile utilizzare il pulsante "Modeller" per aprire una nuova finestra dove creare un report con il meccanismo di query-by-example. Questa operazione, che non è permessa agli utenti normali che, ad esempio, intendano salvare una loro navigazione, perché solo un utente d'alto livello può capire se il codice dell'interrogazione MDX generato dal tool JPivot è espresso nella miglior forma.

3.2.5 GESTIONE DEGLI ELEMENTI DI TIPO "TABELLA"

In questa sezione dell'applicazione vengono gestiti gli elementi di tipo tabella. L'accesso a questa sezione è consentito solo agli utenti con diritti di amministrazione.

Questo tipo di elemento, è pensato per tre specifiche funzioni:

- limitare le operazioni dell'utente alle sole specificate in fase di creazione della tabella;
- rendere disponibile un elemento più compatto e con meno funzioni disponibili, più adatto ad essere incluso in elementi di tipo pagina (vedi paragrafo 3.2.7);
- salvare le impostazioni di stampa di un report in fase di progettazione, senza che sia necessario specificarle in fase di stampa.

Gli attributi che possono essere specificati per questo tipo d'elemento sono:

- nome della tabella;
- descrizione della tabella;
- l'attributo visibile;
- se la tabella deve essere visualizzata così come è stata specificata nell'interrogazione MDX o con gli assi invertiti;
- se eliminare le righe e le colonne vuote;
- se mostrare o meno l'intestazione con nome, descrizione, filtri, …;
- se la tabella è collegata a un report;
- lo schema da utilizzare per l'interrogazione;
- la query MDX da utilizzare per l'interrogazione;
- se visualizzare la toolbar e se si quali pulsanti visualizzare;
- il titolo della pagina in stampa della tabella;

- l'orientazione della pagina in stampa della tabella;
- la dimensione del foglio in stampa, come formato standardizzato o specificando i valori;
- eventualmente l'altezza personalizzata della tabella;
- il testo di Header per la stampa;
- il testo di Footer per la stampa.

Come si può notare dall'elenco degli attributi è possibile derivare una tabella da un report esistente che appartenga alla stessa sezione dell'applicazione (categoria, gruppo di report). Se una tabella è derivata da un report erediterà da quest'ultimo lo schema da utilizzare per recuperare i dati e il testo della query MDX. Qualsiasi modifica a questi due campi nel report padre si ripercuoterà anche su tutte le tabelle collegate.

Per gli attributi Header e Footer della pagina in caso di stampa è stato configurato un valore di default che viene inserito negli attributi della tabella se l'utente non inserisce niente.

Di seguito riportiamo le immagini delle schermate di modifica e di dettaglio di una tabella.

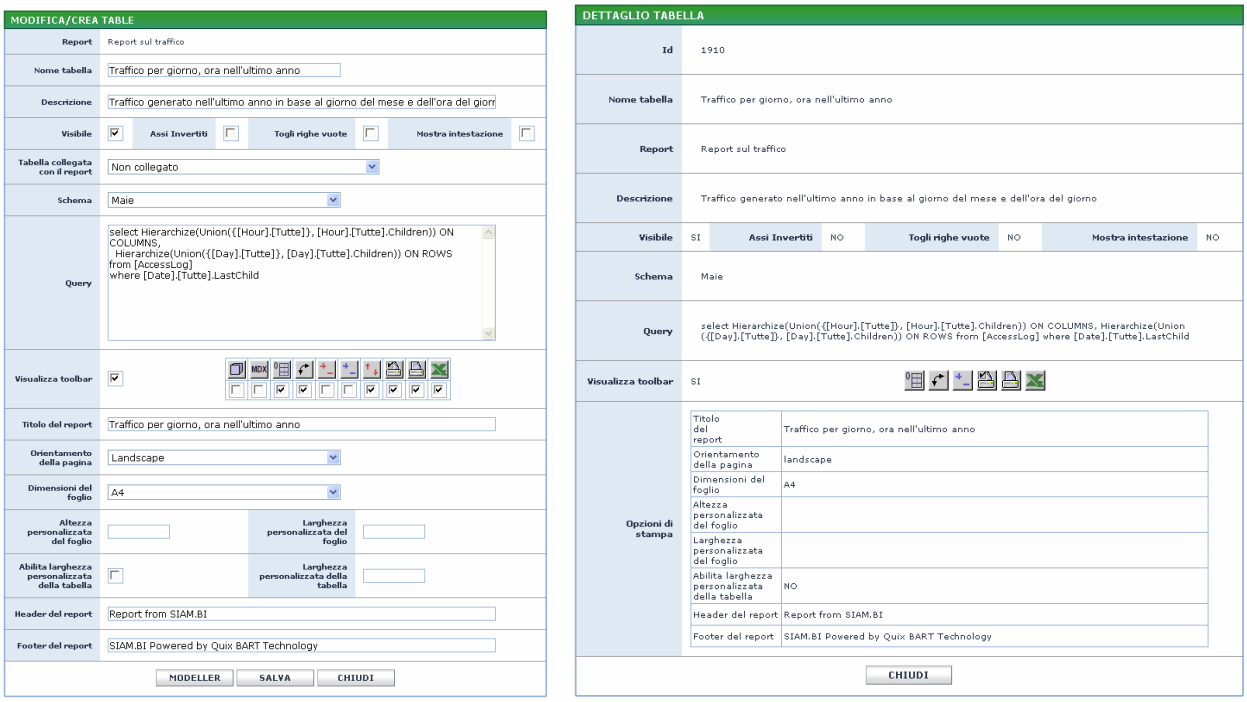


Figura 34 - Dettaglio e modifica di una tabella

Come nel caso precedente possiamo notare a fondo della pagina di modifica il pulsante "Modeller" che permette di creare la tabella con un meccanismo simile a quello query-by-example. Da notare che se una tabella è collegata a un report, si entra nel modeller e si compie qualche azione che modifica l'interrogazione, l'applicazione all'uscita chiederà se salvare le nuove modifiche e in caso affermativo eliminerà il collegamento con il report.

3.2.6 GESTIONE DEGLI ELEMENTI DI TIPO "GRAFICO"

In questa sezione dell'applicazione vengono gestiti gli elementi di tipo tabella. L'accesso a questa sezione è consentito solo agli utenti con diritti d'amministrazione. Le caratteristiche in

termini di funzioni sono simili a quelle disponibili per gli elementi di tipo tabella descritti nel paragrafo precedente. Cambia la funzione di questo elemento, che non è più di mostrare dei dati all'interno di una tabella, ma di mostrare la loro rappresentazione grafica. Le differenze, a livello di funzionalità, con gli elementi di tipo tabella sono:

- non è possibile visualizzare la toolbar;
- non si impostano le proprietà dell'elemento per la stampa, ma quelle per la visualizzazione del grafico;
- un utente che visualizza il grafico non può effettuare operazione di drill-down, roll-up o altre sul grafico.

I tipi di grafici attualmente supportati dal sistema sono:

1. A barre.
2. A barre tridimensionali.
3. A barre impilate.
4. A barre tridimensionali impilate.
5. A linee.
6. Ad area.
7. Ad area impilata.
8. A torta.

Per i primi sette tipi di grafici sono disponibili sia le versioni verticali che quelle orizzontali, mentre per i grafici a torta sono disponibili le versioni per riga e per colonna che cambiano il modo di interpretare i dati producendo grafici differenti (vedi paragrafo 3.4.3).

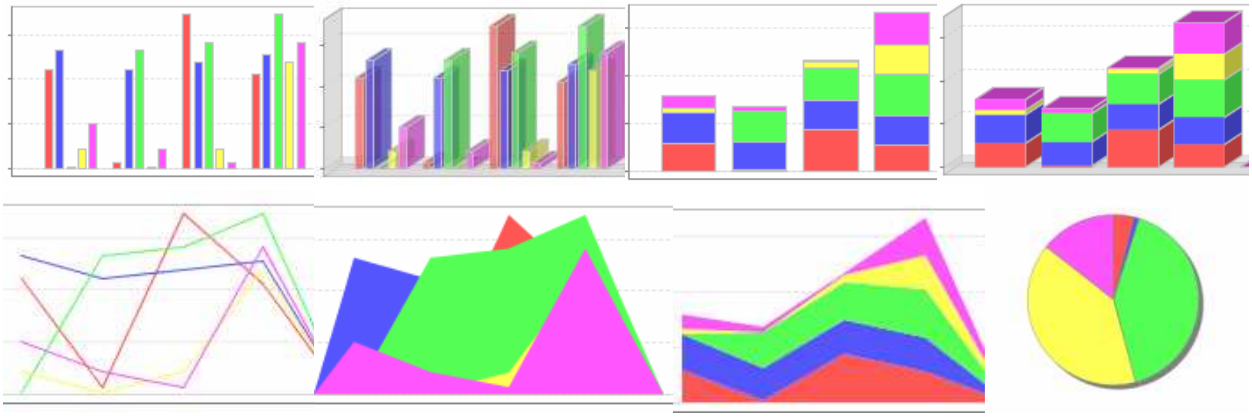


Figura 35 - Esempio di tipi di grafici

Gli attributi che possono essere specificati per questo tipo di elemento sono:

- nome del grafico;
- descrizione del grafico;
- se è visibile;
- se invertire gli assi dell'interrogazione;
- se rimuovere dall'interrogazione le righe e le colonne vuote;
- se mostrare o meno l'intestazione;
- se è collegato a un report;
- lo schema da utilizzare come origine dei dati;

- il testo dell'interrogazione da utilizzare per recuperare i dati;
- il tipo di grafico;
- il titolo del grafico;
- il carattere, in termini di font, dimensione e stile del titolo del grafico;
- il titolo dell'asse orizzontale;
- il titolo dell'asse verticale;
- il carattere, in termini di font, dimensione e stile dei titoli dell'asse verticale e orizzontale;
- il carattere, in termini di font, dimensione e stile delle etichette degli assi e la rotazione delle etichette dell'asse orizzontale in grafici verticali;
- se mostrare la legenda e in che posizione rispetto al grafico;
- il carattere, in termini di font, dimensione e stile della legenda;
- se mostrare i filtri, in che posizione rispetto al grafico e con che allineamento;
- il carattere, in termini di font, dimensione e stile dei filtri;
- la dimensione in pixel dell'area del grafico;
- il colore dello sfondo del grafico, nei tre canali R, G e B.

Per quello che riguarda il collegamento del grafico a un report, il meccanismo è identico a quello già descritto nel paragrafo precedente per le tabelle.

Da notare che sono presenti anche opzioni, come quella che nasconde le righe e le colonne vuote, non proprie di questo elemento (è un grafico, non ha ne righe ne colonne) ma che modificano i dati di partenza da cui generare il grafico. In pratica, nascondere le righe e le colonne vuote significa eliminare dal grafico le posizioni in cui il grafico avrebbe altezza 0. Questa opzione, inutile in un grafico a torta, può essere utile per grafici come quelli a barre per risparmiare spazio (quello occupato dalle barre di altezza 0) o può portare a modificazioni del grafico da valutare con attenzione (come in un grafico a linea o a area, dove l'assenza di uno zero tra valori positivi, trasforma un tratto di grafico a "U" in una linea retta).

Di seguito riportiamo le immagini delle schermate di modifica e di dettaglio di un grafico.

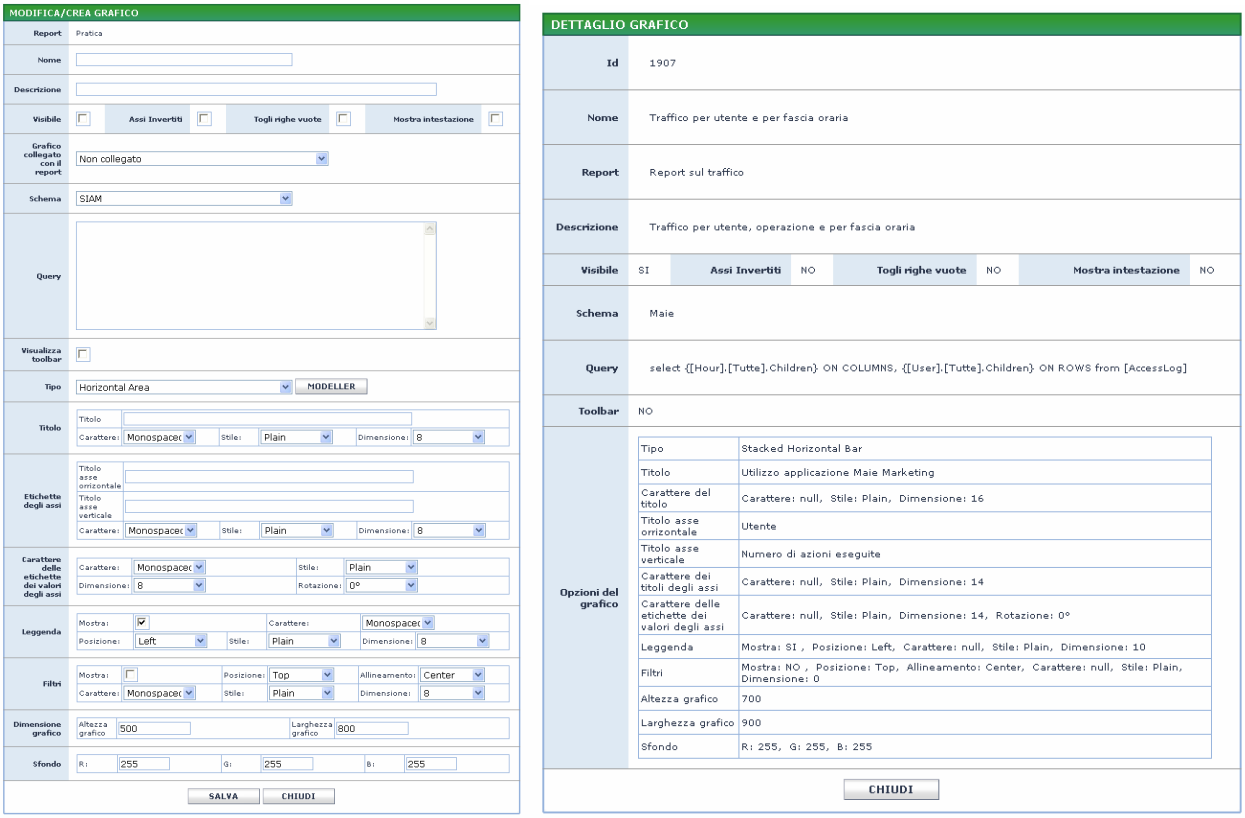


Figura 36 - Dettaglio e modifica di un grafico

Come nel caso precedente possiamo notare al centro della pagina di modifica il pulsante "Modeller" che permette di creare un grafico con un meccanismo simile a quello query-by-example. Da notare, anche in questo caso, che se un grafico è collegato a un report, si entra nel modeller e si compie qualche azione che modifica l'interrogazione, l'applicazione all'uscita chiederà se salvare le nuove modifiche e in caso affermativo eliminerà il collegamento con il report.

3.2.7 GESTIONE DEGLI ELEMENTI DI TIPO "PAGINA"

In questa sezione dell'applicazione vengono gestiti gli elementi di tipo pagina. L'accesso a questa sezione è consentito solo agli utenti con diritti di amministrazione. Questo tipo d'elemento è pensato per creare pagine che contengono più elementi contemporaneamente, disposti secondo un certo criterio. Gli elementi che possono essere inclusi in una pagina sono tutti quelli di tipo tabella o di tipo grafico.

Sfruttando l'attributo visibile, spetta al progettista decidere se i singoli elementi saranno accessibili o solo visualizzabili all'interno dell'oggetto pagina. Ad ogni oggetto incluso è associata una specifica posizione che a seconda di un template selezionato (vedi paragrafo 3.2.8) ne specifica posizione e spazio occupato.

Gli attributi che possono essere specificati per questo tipo di elemento sono:

- nome della pagina;
- descrizione della pagina;
- se è visibile;

- il template che determina la collocazione grafica degli elementi;
- una lista di elementi contenuti.

La procedura di creazione di una pagina comprende due passi che sono illustrati nelle seguenti due figure.

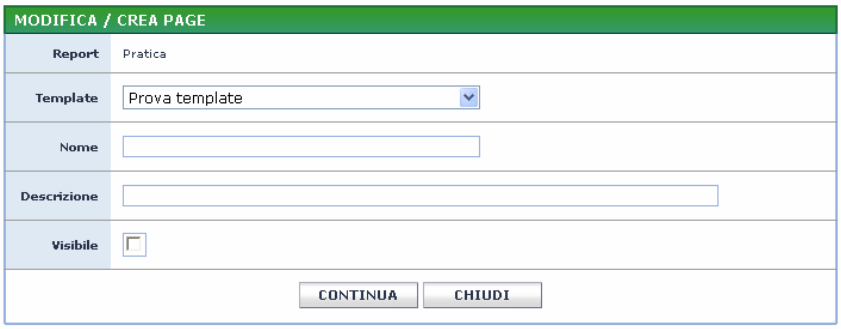


Figura 37 - Creazione di una pagina (1° Passo)

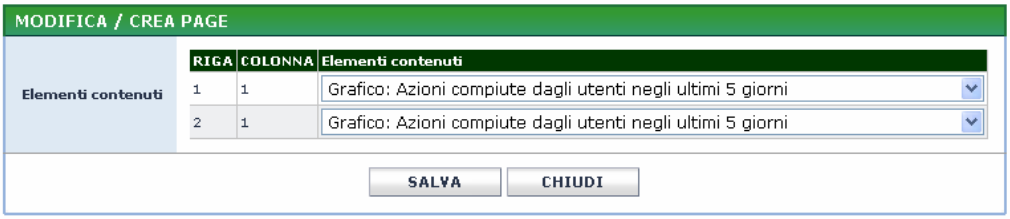


Figura 38 - Creazione di una pagina (2° Passo)

Come al termine di ogni procedura di creazione e/o modifica, è presente una schermata di dettaglio, illustrata nella figura seguente.

Figura 39 - Dettaglio di una pagina

3.2.8 GESTIONE DEI TEMPLATE

In questa sezione dell'applicazione vengono gestiti i template che devono essere utilizzati per modificare la composizione delle pagine. L'accesso a questa sezione è consentito solo agli utenti con diritti d'amministrazione.

Nella figura seguente è illustrata l'interfaccia che elenca i template disponibili all'interno dell'applicazione.

Figura 40 - Elenco dei template

Gli attributi che possono essere specificati per questo elemento sono:

- il nome del template;
- la descrizione del template;
- le dimensioni della griglia su cui si basa;
- le posizioni che contiene.

Come si può notare dall'elenco degli attributi, ogni template è basato su una griglia che rappresenta il numero e la disposizione delle celle che si hanno a disposizione. Ad esempio, una griglia 3 x 2 sarà una tabella formata da 3 righe e 2 colonne (6 celle in totale). Per ogni posizione che si specifica sarà necessario indicare:

- il numero della riga da cui parte la posizione;
- il numero della colonna da cui parte la posizione;
- il numero di righe occupate dalla posizione;
- il numero di colonne occupate dalla posizione.

Nelle due figure di seguito possiamo vedere le schermate di modifica e di dettaglio di un template.

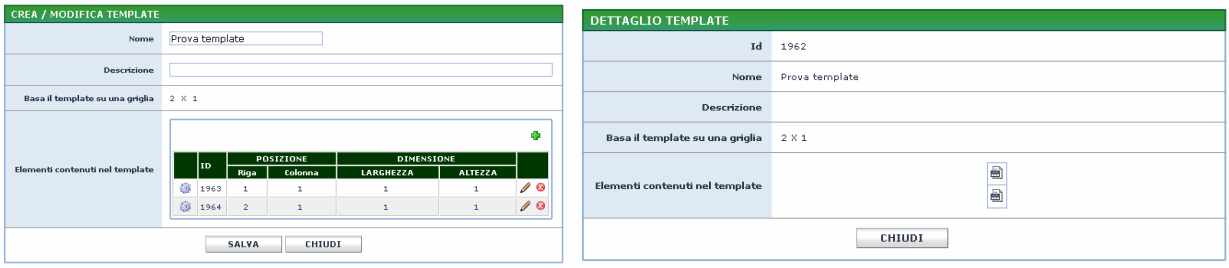


Figura 41 - Dettaglio e modifica di un template

Nella seguente figura possiamo notare l'utilizzo delle dimensioni delle posizioni per creare template più complicati.

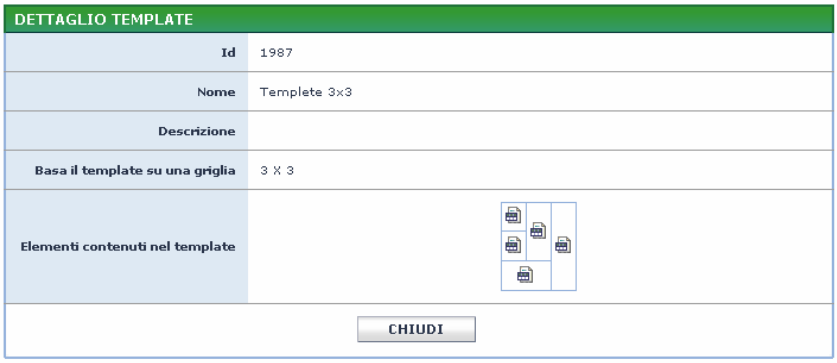


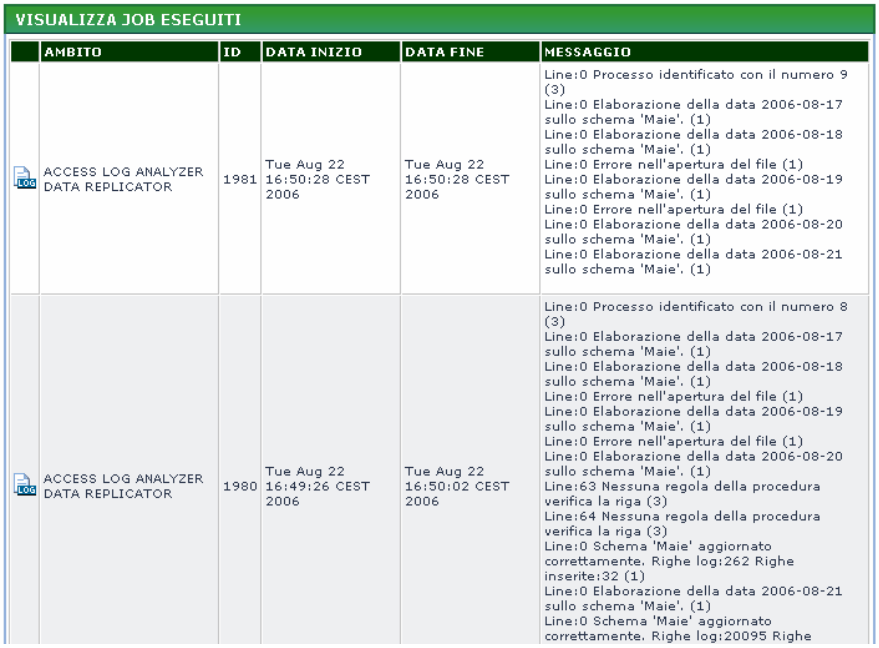
Figura 42 - Esempio di template basato su una griglia 3 x 3

Nel modificare questo elemento, bisogna ricordare che ogni modifica a un template si ripercuoterà su tutte le pagine che avranno adottato quel template.

3.2.9 GESTIONE DELLE PROCEDURE SCHEDULATE

Come già visto in precedenza e come sottolineato dai paragrafi 3.3.3 e 3.3.6, le procedure di alimentazione svolgono un ruolo molto importante nella vita di uno schema multidimensionale. Per questo motivo, all'interno dell'applicazione sviluppata, è stata riservata agli amministratori un'area in cui è possibile controllare lo svolgimento delle ultime procedure notturne eseguite sul sistema.

Come possiamo notare dalla figura successiva, la sezione è composta di una tabella che riporta in ordine cronologico decrescente le ultime procedure eseguite sul sistema.



VISUALIZZA JOB ESEGUITI

	AMBITO	ID	DATA INIZIO	DATA FINE	MESSAGGIO
	ACCESS LOG ANALYZER DATA REPLICATOR	1981	Tue Aug 22 16:50:28 CEST 2006	Tue Aug 22 16:50:28 CEST 2006	Line:0 Processo identificato con il numero 9 (3) Line:0 Elaborazione della data 2006-08-17 sullo schema 'Maie'. (1) Line:0 Elaborazione della data 2006-08-18 sullo schema 'Maie'. (1) Line:0 Errore nell'apertura del file (1) Line:0 Elaborazione della data 2006-08-19 sullo schema 'Maie'. (1) Line:0 Errore nell'apertura del file (1) Line:0 Elaborazione della data 2006-08-20 sullo schema 'Maie'. (1) Line:0 Elaborazione della data 2006-08-21 sullo schema 'Maie'. (1)
	ACCESS LOG ANALYZER DATA REPLICATOR	1980	Tue Aug 22 16:49:26 CEST 2006	Tue Aug 22 16:50:02 CEST 2006	Line:0 Processo identificato con il numero 8 (3) Line:0 Elaborazione della data 2006-08-17 sullo schema 'Maie'. (1) Line:0 Elaborazione della data 2006-08-18 sullo schema 'Maie'. (1) Line:0 Errore nell'apertura del file (1) Line:0 Elaborazione della data 2006-08-19 sullo schema 'Maie'. (1) Line:0 Errore nell'apertura del file (1) Line:0 Elaborazione della data 2006-08-20 sullo schema 'Maie'. (1) Line:63 Nessuna regola della procedura verifica la riga (3) Line:64 Nessuna regola della procedura verifica la riga (3) Line:0 Schema 'Maie' aggiornato correttamente. Righe log:262 Righe inserite:32 (1) Line:0 Elaborazione della data 2006-08-21 sullo schema 'Maie'. (1) Line:0 Schema 'Maie' aggiornato correttamente. Righe log:20095 Righe

Figura 43 - Analisi dei risultati delle procedure notturne

In questa sezione si possono ricavare informazioni importanti, che non riguardano solo la sicurezza che la procedura sia stata eseguita correttamente, ma anche i tempi impiegati da ciascuna procedura ed eventuali errori o avvisi generati dalle procedure.

3.2.10 GESTIONE DELLE REGOLE DI ANALISI DEI LOG DI ACCESSO

Come vedremo in seguito sarà possibile analizzare i log di accesso di un'applicazione per estrarne informazioni utili da inserire in un database multidimensionale. Questo meccanismo, sarà spiegato in dettaglio in seguito (vedi paragrafi 3.3.5 e 3.3.6). La parte che viene descritta in questo paragrafo riguarda l'inserimento delle regole che permettono di capire che azione è stata eseguita a partire dall'URL chiamato. Nella figura seguente, viene mostrata la lista di tutte le possibili procedure già inserite nel sistema.

Figura 44 - Lista delle procedure disponibili

Come per ogni elemento sono disponibili anche le schermate per la modifica e il dettaglio.

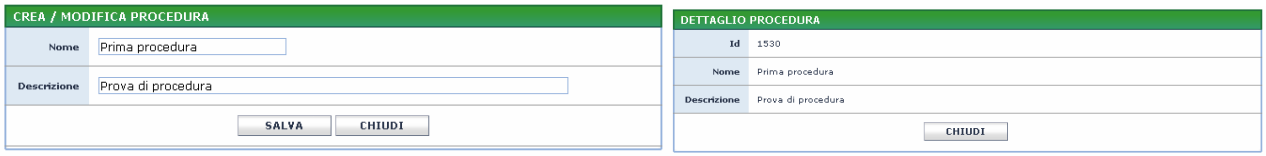


Figura 45 - Dettaglio e modifica di una procedura

All'interno di ogni procedura sono contenute delle regole per l'analisi dei file di log. Nell'esempio precedente, la procedura visualizzata in figura contiene 62 regole. Naturalmente, per eseguire operazioni di analisi, sarà necessario concentrarsi solo su gli URL che meritano l'attenzione. Per questo, come possiamo notare dalla prima pagina di regole della procedura precedente riportata nella figura seguente, tutte le prime regole sono inserite per saltare tutti gli URL che sicuramente non contengono informazioni interessanti.

Figura 46 - Elenco delle regole

Un'analisi più dettagliata della logica che guida le regole è discussa nel paragrafo 3.3.6. Nelle immagini successive vengono illustrate le interfacce di dettaglio e di modifica di una regola.

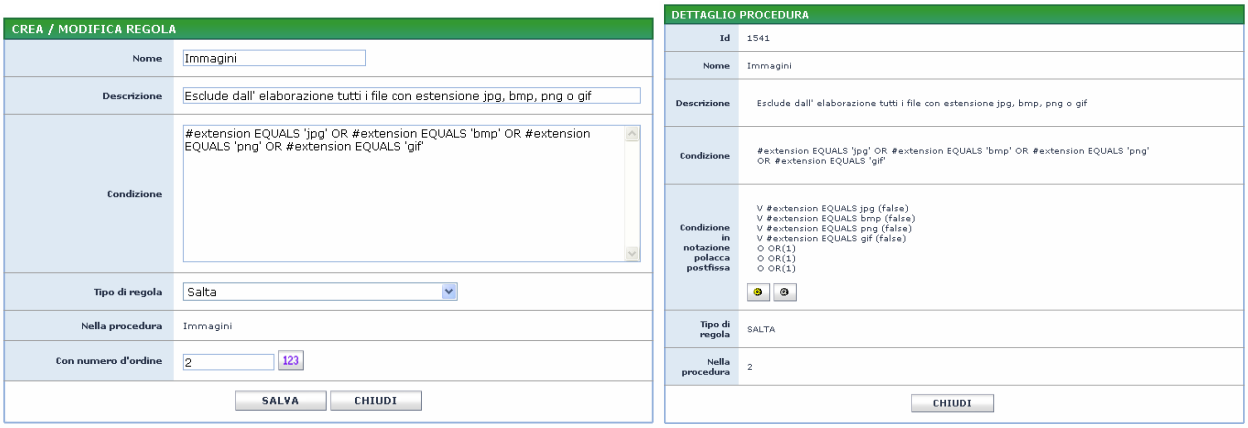


Figura 47 - Dettaglio e modifica di una regola

3.2.11 FUNZIONI MESSE A DISPOSIZIONE PER LA NAVIGAZIONE DEI CUBI

All'interno di questo paragrafo analizzeremo le funzioni messe a disposizione degli utenti per navigare i risultati di un'interrogazione. Il numero d'operazioni disponibili dipenderà comunque dal ruolo dell'utente, dal tipo di elemento che sta visualizzando e dalle proprietà di visualizzazione di quest'ultimo.

Durante la navigazione, l'utente può incontrare diversi tipi di toolbar. Analizziamo le varie opzioni che si potranno presentare a un'utente.


Con questo pulsante si esce dalla navigazione dei dati ritornando alla lista degli elementi contenuti nell'ultimo gruppo di report visualizzato.


Con questo pulsante si nasconde temporaneamente il menù a sinistra per utilizzare tutto lo spazio disponibile per la visualizzazione dei dati. Una volta premuto comparirà un pulsante speculare che permetterà di far ricomparire il menù.


Con questo pulsante si nasconde temporaneamente l'intestazione e il fondo della pagina per utilizzare tutto lo spazio disponibile per la visualizzazione dei dati. Una volta premuto comparirà un pulsante speculare che permetterà di far ricomparire i due elementi.


Con questo pulsante, che compare solo all'interno delle funzioni di "Modeller" è possibile salvare le modifiche effettuate a una interrogazione con un meccanismo di query-by-example.


Con questo pulsante è possibile visualizzare la funzione di "OLAP Navigator" con cui è possibile cambiare le *misure* e le dimensioni dell'interrogazione ed effettuare operazioni di slice dei dati.

Con questo pulsante è possibile visualizzare ed eventualmente modificare il testo MDX dell'interrogazione corrente.


Con questo pulsante è possibile invertire gli assi di analisi.


Con questo pulsante è possibile eliminare dalla visualizzazione eventuali righe e colonne vuote.


Con questo pulsante è possibile attivare la funzione di "drill-member" che permette la classica operazione di "drill-down" sul cubo.


Con questo pulsante è possibile attivare la funzione di "drill-position" che permette di effettuare operazioni di "drill-down" limitate unicamente alla posizione corrente.


Con questo pulsante è possibile attivare la funzione di "drill-replace" che permette di effettuare operazioni di "drill-down" e di "slice" contemporaneamente.


Con questo pulsante è possibile visualizzare un grafico costruito sui dati attualmente visualizzati.


Con questo pulsante è possibile visualizzare le proprietà con cui verrà generato il grafico dal pulsante precedente.


Con questo pulsante è possibile visualizzare le proprietà con cui verranno esportati i dati visualizzati


Con questo pulsante è possibile esportare i dati in formato pdf.


Con questo pulsante è possibile esportare i dati in formato excel.

Analizziamo più in dettaglio alcune delle funzioni messe a disposizione. Sicuramente, la funzione che semplifica notevolmente la navigazione a utenti poco esperti e che non hanno intenzione di imparare il linguaggio MDX è quella di "OLAP Navigator". Questa funzione permette in modo grafico di effettuare operazioni come il cambio di dimensioni di analisi o di *misure* attraverso un'interfaccia grafica.

La figura seguente mostra la schermata principale dell'applicazione.

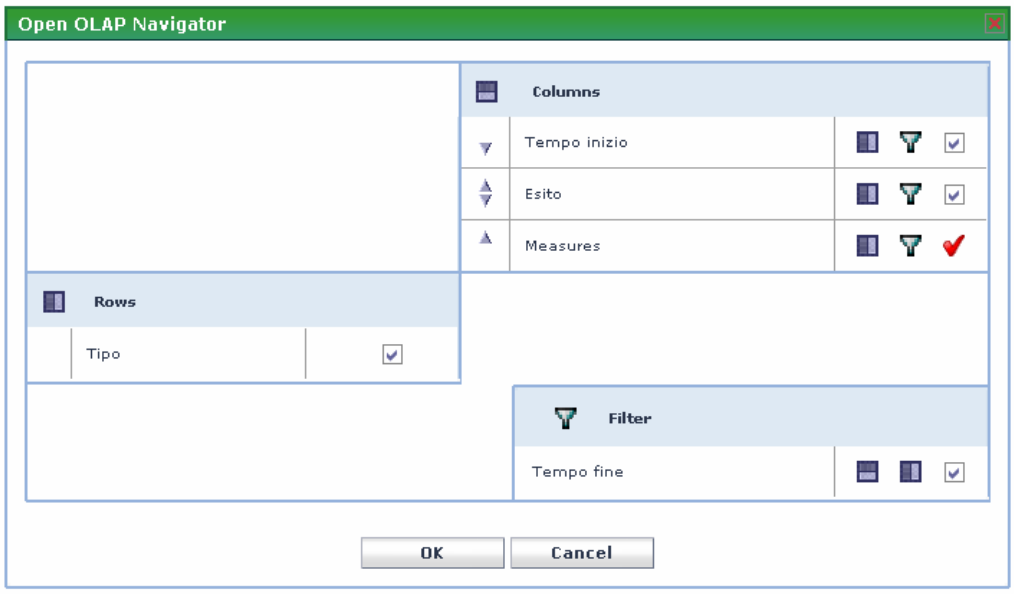


Figura 48 - OLAP Navigator

Per ogni dimensione, compresa quella fittizia delle *misure*, è possibile spostarne la collocazione sulle righe, piuttosto che sulle colonne, piuttosto che tra quelle non usate su cui è possibile fare filtri. Con l'apposito pulsante, con il segno di spunta, è possibile effettuare operazioni di "slice" sulle dimensioni selezionate e di filtro sulle dimensioni non selezionate. Nella figura seguente è mostrata l'interfaccia di selezione di un filtro su un'asse temporale.

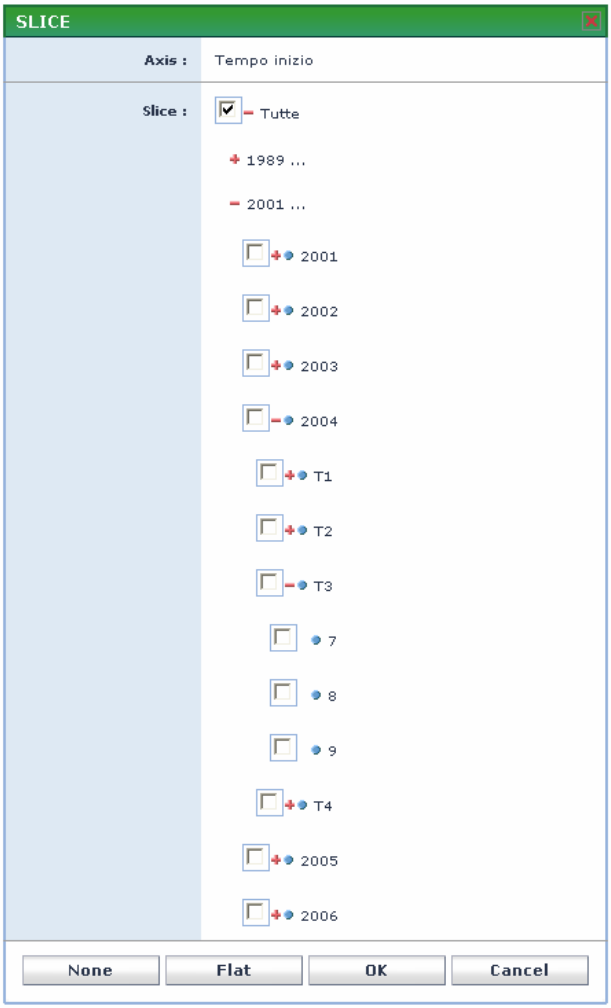


Figura 49 - Slice di una dimensione

La seconda funzione, che permette di modificare l'interrogazione corrente è quella che mostra l'interrogazione MDX. Sicuramente la funzione di "OLAP Navigator" è un ottimo aiuto per costruire semplici interrogazioni, ma se si vogliono ottenere report complessi e un controllo ottimale delle operazioni che dovrà svolgere il server OLAP l'unica alternativa è modificare l'interrogazione MDX.

La figura seguente mostra la schermata di modifica dell'interrogazione MDX.

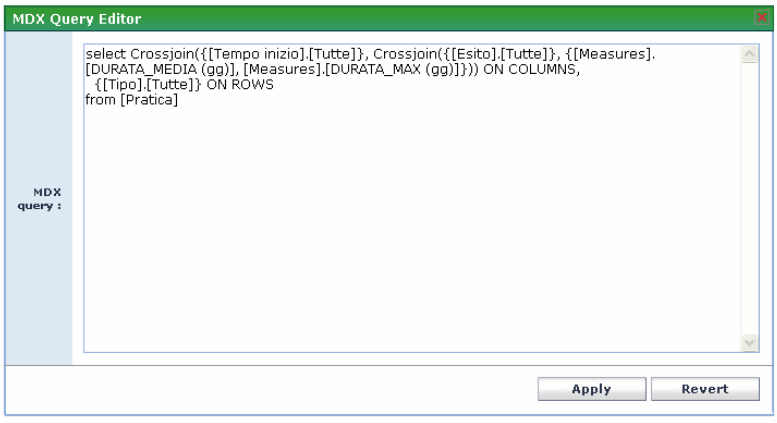


Figura 50 - Modifica dell'interrogazione MDX

Un ultima funzione molto utile, è la possibilità di ordinare la visualizzazione dei membri a seconda del valore di una *misura*, come mostrato nella figura seguente applicata a una *gerarchia* padre-figlio.

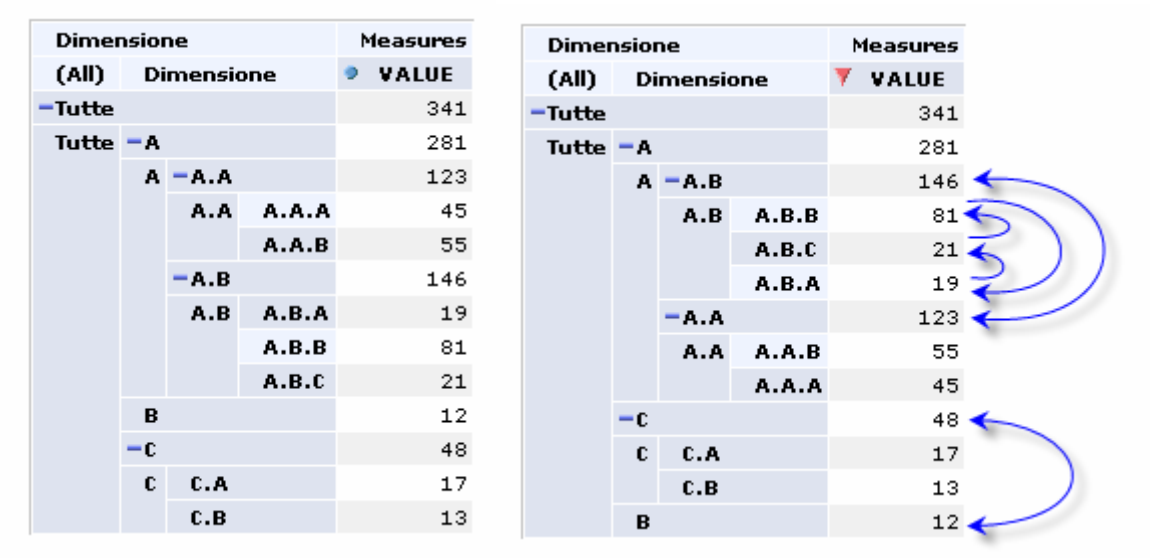


Figura 51 - Funzione di ordinamento su una misura in una gerarchia padre-filgio

3.3 CONTESTI APPLICATIVI DI TEST PER IL PROTOTIPO

3.3.1 PRIMO CONTESTO APPLICATIVO

Il prototipo realizzato è stato testato su due contesti applicativi. Il primo riguarda un servizio di reportistica da affiancare a un'applicazione già esistente che gestisce il ciclo di vita delle pratiche ambientali di una pubblica amministrazione.

All'interno di questo contesto sono stati individuati i tipi di elementi interessanti per la parte di reportistica e per ognuno di essi sono state identificate le dimensioni e le *misure* caratteristiche per quel tipo da inserire all'interno di un cubo. In particolare per questo contesto sono stati creati i seguenti cubi:

1. "Pratica": contiene i dati che riguardano le pratiche che sono state aperte all'interno dell'applicazione e che attualmente sono concluse. Per ogni pratica, le dimensioni interessanti sono:
    - "Tipo": rappresenta il tipo della pratica in esame, è strutturato con una *gerarchia* che comprende i livelli "area", "tema", "tipo procedimento" e "procedimento" dove "tipo procedimento" può assumere i valori 'C' (Controllo) o 'A' (Autorizzazione);
    - "Tempo di inizio": rappresenta la data di inizio della pratica, per quello che riguarda l'applicazione viene strutturata in anno, trimestre e mese;
    - "Tempo di fine": rappresenta la data di chiusura della pratica, per quello che riguarda l'applicazione viene strutturata in anno, trimestre e mese;

- "Esito": rappresenta l'esito che ha avuto la pratica, è strutturata su un unico livello e può assumere i valori: 0 (Da definire), 1 (Positivo), 2 (Richiesta di integrazioni), 3 (Positivo con prescrizione), 4 (Parere non dovuto), 5 (Archiviazione).

Le *misure* interessanti per le pratiche aggregate per le dimensioni precedenti sono:

- "Numero di pratiche";
- "Durata media";
- "Durata massima".

2. "Atti": contiene i dati che riguardano gli atti che sono collegati alle pratiche inserite nel sistema. Per ogni atto, le dimensioni interessanti sono:
   - "Tipo": rappresenta il tipo dell' atto in esame, è strutturato con una *gerarchia* che comprende i livelli "area" e "tema";
   - "Tipo autorizzazione": rappresenta il tipo di autorizzazione dell' atto in esame, è strutturato con una *gerarchia* che comprende i livelli "tipo autorizzazione" e "tipo documento" e il livello "tipo autorizzazione" può assumere i valori 'N' (Normale), 'A' (Autorizzazione), 'D' (Diniego), 'R' (Revoca), 'M' (Modifica), 'V' (Rinnovo), 'S' (Sospensione) o 'Z' (Volturazione);
   - "Tempo": rappresenta la data di un atto, per quello che riguarda l'applicazione viene strutturata in anno, trimestre e mese.

   Si può notare come la dimensione "Tipo autorizzazione" potrebbe essere inclusa nella dimensione "Tipo" sottoforma di due livelli supplementari. È stata scelta la soluzione con due gerarchie distinte in quanto quella con *gerarchia* unica non riuscirebbe a rispondere a interrogazioni del tipo: "quanti atti di una specifica area sono autorizzazioni?".

   L'unica *misura* interessante per gli atti aggregati per le dimensioni precedenti è il "numero di atti".
3. "Attività": contiene i dati che riguardano le attività che sono collegate agli atti inseriti nel sistema. Per ogni attività, le dimensioni interessanti sono:
   - "Tipo": rappresenta il tipo della pratica in esame, è strutturato con una *gerarchia* che comprende i livelli "area", "tema" e "attività";
   - "Tempo di inizio": rappresenta la data di inizio dell'attività, per quello che riguarda l'applicazione viene strutturata in anno, trimestre e mese;
   - "Tempo di fine": rappresenta la data di fine dell'attività, per quello che riguarda l'applicazione viene strutturata in anno, trimestre e mese;
   - "Esito": rappresenta l'esito che ha avuto l'attività, è strutturata su un unico livello e può assumere i valori: 1 (Non definito), 2 (Conclusa), 3 (Archiviata).

   Le *misure* interessanti per le pratiche aggregate per le dimensioni precedenti sono:

   - "Numero di pratiche";
   - "Durata media";
   - "Durata massima".

3.3.2 SCHEMI RELAZIONALI DEI CUBI

Il modello relazionale dei dati contenuti nei tre cubi segue i seguenti tre schemi ER. Si può notare come i tre modelli ER rispettino il classico schema a stella tipico dei data warehouse.

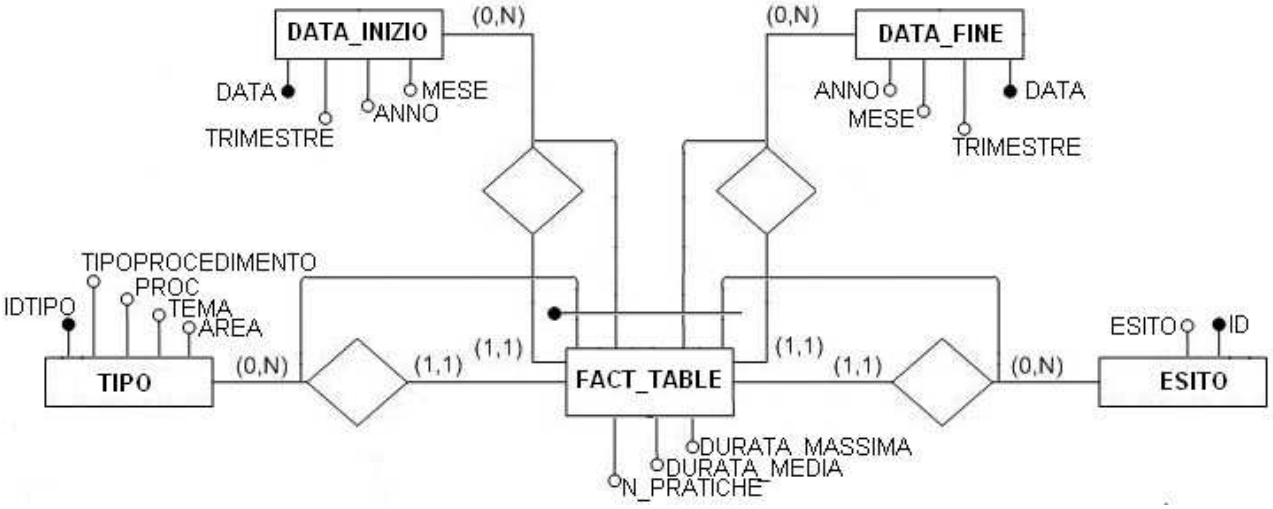


Figura 52 - Modello ER dello schema a stella del cubo Pratica

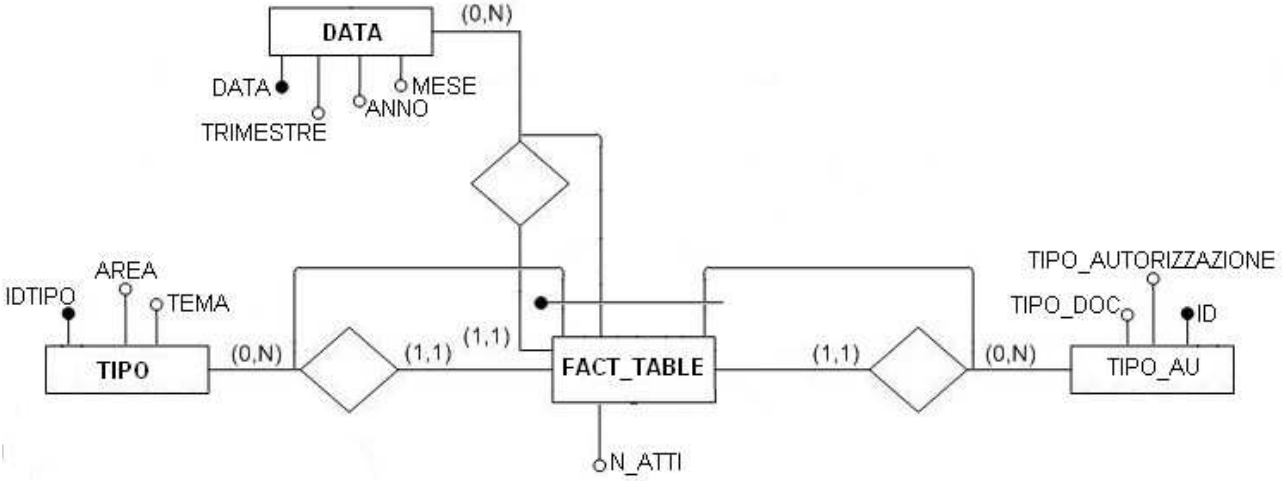


Figura 53 - Modello ER dello schema a stella del cubo Atti

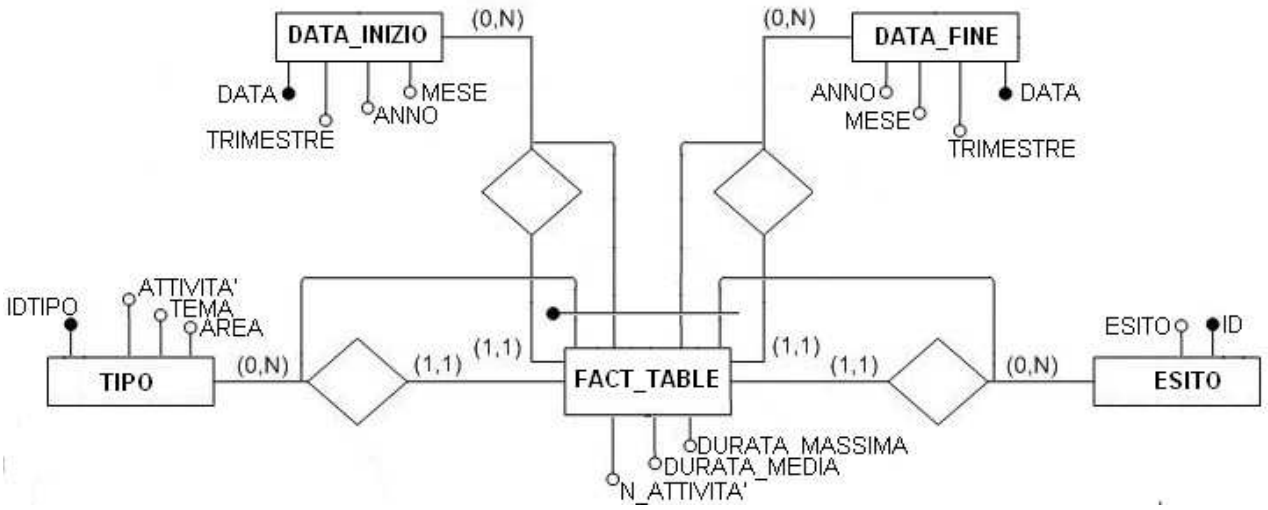


Figura 54 - Modello ER dello schema a stella del cubo Attività

3.3.3 ALIMENTAZIONE DEI DATI

Un aspetto interessante di questo tipo di campo d'applicazione è la tecnica di alimentazione dei dati scelto. Ricordiamo dal paragrafo 2.8 che le tecniche di alimentazione possibili sono:

- statica;
- a marche temporali;
- comparazione di file;
- assistita dall'applicazione;
- trigger;
- log;

Alcuni vincoli posti all'interno di questo contesto limitano le possibilità di scelta.

- L'applicazione già esistente esclude la tecnica assistita dall'applicazione in quanto quest'ultima richiederebbe pesanti modifiche all'applicazione.
- La presenza, all'interno del database, dei soli dati attuali, esclude l'utilizzo della tecnica basata sulla comparazione di file.
- L'applicazione può memorizzare i dati su vari tipi di DBMS. Questo vincolo rende complicata sia la tecnica basata su log che quella basata su trigger, perché in entrambi i casi richiederebbe di riscrivere la stessa procedura di alimentazione dei dati per ogni tipo di DBMS.
- Visto che l'applicazione viene eseguita anche su database che non supportano marche temporali, l'utilizzo di questa tecnica significherebbe modificare tutti gli schemi di database già installati aggiungendo una colonna per ogni tabella interessata e che i dati sono transitori, quindi una eventuale modifica non può essere dedotta dai dati, anche la tecnica basata su marche temporali non può essere applicata.

L'unica scelta che rimane è quella di una tecnica di alimentazione statica. Questa scelta viene anche incoraggiata dalla sua semplicità e dal numero di dati relativamente limitato. Nella

tabella successiva sono riportati le grandezze in numero di tuple, delle tabelle che costituiscono i cubi.

Cubo	*Fact table*	*Tipo*	*Tipo autor.*	*Data inizio*	*Data fine*	*Esito*
Pratiche	2˙005	43	-	895	946	3
Atti	2˙647	10	90	1˙212	-	-
Attività	1˙571	393	-	1˙055	421	2

Tabella 16 - Cardinalità delle tabelle dei cubi del contesto applicativo siambi

Per provvedere alla popolazione delle tabelle è stata creata una procedura che viene lanciata ogni notte che si occupa di cancellare i dati presenti all'interno dei tre cubi, e che provveda a lanciare le interrogazioni necessarie a ripopolarle. Riportiamo, ad esempio, la query per popolare la dimensione tempo del cubo "Pratica" nella versione per "MySQL".

```
INSERT INTO BART_TEMPO_PRATICA_M
SELECT DISTINCT datariczione AS DATA, YEAR(datariczione) AS ANNO,
    MONTH(datariczione) AS MESE, CONCAT('T',QUARTER(datariczione))
                                                        AS TRIMESTRE
from PRATICA
WHERE datariczione IS NOT NULL
AND STATO='CL';
```

La query riportata dipende dal DBMS visto che utilizza funzioni come YEAR, MONTH, … che sono caratteristiche per ogni DBMS. Possiamo notare come la query non si limiti ad alimentare la tabella dimensionale, ma effettui anche operazioni di trasformazione (come concatenare la 'T' al trimestre) e di pulitura (esclude tutte le date nulle o che corrispondono a pratiche non concluse).

La query seguente, che si occupa di alimentare la tabella dimensionale dei tipi all'interno del cubo "Pratica", mostra un esempio di operazioni di trasformazione più complesse in cui il valore del campo "TIPOPROC" viene decodificato.

```
INSERT INTO BART_TIPO_PRATICA_M (AREA, TEMA, PROCEDIMENTO,
                                                IDPROCEDIMENTO, TIPOPROC )
SELECT AREA.nome AS AREA, TEMA.t_descrizione AS TEMA,
     PROCEDIMENTO.nome AS PROCEDIMENTO,
      PROCEDIMENTO.id AS IDPROCEDIMENTO,
      CASE PROCEDIMENTO.tipoproc
            WHEN 'C' THEN 'Controllo'
            WHEN 'A' THEN 'Autorizzazione'
            ELSE 'Altro'
      END AS TIPOPROC
from (AREA join TEMA ON AREA.id=TEMA.t_idarea) join
                            PROCEDIMENTO on TEMA.t_id=PROCEDIMENTO.idtema;
```

Nel prossimo esempio di query, invece, possiamo notare operazioni di pulitura molto più complesse. La query serve per alimentare la fact table del cubo "Pratica" e si pone come scopo di escludere dal calcolo della durata media e massima quelle pratiche che probabilmente hanno errori di inserimento nelle date di inizio o di fine. Per fare ciò, non è più possibile agire come in precedenza escludendo quei record, in quanto questo provocherebbe un decremento del numero di pratiche non voluto. La strada scelta è stata quella di identificare questi record, inserirli ugualmente nella fact table (in questo modo il conteggio non cambia) e modificare la

loro durata ponendola pari alla media delle altre (in modo che non influiscano sul calcolo della durata media). La query tiene conto inoltre del numero di giorni di sospensione della pratica togliendoli dal computo totale.

```
INSERT INTO BART_FACT_TABLE_PRATICA_M
SELECT idprocedimento AS IDPROCEDIMENTO,
       daricezione AS DATA,
       dataconclusione AS DATAFINE,
    (SELECT SUM(DATEDIFF(DATAFINE,DATAINIZIO))
     FROM SOSPENSIONE
    WHERE NOT DATAFINE='2999-12-31'
  1 AND NOT DATEDIFF(DATAFINE,DATAINIZIO)<0
    AND SOSPENSIONE.IDPRATICA=P1.ID
    GROUP BY IDPRATICA) AS GIORNISOSPENSIONE,
    DATEDIFF(dataconclusione,daricezione) AS GIORNIDURATAORIG,
  2 IF(
    ABS(DATEDIFF(dataconclusione,daricezione)-
        (SELECT avg(DATEDIFF(dataconclusione,daricezione)
        -IF((
             SELECT SUM(DATEDIFF(DATAFINE,DATAINIZIO))
             FROM SOSPENSIONE
             WHERE NOT DATAFINE='2999-12-31'
          3  AND NOT DATEDIFF(DATAFINE,DATAINIZIO)<0
             AND SOSPENSIONE.IDPRATICA=P3.ID
             GROUP BY IDPRATICA) IS NOT NULL
        ,
          5  (SELECT SUM(DATEDIFF(DATAFINE,DATAINIZIO))
             FROM SOSPENSIONE
  6          WHERE NOT DATAFINE='2999-12-31'
          4  AND NOT DATEDIFF(DATAFINE,DATAINIZIO)<0
             AND SOSPENSIONE.IDPRATICA=P3.ID
             GROUP BY IDPRATICA)
        , 0)
        ) FROM PRATICA P3
        WHERE dataconclusione IS NOT NULL
         AND NOT dataconclusione='2999-12-31'
        AND DATEDIFF(dataconclusione,daricezione)>0
        AND DATEDIFF(dataconclusione,daricezione)< ?)
    ) > ?
    OR dataconclusione is null
    or DATEDIFF(dataconclusione,daricezione) < 1
   ,
    (SELECT avg(DATEDIFF(dataconclusione,daricezione)
        -IF((
             SELECT SUM(DATEDIFF(DATAFINE,DATAINIZIO))
             FROM SOSPENSIONE
             WHERE NOT DATAFINE='2999-12-31'
          7  AND NOT DATEDIFF(DATAFINE,DATAINIZIO)<0
             AND SOSPENSIONE.IDPRATICA=P4.ID
             GROUP BY IDPRATICA) IS NOT NULL
    9   ,
             (SELECT SUM(DATEDIFF(DATAFINE,DATAINIZIO))
             FROM SOSPENSIONE
             WHERE NOT DATAFINE='2999-12-31'
 10       8  AND NOT DATEDIFF(DATAFINE,DATAINIZIO)<0
```

```
      AND SOSPENSIONE.IDPRATICA=P4.ID
      GROUP BY IDPRATICA)
    , 0))
  FROM PRATICA P4
  WHERE dataconclusione IS NOT NULL
  AND NOT dataconclusione='2999-12-31'
 AND DATEDIFF(dataconclusione,dataricezione)>0
 AND DATEDIFF(dataconclusione,dataricezione)< ? )
,
  DATEDIFF(dataconclusione,dataricezione)-
    IF((
      SELECT SUM(DATEDIFF(DATAFINE,DATAINIZIO))
      FROM SOSPENSIONE
      WHERE NOT DATAFINE='2999-12-31'
   11 AND NOT DATEDIFF(DATAFINE,DATAINIZIO)<0
      AND SOSPENSIONE.IDPRATICA=P1.ID
      GROUP BY IDPRATICA) IS NOT NULL
14
   13 ,
      (SELECT SUM(DATEDIFF(DATAFINE,DATAINIZIO))
      FROM SOSPENSIONE
      WHERE NOT DATAFINE='2999-12-31'
   12 AND NOT DATEDIFF(DATAFINE,DATAINIZIO)<0
      AND SOSPENSIONE.IDPRATICA=P1.ID
      GROUP BY IDPRATICA), 0)
) AS GIORNIDURATA, id AS ID, esito AS ESITO
from PRATICA P1
where STATO='CL';
```

La query per prima cosa calcola i giorni di sospensione (1) e la durata della pratica come differenza tra le due date (2) e li inserisce in due appositi campi per usi futuri. Nella parte contrassegnata con (6) la query calcola se una pratica ha una durata che differisce dalla media di tutte le durate (tolti i giorni di sospensione(5), se esistono(3)) più di una certa soglia o con data di conclusione non nulla o con durata uguale a 0 o negativa.

Se la condizione (6) è vera, allora la pratica viene considerata con una durata errata, e nella parte contrassegnata con (10) la query calcola la durata media delle altre pratiche (9) escludendo i giorni di sospensione (8), se esistono (7), scartando nel computo della media, le pratiche con data conclusione nulla o uguale al 31-12-2999 o durata uguale a 0, minire di 0 o maggiore della soglia prefissata.

Se la condizione (6) non è vera, allora la durata della pratica è corretta, e il campo della durata viene riempito (14) con la durata della pratica tolti i giorni di sospensione (13) che vengono calcolati in (12) se esistono (11).

La query dovrebbe avere altre due sottoquery inserite nei confronti con la soglia che sottraggono dalla durata della pratica una quantità pari alla media delle durate delle pratiche (la logica della query considera errata una pratica se differisce più di una soglia dalla media). Nella query precedente le due sottoquery non sono state inserite in quanto avrebbero rappresentato una complicazione (sia in scrittura che in esecuzione dell'interrogazione) inutile visto che si può ottenere un effetto quasi equivalente impostando la soglia uguale alla soglia voluta incrementata della durata media attesa delle pratiche.

3.3.4 SCHEMA DEL PRIMO CONTESTO APPLICATIVO

Di seguito riportiamo la definizione dello schema da fornire a Mondrian per questo contesto applicativo.

```
<?xml version="1.0" encoding="UTF-8"?>
<Schema name="Siam">
   <Cube name="Pratica">
      <Table name="BART_FACT_TABLE_PRATICA_M"/>
      <Dimension foreignKey="IDPROCEDIMENTO" name="Tipo">
         <Hierarchy allMemberName="Tutte" hasAll="true"
                                                 primaryKey="IDPROCEDIMENTO">
            <Table name="BART_TIPO_PRATICA_M"/>
            <Level column="AREA" name="AREA" type="String"
                                                  uniqueMembers="true"/>
            <Level column="TEMA" name="TEMA" type="String"
                                                 uniqueMembers="false"/>
            <Level column="TIPOPROC" name="TIPOPROC" type="String"
                                                 uniqueMembers="false"/>
            <Level column="PROCEDIMENTO" name="PROCEDIMENTO"
                                       type="String" uniqueMembers="false"/>
         </Hierarchy>
      </Dimension>
      <Dimension foreignKey="DATA" name="Tempo inizio">
         <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="DATA">
            <Table name="BART_TEMPO_PRATICA_M"/>
            <Level column="ANNO" name="ANNO_INIZIO" type="Numeric"
                                                  uniqueMembers="true"/>
            <Level column="TRIMESTRE" name="TRIMESTRE_INIZIO"
                                       type="String" uniqueMembers="false"/>
            <Level column="MESE" name="MESE_INIZIO" type="Numeric"
                                                 uniqueMembers="false"/>
         </Hierarchy>
      </Dimension>
      <Dimension foreignKey="DATAFINE" name="Tempo fine">
         <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="DATA">
            <Table name="BART_TEMPO_FINE_PRATICA_M"/>
            <Level column="ANNO" name="ANNO_FINE" type="Numeric"
                                                  uniqueMembers="true"/>
            <Level column="TRIMESTRE" name="TRIMESTRE_FINE"
                                       type="String" uniqueMembers="false"/>
            <Level column="MESE" name="MESE_FINE" type="Numeric"
                                                 uniqueMembers="false"/>
         </Hierarchy>
      </Dimension>
      <Dimension foreignKey="ESITO" name="Esito">
         <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="ID">
            <Table alias="" name="BART_ESITO_PRATICA_M"/>
            <Level column="ESITO" name="ESITO" uniqueMembers="true"/>
         </Hierarchy>
      </Dimension>
      <Measure aggregator="count" column="ID" formatString="Standard"
                                                      name="N_PRATICHE"/>
      <Measure aggregator="avg" column="GIORNIDURATA"
                          formatString="#,###.00" name="DURATA_MEDIA (gg)"/>
```

```
      <Measure aggregator="max" column="GIORNIDURATA"
                              formatString="Standard" name="DURATA_MAX (gg)"/>
   </Cube>
   <Cube name="Atto">
      <Table name="BART_FACT_TABLE_ATTO_M"/>
      <Dimension foreignKey="IDTEMA" name="Tipo">
         <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="ID">
            <Table name="BART_TIPO_ATTO_M"/>
            <Level column="AREA" name="AREA" type="String"
                                                  uniqueMembers="true"/>
            <Level column="TEMA" name="TEMA" type="String"
                                                 uniqueMembers="false"/>
          </Hierarchy>
      </Dimension>
      <Dimension foreignKey="IDTIPODOC" name="TipoAu">
         <Hierarchy allMemberName="Tutte" hasAll="true"
                                                 primaryKey="IDTIPODOC">
            <Table name="BART_TIPODOC_ATTO_M"/>
            <Level column="TIPOAU" name="TIPOAU" type="String"
                                                  uniqueMembers="true"/>
            <Level column="TIPODOC" name="TIPODOC" type="String"
                                                uniqueMembers="false"/>
          </Hierarchy>
      </Dimension>
      <Dimension foreignKey="DATA" name="Tempo">
         <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="DATA">
            <Table name="BART_TEMPO_ATTO_M"/>
            <Level column="ANNO" name="ANNO" type="Numeric"
                                                  uniqueMembers="true"/>
            <Level column="TRIMESTRE" name="TRIMESTRE" type="String"
                                               uniqueMembers="false"/>
            <Level column="MESE" name="MESE" type="Numeric"
                                                 uniqueMembers="false"/>
          </Hierarchy>
      </Dimension>
      <Measure aggregator="count" column="ID" formatString="Standard"
                                                          name="N_ATTI"/>
   </Cube>
   <Cube name="Attivita">
      <Table name="BART_FACT_TABLE_ATTIVITA"/>
      <Dimension foreignKey="IDTIPOATTIVITA" name="Tipo">
         <Hierarchy allMemberName="Tutte" hasAll="true"
                                           primaryKey="IDTIPOATTIVITA">
            <Table name="BART_TIPO_ATTIVITA"/>
            <Level column="AREA" name="AREA" type="String"
                                                  uniqueMembers="true"/>
            <Level column="TEMA" name="TEMA" type="String"
                                               uniqueMembers="false"/>
            <Level column="ATTIVITA" name="ATTIVITA" type="String"
                                                 uniqueMembers="false"/>
          </Hierarchy>
      </Dimension>
      <Dimension foreignKey="DATA" name="Tempo inizio">
         <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="DATA">
            <Table name="BART_TEMPO_ATTIVITA"/>
```

```
            <Level column="ANNO" name="ANNO_INIZIO" type="Numeric"
                                                    uniqueMembers="true"/>
            <Level column="TRIMESTRE" name="TRIMESTRE_INIZIO"
                                      type="String" uniqueMembers="false"/>
            <Level column="MESE" name="MESE_INIZIO" type="Numeric"
                                                   uniqueMembers="false"/>
         </Hierarchy>
      </Dimension>
      <Dimension foreignKey="DATAFINE" name="Tempo fine">
        <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="DATA">
            <Table name="BART_TEMPO_FINE_ATTIVITA"/>
            <Level column="ANNO" name="ANNO_FINE" type="Numeric"
                                                    uniqueMembers="true"/>
            <Level column="TRIMESTRE" name="TRIMESTRE_FINE"
                                      type="String" uniqueMembers="false"/>
            <Level column="MESE" name="MESE_FINE" type="Numeric"
                                                   uniqueMembers="false"/>
         </Hierarchy>
      </Dimension>
      <Dimension foreignKey="esito" name="Esito">
        <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="ID">
            <Table name="BART_ESITO_ATTIVITA"/>
            <Level column="ESITO" name="ESITO" type="String"
                                                    uniqueMembers="true"/>
         </Hierarchy>
      </Dimension>
      <Measure aggregator="count" column="ID" formatString="Standard"
                                                      name="N_ATTIVITA"/>
      <Measure aggregator="avg" column="GIORNIDURATA"
                           formatString="#,###.00" name="DURATA_MEDIA (gg)"/>
      <Measure aggregator="max" column="GIORNIDURATA"
                            formatString="Standard" name="DURATA_MAX (gg)"/>
   </Cube>
</Schema>
```

3.3.5 SECONDO CONTESTO APPLICATIVO

Il secondo contesto applicativo in cui è stato testato il prototipo riguarda un servizio di reporting per monitorare le attività svolte dagli utenti all'interno di una applicazione Web sviluppata con il framework Jakarta Struts.

Ogni applicazione che utilizza questo framework e utilizza le DispatchAction permette di capire le azioni degli utenti osservando la sequenza di URL di tipo *.do.

3.3.6 TECNICA DI ALIMENTAZIONE

L'aspetto più interessate di questo contesto applicativo è la tecnica di alimentazione del cubo. I dati vengono prelevati ogni notte dai file di log generati da Tomcat, vengono elaborati, vengono estratte le informazioni utili attraverso un sistema di regole configurate dal progettista del cubo, e vengono inserite all'interno dello schema a stella.

Analizziamo queste fasi una per una.

- La prima operazione riguarda la configurazione di Tomcat in modo che generi i file di log degli accessi e che utilizzi un formato adeguato. Per farlo è necessario modificare il file server.xml aggiungendo il seguente frammento:

```
<Valve className="org.apache.catalina.valves.AccessLogValve"
        directory="logs"
        prefix="access_log_"
        suffix=".txt"
        pattern="%h %l %u %t &#34;%r&#34; %s %b %S"
        resolveHosts="false"/>
```

possiamo notare che nella dichiarazione, oltre al nome della classe di tipo valvola che deve essere caricata, è possibile specificare la posizione in cui inserire i file di log, un prefisso e un suffisso per il nome dei file, il formato delle righe e se risolvere o meno i nomi degli host.

Con la configurazione riportata, il server genererà un nuovo file per ogni giorno, posizionandolo all'interno della directory $TOMCAT_HOME/logs/ e chiamandolo access_log_yyyy-mm-dd.txt dove al posto di yyyy-mm-dd verrà inserita la data odierna nel formato indicato. Il formato del file di log è quello standard, eccezion fatta per l'ultimo campo (%S) che include l'identificativo della sessione utente per permettere alla procedura di alimentazione di tracciare le azioni degli utenti. Riportiamo una riga di log di esempio:

```
127.0.0.1    -    administrator    [22/Aug/2006:08:57:53    +0200]    "GET
/siambi/report.do?task=list&searchCategory=1428&reset=true   HTTP/1.1"   200
16957 33C516FE38639753A8FD20DB78888B50
```

Dalla precedente riga di log possiamo ricavare diverse informazioni:

- l'indirizzo IP dell'utente (in questo caso lo stesso host);
- le credenziali con cui è stato loggiato (in questo caso administrator);
- la data e l'ora della richiesta (22 agiosto 2006, alle 8:57:53);
- il metodo http della richiesta (GET);
- URL richiesto (/siambi/report.do?task=list&searchCategory=1428&reset=true);
- il protocollo usato (HTTP 1.1);
- il codice di risposta (200 OK);
- la dimensione della risposta (16˙957 bytes);
- l'id della sessione utente (33C516FE38639753A8FD20DB78888B50).

Se il parametro resolveHosts è uguale a true, Tomcat tenta di risolvere il nome dell'host e di riportare il suo nome al posto dell'indirizzo IP. Per l'applicazione considerata, l'origine della richiesta è poco indicativa (non distingue, ad esempio, tra gli utenti di una stessa LAN) e provocherebbe un rallentamento al server dovuto alle continue richieste di risoluzione di indirizzi.

- Dai file di log generati da Tomcat è possibile estrarre informazioni sulle operazioni effettuate dagli utenti durante la loro navigazione. Requisito fondamentale per raggiungere lo scopo è che l'applicazione utilizzi Struts e che l'applicazione segua alcune linee guida per facilitare l'estrazione dei dati. Nelle applicazioni prese in esame,

gli URL invocati hanno una struttura comune dovuta all'uso delle DispatchAction (Vedi paragrafo 2.9.11) e a una nomenclatura concordata in precedenza delle azioni. Ad esempio, nell'URL precedente:

```
/siambi/report.do?task=list&searchCategory=1428&reset=true
```

si possono ricavare le seguenti informazioni:

- l'utente ha effettuato operazioni all'interno del contesto siambi;
- l'operazione interessa gli oggetti di tipo report;
- l'operazione richiesta è la visualizzazione della lista (task=list).

Quello riportato in precedenza è un caso semplice, in cui è possibile identificare l'azione dell'utente da una sola riga del file di log. Bisogna considerare che nella realtà ci sono molte righe di log che non riguardano azioni (immagini, fogli di stile, script, …) non tutte le azioni vanno a buon fine (codice di risposta > 399) e che l'uso di javascript può portare a richieste al server che possono trarre in inganno. Consideriamo, ad esempio, le seguenti tre righe di log estrapolate in un caso più complesso.

```
127.0.0.1 - administrator [22/Aug/2006:12:13:38 +0200] "GET
/siambi/table.do?task=edit&idreport=1969&searchTipo=0 HTTP/1.1" 200 24171
6A2FAFE4E6F74A30FC5E2D0945B86672

127.0.0.1 - administrator [22/Aug/2006:12:13:48 +0200] "POST
/siambi/table.do? HTTP/1.1" 200 24551 6A2FAFE4E6F74A30FC5E2D0945B86672

127.0.0.1 - administrator [22/Aug/2006:12:14:05 +0200] "POST
/siambi/table.do? HTTP/1.1" 302 - 6A2FAFE4E6F74A30FC5E2D0945B86672
```

Dalla prima riga dell'esempio, possiamo notare che l'utente ha iniziato la modifica di una table (task=edit e id>0). La seconda e la terza riga sono due richieste molto simili (tipo: POST, contesto: siambi, azione: table.do). L'unica differenza è la dimensione della risposta, la prima di 32KB mentre la seconda 0. Da questo dettaglio possiamo capire che nel primo caso si tratta di un refresh scatenato da javascript inviando il form (la pagina che stava visualizzando viene rispedita dal server con le modifiche del caso) mentre nel secondo caso si tratta di un salvataggio, che provoca una redirezione su un altro URL loggato in seguito (per questo la dimensione è 0).

Naturalmente, per fare considerazioni di questo tipo è necessario conoscere molto bene l'applicazione che si sta cercando di tracciare. Queste considerazioni vengono trasformate poi in regole inserite all'interno della procedura di alimentazione.

Nell'immagine successiva possiamo vedere una pagina in cui è riportato un elenco di 10 regole (da notare il paginatore che indica che in totale sono più di 70) create per un'applicazione.

Figura 55 - Esempio di lista di regole

Le azioni contengono tutte al loro interno una condizione. Se questa condizione è vera la procedura eseguirà l'operazione indicata dal tipo di regola, altrimenti procederà a valutare la regola successiva.

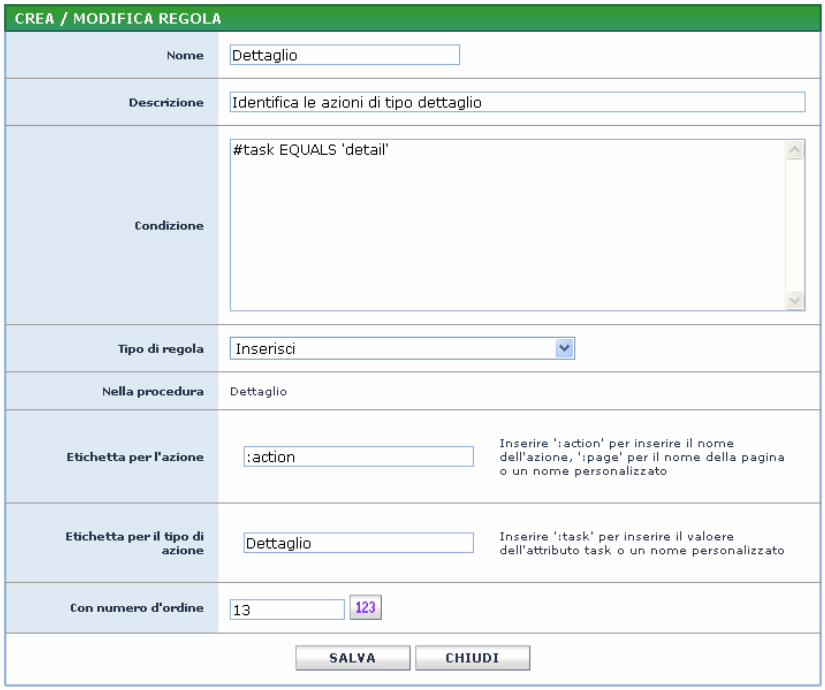


Figura 56 - Esempio di modifica di una regola

Nella figura precedente possiamo notare tutti gli attributi di una regola. In particolare la regola riportata cerca di catturare le visualizzazioni dei dettagli di un'azione qualsiasi (condizione #task EQUALS 'list'). In caso affermativo inserisce una nuova azione nel cubo indicando come nome dell'oggetto modificato il nome indicato nell'URL prima del ".do" e come tipo di azione la stringa "Dettaglio".
Sono possibili 4 tipi di regole:

- "Inserisci": inserisce una nuova azione nel cubo, come nell'esempio precedente;
- "Salta": salta la riga di log perché non interessante;
- "Salva": salva in una variabile di cui è possibile specificare il nome, un determinato valore. Le variabili così salvate sono diverse per ogni sessione utente e per ogni file di log. Viene utilizzato per tracciare le azioni degli utenti su più righe di log;
- "Cancella": elimina un valore salvato con una precedente regola di tipo "Salva".

Da notare come l'ordine delle regole modifica il comportamento dell'applicazione.

- Dall'applicazione delle regole sulle righe dei file di log vengono identificate le azioni compiute degli utenti. Queste ultime devono essere inserite in un cubo strutturato sulle dimensioni del problema. All'interno di questo contesto sono state identificate le seguenti dimensioni:
  - "DateTime", composta dai livelli Anno, Trimestre, Mese, Giorno, Ora;
  - "AzioneTask", composta dai livelli Azione e Task;
  - "User", composta da un unico livello che contiene il nome dell'utente.

  L'unica *misura* interessante per l'applicazione è il numero di azioni. All'interno dello stesso cubo sono state create ulteriori dimensioni per poter incrociare dati che appartengono a una delle dimensioni precedenti (ad esempio, per visualizzare il traffico in funzione dell'ora del giorno senza la dipendenza dal giorno). Le dimensioni che sono state aggiunte sono:
  - "Date", composta dai livelli Anno, Trimestre, Mese e Giorno;
  - "Day", composto dall'unico livello Giorno;
  - "Hour", composta dall'unico livello Ora;
  - "Azione", composta dall'unico livello Azioni;
  - "Task", composta dall'unico livello Task.

3.3.7 SCHEMA RELAZIONALE DEL CUBO

Nella figura successiva è riportato lo schema ER del cubo in esame.

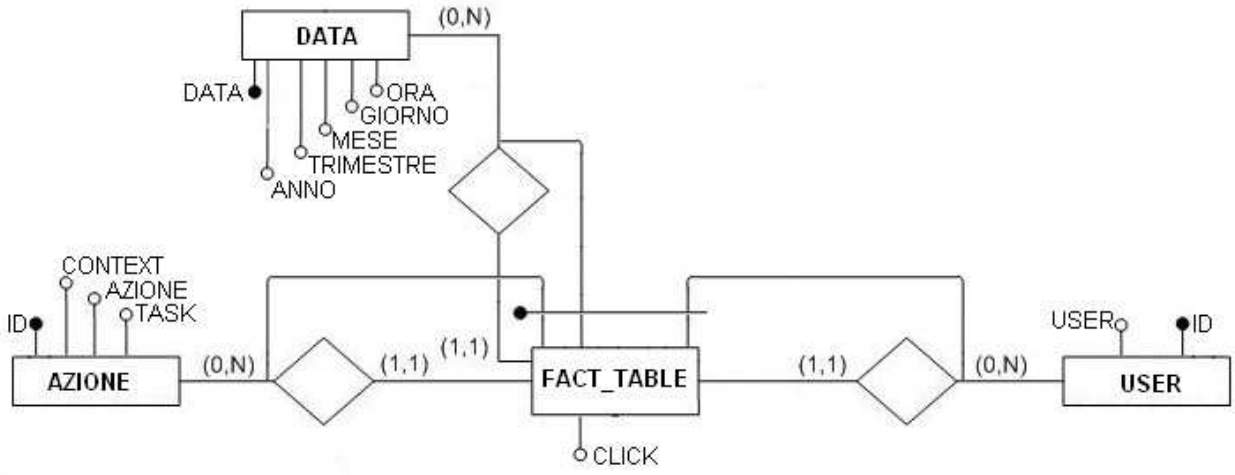


Figura 57 - Modello ER dello schema a stella del contesto di analisi dei log di accesso

3.3.8 PROBLEMI DI ALIMENTAZIONE

Al contrario del contesto precedente, in questo caso si tratta di un cubo ad aggiornamento incrementale. Infatti, i dati vecchi non possono più variare e si hanno a disposizione solo quelli nuovi. In questo caso, la procedura eseguita ogni notte non farà altro che inserire all'interno del cubo i nuovi dati. Questa caratteristica, che a prima vista può sembrare una semplificazione della procedura, nasconde un problema di non facile soluzione. Infatti, data la natura dei dati, le situazioni in cui la procedura di alimentazione venga interrotta a metà lavoro o che due procedure siano in esecuzione più processi di alimentazione in contemporanea sono critiche.

La soluzione del secondo problema è data dal gestore dei processi schedulati che si preoccupa di accodare e eseguire in serie i vari job lanciati. Il primo problema, invece, non può mai essere completamente eliminato (pensiamo ad esempio al caso in cui il sistema rimanga senza corrente o abbia un guasto). Quindi, l'unica soluzione è che la successiva procedura si accorga che la precedente è terminata in modo errato e che sistemi i dati contenuti nel cubo.

Il prerequisito fondamentale per raggiungere lo scopo, è che i processi di aggiornamento siano numerati. Per questo, all'interno del database che contiene il cubo, è presente una tabella, con una colonna e una sola tupla che contiene un numero che identifica univocamente il prossimo processo in esecuzione che dopo aver letto il proprio identificativo provvederà a incrementare la cella di 1.

Con questo prerequisito, la procedura può raggiungere lo scopo di risolvere il problema dell'alimentazione con la seguente strategia.

- Dopo aver letto il proprio identificativo, la procedura legge una tabella speciale in cui in ogni record sono presenti:
  - l'identificativo del processo che ha scritto il record (PID);
  - una data (DATAOK) che identifica l'ultima data processata correttamente;
  - una data (DATACORR) che identifica la data che sta elaborando la procedura;
  - una data (DATAOGGI) che identifica la data in cui è andata in esecuzione la procedura;
  - un flag (K) che identifica se la procedura per quella data è andata a buon fine.

In situazioni di normale funzionamento la situazione della tabella sarà sempre quella riportata in figura (a meno dei valori delle date e dei pid).

PID	DATAOK	DATACORR	DATAOGGI	OK
3	2006-06-27	2006-06-27	2006-06-28	1
3	2006-06-27	2006-06-28	2006-06-28	0

Figura 58 - Tabella dello stato di aggiornamento

In questa situazione, l'ultimo processo che è andato in 28 giugno 2006, ha aggiornato correttamente il file di log relativo al giorno 27 e il prossimo processo inizierà cercando il 28.

Il campo DATAOK serve per gestire il caso in cui un file di log non venga trovato. In questo caso la procedura interessata continuerà a cercare i file successivi e a elaborarli, ma nella tabella rimarrà un record aggiuntivo con DATAOK = data precedente e con OK = 0. Le successive procedure ripartiranno da quella data cercando il file saltato e successivamente quelli che dovrebbero elaborare normalmente. Se un file non viene trovato per un numero di giorni maggiore di una certa soglia, la procedura rinuncerà riportando DATAOK al valore corretto.

Nel caso in cui una procedura sia terminata prematuramente, avrà lasciato OK = 1 nell'ultimo file aggiornato correttamente e quindi la procedura successiva ripartirà da quella.

- Dopo aver ricavato nel punto precedente la data di inizio, la procedura cercherà ogni tupla di tutte le tabelle del cubo, identificata da un pid diverso da 0. Queste tuple sono state inserite da un processo precedente che non è riuscito a terminare correttamente (come ultima operazione doveva riportare a 0 tutti questi campi) e quindi il processo corrente procederà con la loro eliminazione.
- Dopo i due punti precedenti la procedura è pronta per iniziare il proprio lavoro, per ogni file di log ripeterà le seguenti operazioni:
  - inserirà un nuovo record nella tabella dello stato di aggiornamento;
  - cercherà i dati nei file di log e provvederà a inserirli correttamente nel database;
  - aggiornerà la tabella dello stato di aggiornamento;
  - azzererà il campo PID dei record che ha inserito.

  La procedura se, in un'operazione, dovesse riscontrare qualche problema, non svolgerebbe le successive e passerebbe al file successivo. L'unico caso in cui la procedura potrebbe avere problemi è se termina tra il terzo e il quarto passo.
  Per evitare il problema le operazioni sono state inserite all'interno di una transazione.

PID	DATAOK	DATACORR	DATAOGGI	OK
1	2006-08-17	2006-08-17	2006-08-17	1
6	2006-08-17	2006-08-18	2006-08-22	0
6	2006-08-17	2006-08-19	2006-08-22	0
6	2006-08-17	2006-08-20	2006-08-22	0
6	2006-08-17	2006-08-21	2006-08-22	0
7	2006-08-17	2006-08-18	2006-08-22	0
7	2006-08-17	2006-08-19	2006-08-22	0
7	2006-08-17	2006-08-20	2006-08-22	0
7	2006-08-17	2006-08-21	2006-08-22	0
8	2006-08-17	2006-08-18	2006-08-22	0
8	2006-08-17	2006-08-19	2006-08-22	0
8	2006-08-17	2006-08-20	2006-08-22	1
8	2006-08-17	2006-08-21	2006-08-22	1
9	2006-08-17	2006-08-18	2006-08-22	0
9	2006-08-17	2006-08-19	2006-08-22	0

Figura 59 - Esempio di tabella degli stati di aggiornamento

Consideriamo la tabella riportata nella figura precedente. Inizialmente i dati erano aggiornati al 17 agosto. Per 5 giorni non sono stati eseguiti aggiornamenti. Il giorno 22 è stata eseguita la procedura di aggiornamento 4 volte (PID 6, 7, 8 e 9). Nei primi due casi (PID 6 e 7) la procedura falliva proprio tra il terzo e il quarto passo (abbiamo introdotto appositamente un errore), la transazione falliva e il DBMS effettuava il rollback dei dati (Tutti con ok=0). La procedura con PID uguale a 8, è stata avviata senza errori, non ha trovato i log del 18 e 19 agosto e ha aggiornato correttamente i dati del 20 e del 21. Avviando un'ultima volta la procedura (sempre senza errori), quest'ultima si accorge di dover cercare dati dal 18 e che 20 e 21 sono già aggiornati.

3.3.9 SCHEMA DEL CUBO

Riportiamo qui di seguito la descrizione del cubo associato a questo contesto applicativo.

```
<?xml version="1.0" encoding="UTF-8"?>
<Schema name="MaieCRM">
   <Cube name="AccessLog">
      <Table name="BART_ALA_FACT_TABLE"/>
      <Dimension foreignKey="IDDATETIME" name="DateTime">
         <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="ID">
            <Table name="BART_ALA_DATE"/>
            <Level column="ANNO" name="ANNO" type="Numeric"
                                                  uniqueMembers="true"/>
            <Level column="TRIMESTRE" name="TRIMESTRE" type="String"
                                                 uniqueMembers="false"/>
            <Level column="MESE" name="MESE" type="Numeric"
                                                  uniqueMembers="false"/>
            <Level column="GIORNO" name="GIORNO" type="Numeric"
                                                  uniqueMembers="false"/>
            <Level column="ORA" name="ORA" type="Numeric"
```

```
                                                    uniqueMembers="false"/>
      </Hierarchy>
   </Dimension>
   <Dimension foreignKey="IDDATETIME" name="Date">
     <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="ID">
        <Table name="BART_ALA_DATE"/>
        <Level column="ANNO" name="ANNO" type="Numeric"
                                                     uniqueMembers="true"/>
        <Level column="TRIMESTRE" name="TRIMESTRE" type="String"
                                                    uniqueMembers="false"/>
        <Level column="MESE" name="MESE" type="Numeric"
                                                    uniqueMembers="false"/>
        <Level column="GIORNO" name="GIORNO" type="Numeric"
                                                    uniqueMembers="false"/>
      </Hierarchy>
   </Dimension>
   <Dimension foreignKey="IDDATETIME" name="Day">
     <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="ID">
        <Table name="BART_ALA_DATE"/>
        <Level column="GIORNO" name="GIORNO" type="Numeric"
                                                     uniqueMembers="true"/>
      </Hierarchy>
   </Dimension>
   <Dimension foreignKey="IDDATETIME" name="Hour">
     <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="ID">
        <Table name="BART_ALA_DATE"/>
        <Level column="ORA" name="ORA" type="Numeric"
                                                    uniqueMembers="true"/>
      </Hierarchy>
   </Dimension>
   <Dimension foreignKey="IDAZIONETASK" name="AzioneTask">
     <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="ID">
        <Table name="BART_ALA_ACTION"/>
        <Level column="AZIONE" name="AZIONE" type="String"
                                                    uniqueMembers="true"/>
        <Level column="TASK" name="TASK" type="String"
                                                    uniqueMembers="false"/>
      </Hierarchy>
   </Dimension>
   <Dimension foreignKey="IDAZIONETASK" name="Azione">
     <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="ID">
        <Table name="BART_ALA_ACTION"/>
        <Level column="AZIONE" name="AZIONE" type="String"
                                                     uniqueMembers="true"/>
      </Hierarchy>
   </Dimension>
   <Dimension foreignKey="IDAZIONETASK" name="Task">
     <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="ID">
        <Table name="BART_ALA_ACTION"/>
        <Level column="TASK" name="TASK" type="String"
                                                     uniqueMembers="true"/>
      </Hierarchy>
   </Dimension>
   <Dimension foreignKey="IDUTENTE" name="User">
```

```
        <Hierarchy allMemberName="Tutte" hasAll="true" primaryKey="ID">
           <Table name="BART_ALA_USER"/>
           <Level column="UTENTE" name="NOME" type="String"
                                                  uniqueMembers="true"/>
         </Hierarchy>
      </Dimension>
      <Measure aggregator="sum" column="CLICK" formatString="Standard"
                                                              name="CLICK"/>
   </Cube>
</Schema>
```

3.4 ESEMPI DI INTERROGAZIONI REALIZZATE

In questo paragrafo analizzeremo alcune interrogazioni MDX utilizzate dall'applicazione.

3.4.1 INTERROGAZIONI CON PERIODI TEMPORALI

Chiunque crei interrogazioni che prevedano l'uso di insiemi e membri calcolati in combinazione con dimensioni di tipo tempo, potrà rendersi conto della difficoltà di scrivere interrogazioni che non richiedano una manutenzione periodica. Consideriamo, ad esempio, il primo contesto applicativo (vedi paragrafo 3.3.1) e supponiamo di dover creare un'interrogazione che confronti il numero di atti di ogni tipo negli ultimi due anni.

Proprio l'ultima parte dell'interrogazione ("negli ultimi due anni") è quella più difficoltosa da esprimere. Infatti, il metodo più semplice e più immediato, sarebbe di indicare per esplicito l'elenco degli anni da includere nei risultati.

```
SELECT {[Tempo].[2005], [Tempo].[2006]} ON …
```

Il primo limite di questa modalità è l'esplicito elenco di tutti i membri da visualizzare nei risultati che può diventare molto lungo in caso di altre query del tipo "mostrare le *misure* negli ultimi 30 mesi". Per risolvere questo problema è possibile utilizzare la funzione ParallelPeriod messa a disposizione da MDX in combinazione con l'operatore intervallo (:).

```
SELECT {ParallelPeriod([Tempo].[Anno],[Tempo].[2006]
                                              ,-1):[Tempo].[2006]} ON …
```

La funzione ParallelPeriod accetta tre parametri, in successione, un livello, un membro e un intero. La funzione restituisce il cugino del membro indicato come secondo parametro, che si trova nella stessa posizione relativa al livello indicato come primo parametro, considerando come antenato del risultato il membro che si trova nel livello specificato e con posizione che differisce dell'intero inserito come terzo parametro rispetto all'antenato del membro indicato come secondo parametro.

Considerando l'esempio, la funzione ParallelPeriod, calcola l'antenato di [Tempo].[2006] nel livello [Tempo].[Anno] (che sarà il membro stesso), cercherà il membro precedente (-1) e cercherà il membro con la stessa posizione relativa, restituendo [Tempo].[2005].

Consideriamo un esempio più complesso:

```
ParallelPeriod([Tempo].[Anno],[Tempo].[2006].[10].[26],-1)
```

Questa funzione restituirà il membro [Tempo].[2005].[10].[26] (stessa posizione relativa, un anno prima).

Questa modalità ha ancora un problema non trascurabile: ogni anno richiederà una manutenzione per sostituire a 2006, 2007. L'anno seguente si ripresenterà lo stesso problema. Non esiste una soluzione generale per questo problema, in quanto in MDX non esiste una funzione (come now()) che restituisca la data corrente del sistema. Anche in SQL, non viene resa disponibile questa funzione (almeno nello standard) ma molti DBMS sopperiscono alla mancanza provvedendo a darne una implementazione tra le funzioni offerte.

Mondrian non offre questo tipo di funzione. Tutte le funzioni che trattano dimensioni temporali in Mondrian come OpenPeriod, ClosePeriod, ParallelPeriod, … richiedono come parametro il membro che rappresenta la data di riferimento.

Per ottenere il risultato ricercato, esistono alcune tecniche più o meno complicate a seconda del contesto applicativo di riferimento.

- La soluzione più generale al problema consiste nell'aggiungere uno strato di software che emuli il comportamento della funzione now(), che riceve in ingresso la query MDX, sostituisca il valore della funzione now con il membro corretto e poi provveda a passarla a Mondrian. Questa soluzione è sicuramente la più generale, ma anche la più complessa da realizzare.
- Una seconda soluzione, applicabile solo in alcuni contesti, in cui le dimensioni tempo contengono solo date passate o al più odierne. In questo caso, utilizzando la funzione LastChild, è possibile calcolare l'ultimo membro di una dimensione in un certo livello, quindi nel nostro caso l'anno o il trimestre o il mese corrente. In realtà si avranno alcune leggere differenze (sarà considerato come oggi l'ultima data inserita nel cubo). Si tratta di valutare se questa differenza influisce o meno sui risultati trovati. Consideriamo i due contesti applicativi presi in esame per il prototipo.
    - Nel secondo contesto applicativo, la dimensione data contiene, in condizioni normali, tutte le date fino a quella del giorno precedente (ultimo file di log elaborato). In questo caso, questa seconda soluzione è sicuramente esatta (anche se oggi appartenesse a un periodo diverso, non ci sarebbero dati per il nuovo periodo).
    - Nel primo contesto applicativo, è possibile avere differenze più grandi (l'ultima pratica può risalire a un mese fa). In questo caso, la soluzione è utilizzabile se si utilizza un livello abbastanza alto (in caso di interrogazioni interessate agli ultimi anni non ci sono problemi che sorgerebbero se si fosse interessati agli ultimi mesi) e/o un numero di membri compresi nell'intervallo abbastanza alto (negli ultimi 12 trimestri non avrebbe problemi riscontrabili in query interessate agli ultimi due trimestri).
- Una terza soluzione, valida solo per membri che coinvolgono periodi temporali calcolati su livelli alti (es: anni), consiste nel definire l'insieme "Ultimi anni" all'interno del file di schema (vedi paragrafo 2.4.11). Questo file andrà aggiornato tutti gli anni, ma questa unica modifica aggiornerebbe automaticamente tutte le query definite su quel sistema.
- L'ultima soluzione, che è più una estensione della precedente, consiste nel creare una procedura che si preoccupi periodicamente di aggiornare il file di schema, aggiornando le definizioni degli insiemi personalizzati e proceda al reset dell'applicazione Mondrian.

Le soluzioni basate sulle funzioni temporali o sulle funzioni di parentela dei membri soffrono di un ulteriore problema. Se si cerca di costruire l'insieme degli ultimi due anni, sia con

l'aiuto di funzioni come ParallelPeriod, sia con LastChild e PrevMember, se non sono presenti almeno due anni all'interno dei dati del cubo, il risultato sarà vuoto.

Viceversa, nel caso dell'insieme definito nello schema, è possibile utilizzare le funzioni Head e Tail per recuperati i primi o gli ultimi elementi dell'insieme. Queste funzioni, al contrario delle precedenti, se non trovano il numero di elementi cercati, si limitano a restituire quelli che ci sono, risolvendo il problema.

All'interno del prototipo realizzato è stata utilizzata la seconda soluzione. Vediamo, ad esempio, la query corrispondente all'esempio iniziale:

```
select {([Tempo].[Tutte].LastChild.PrevMember.PrevMember :
                                    [Tempo].[Tutte].LastChild)} ON COLUMNS,
  Crossjoin({[Tipo].[Tutte]}, {[TipoAu].[Tutte].Children}) ON ROWS
from [Atto]
where [Measures].[N_ATTI]
```

L'esecuzione dell'interrogazione potrà produrre i seguenti risultati:

			Tempo	
Tipo	TipoAu		Tutte	
(All)	(All)	TIPOAU	+2005	+2006
+Tutte	Tutte	+Autorizzazione	282	58
		+Diniego		
		+Modifica	24	17
		+Normale		
		+Revoca		1
		+Rinnovo		1
		+Sospensione		

Figura 60 - Risultati dell'interrogazione con periodo dinamico

3.4.2 INTERROGAZIONI CON MISURE CALCOLATE IN PIU' PASSI

Consideriamo un secondo problema comune nella scrittura d'interrogazioni MDX. È possibile che si sia interessati a *misure* che non siano direttamente presenti all'interno del cubo, ma ricavabili da esse.

Nell'esempio successivo, si vuole calcolare la differenza percentuale delle durate medie delle pratiche dell'anno precedente, rispetto a quello corrente. Per farlo, abbiamo bisogno della durata media dell'anno corrente (già disponibile sotto forma di *misura*), quella dell'anno precedente (calcolabile con la prima formula della query) e di calcolare la differenza percentuale (terza formula della query d'esempio).

Nell'esempio proposto, oltre alla durata media, viene considerata anche la durata massima, e l'operazione non viene eseguita solo per l'anno corrente ma per gli ultimi 5.

```
with
member [Measures].[Durata Media Passata (gg)] as
           '(ParallelPeriod([Tempo fine].[ANNO_FINE], 1.0,
           [Tempo fine].CurrentMember),
           [Measures].[DURATA_MEDIA (gg)])', SOLVE_ORDER = 1.0
member [Measures].[Durata Media Corrente (gg)] as
           '([Tempo fine].CurrentMember,
           [Measures].[DURATA_MEDIA (gg)])', SOLVE_ORDER = 2.0
member [Measures].[Durata media Diff (%)] as
```

```
           '(([Measures].[Durata Media Corrente (gg)] –
           [Measures].[Durata Media Passata (gg)]) /
           [Measures].[Durata Media Passata (gg)])',
           FORMAT_STRING = "###.##%", SOLVE_ORDER = 3.0
member [Measures].[Durata Massima Passata (gg)] as
           '(ParallelPeriod([Tempo fine].[ANNO_FINE], 1.0,
           [Tempo fine].CurrentMember),
           [Measures].[DURATA_MAX (gg)])', SOLVE_ORDER = 4.0
member [Measures].[Durata Massima Corrente (gg)] as
           '([Tempo fine].CurrentMember,
           [Measures].[DURATA_MAX (gg)])', SOLVE_ORDER = 5.0
member [Measures].[Durata Massima Diff (%)] as
           '(([Measures].[Durata Massima Corrente (gg)] –
           [Measures].[Durata Massima Passata (gg)]) /
           [Measures].[Durata Massima Passata (gg)])',
           FORMAT_STRING = "###.##%", SOLVE_ORDER = 6.0
set [Ultimi Anni] as
           '{([Tempo fine].[Tutte].lastChild.prevMember.
           prevMember.prevMember.prevMember:
            [Tempo fine].[Tutte].lastChild)}'
select NON EMPTY {[Tipo].[Tutte]} ON COLUMNS,
NON EMPTY Crossjoin([Ultimi Anni],
     {[Measures].[Durata Media Corrente (gg)],
     [Measures].[Durata Massima Corrente (gg)],
     [Measures].[Durata Media Passata (gg)],
     [Measures].[Durata Massima Passata (gg)],
     [Measures].[Durata media Diff (%)],
     [Measures].[Durata Massima Diff (%)]}) ON ROWS
from [Pratica]
```

L'esecuzione dell'interrogazione potrà produrre i seguenti risultati:

Tempo fine		Measures	Tipo
(All)	ANNO_FINE	Measure	+Tutte
Tutte	+2002	Durata Media Corrente (gg)	143.28
		Durata Massima Corrente (gg)	318
		Durata Media Passata (gg)	154.87
		Durata Massima Passata (gg)	157
		Durata media Diff (%)	-7.49%
		Durata Massima Diff (%)	102.55%
	+2003	Durata Media Corrente (gg)	168.29
		Durata Massima Corrente (gg)	693
		Durata Media Passata (gg)	143.28
		Durata Massima Passata (gg)	318
		Durata media Diff (%)	17.46%
		Durata Massima Diff (%)	117.92%
	+2004	Durata Media Corrente (gg)	158.84
		Durata Massima Corrente (gg)	739
		Durata Media Passata (gg)	168.29
		Durata Massima Passata (gg)	693
		Durata media Diff (%)	-5.62%
		Durata Massima Diff (%)	6.64%
	+2005	Durata Media Corrente (gg)	163.39
		Durata Massima Corrente (gg)	463
		Durata Media Passata (gg)	158.84
		Durata Massima Passata (gg)	739
		Durata media Diff (%)	2.86%
		Durata Massima Diff (%)	-37.35%
	+2006	Durata Media Corrente (gg)	135.82
		Durata Massima Corrente (gg)	860
		Durata Media Passata (gg)	163.39
		Durata Massima Passata (gg)	463
		Durata media Diff (%)	-16.87%
		Durata Massima Diff (%)	85.75%

Figura 61 - Rsultati dell'interrogazione con misure calcolate da più passi

Consideriamo un ulteriore esempio di interrogazione con *misure* calcolate, utilizzate per costruire un grafico rappresentativo del traffico medio nelle varie ore della giornata. La query può essere scritta come segue:

```
with
set [UltimiGiorni] as
     '{([Date].[Tutte].LastChild.LastChild.LastChild.LastChild :
     [Date].[Tutte].LastChild.LastChild.LastChild.LastChild.
     PrevMember.PrevMember.PrevMember.PrevMember)}'
member [Measures].[Avg] as
     '(Sum([UltimiGiorni], [Measures].[CLICK]) / 3.0)',
      SOLVE_ORDER = 2.0
select {[Measures].[Avg]} ON COLUMNS,
{[Hour].[Tutte].Children} ON ROWS
from [AccessLog]
```

Il grafico risultante sarà simile a quello riportato nella figura successiva:

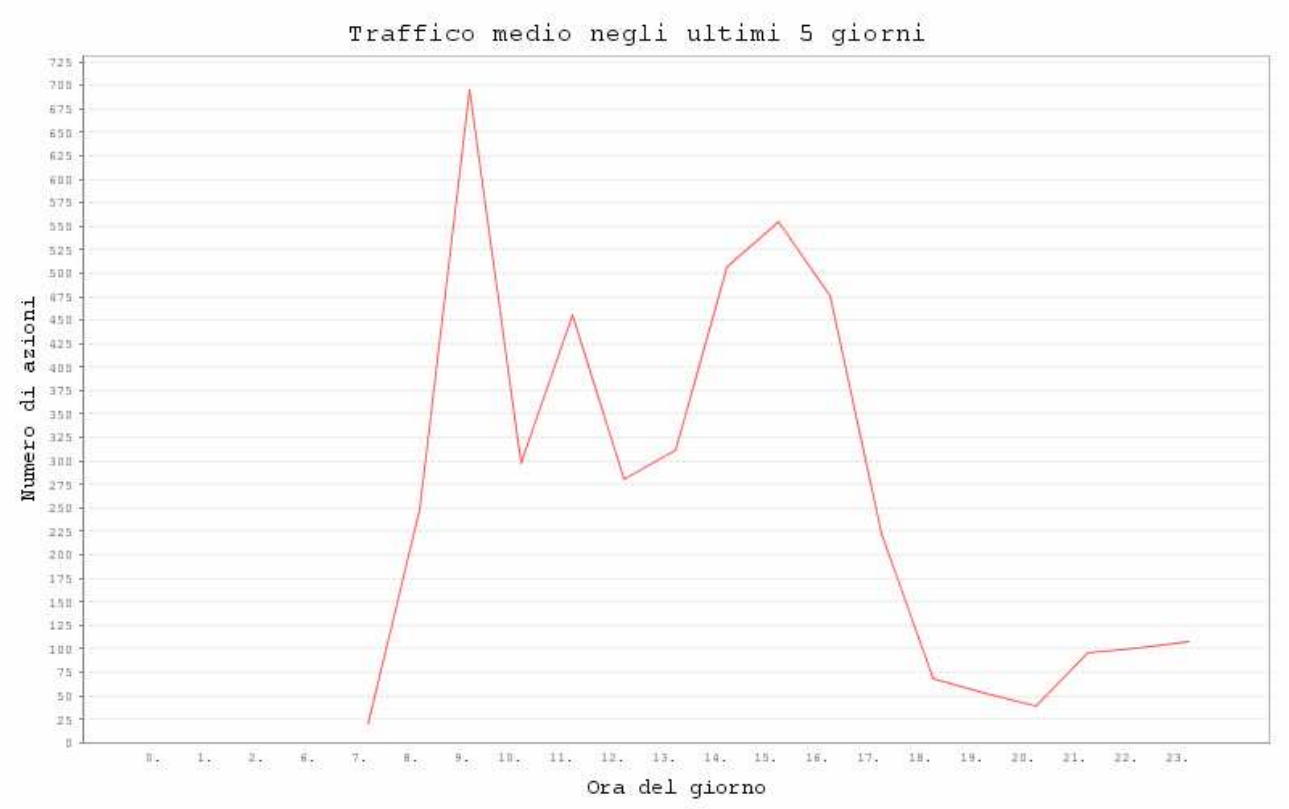


Figura 62 - Esempio di grafico come risultato di una interrogazione

3.4.3 INTERROGAZIONI UTILIZZATE PER PIÙ GRAFICI

Una peculiarità dei grafici all'interno del prototipo è la possibilità di generare due grafici diversi (non solo per l'aspetto grafico) da un'unica interrogazione. Consideriamo la seguente interrogazione:

```
select {[Azione].[Tutte].[city], [Azione].[Tutte].[company],
      [Azione].[Tutte].[contact], [Azione].[Tutte].[product],
      [Azione].[Tutte].[zona]} ON COLUMNS,
{[Task].[Tutte].[Cancellazione], [Task].[Tutte].[Creazione],
      [Task].[Tutte].[Dettaglio], [Task].[Tutte].[Lista],
     [Task].[Tutte].[Modifica]} ON ROWS
from [AccessLog]
```

Da questa query è possibile ricavare due grafici:

- il primo mostra per ogni tipo di azione (es: cancellazioni) come si distribuiscono sulle azioni (es: 40% su prodotti, 10% sui clienti, …);
- il secondo mostra per ogni azione (es: prodotti) come si distribuiscono i tipi di azione su essa (es: 20% cancellazioni, 60% visualizzazioni, …).

I grafici risultanti potranno essere, ad esempio, simili a questi:

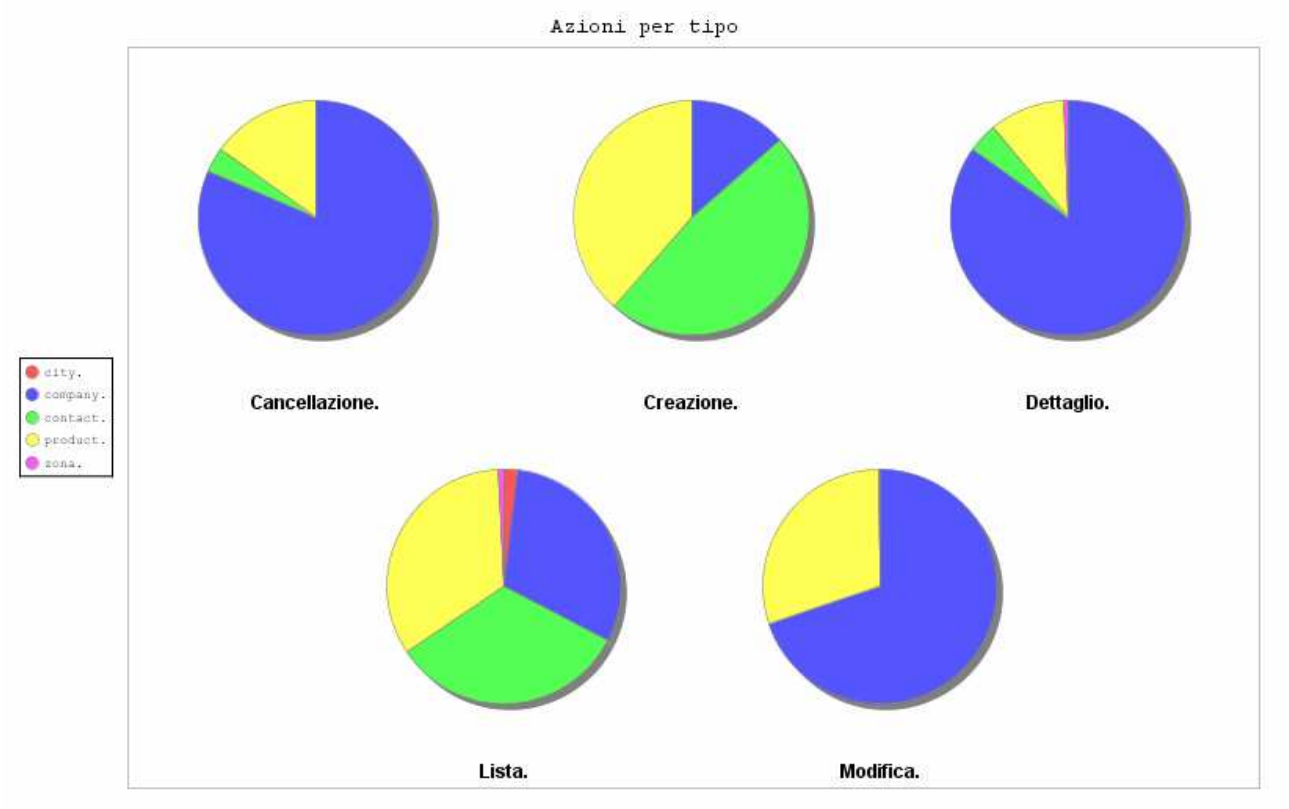


Figura 63 - Grafico per tipo di azione

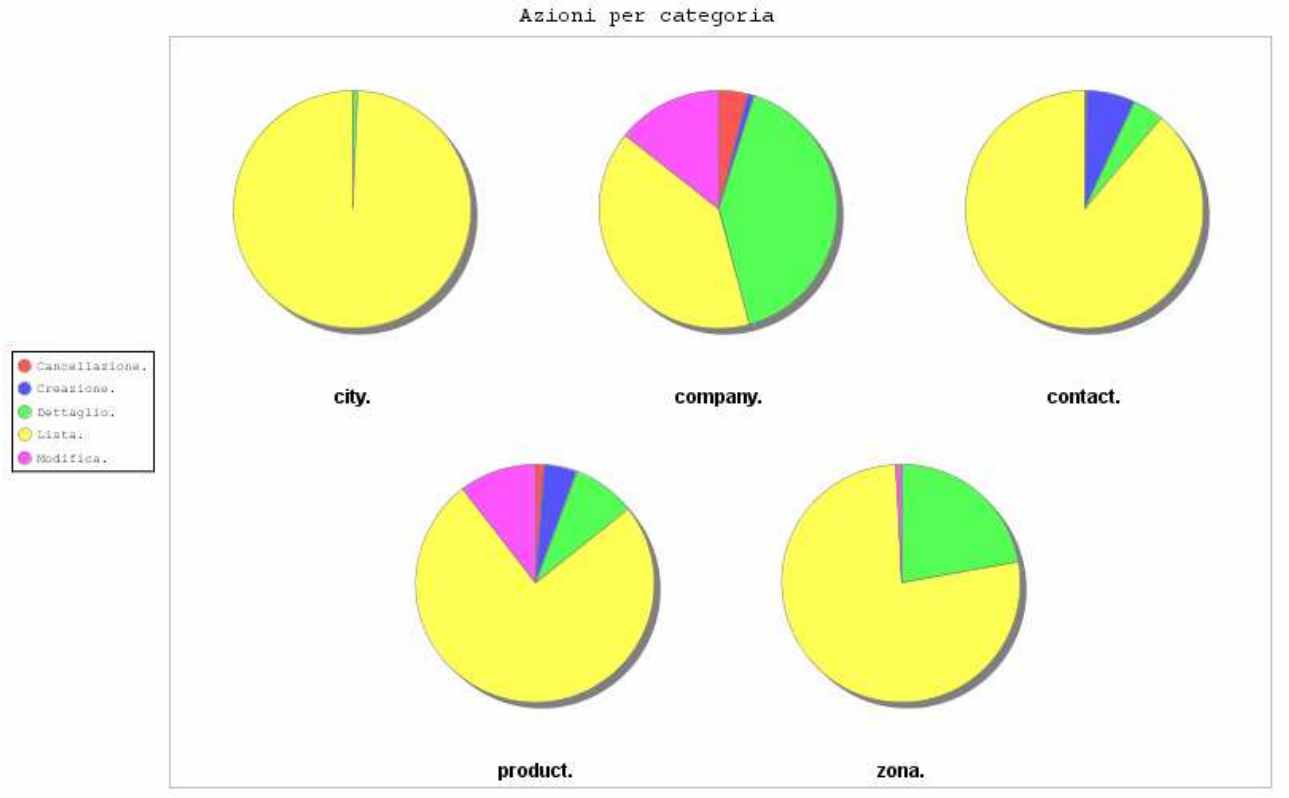


Figura 64 - Grafico per azione

3.5 APPLICAZIONE DELL'ALGORITMO DI SELEZIONE DELLE VISTE A UN CONTESTO REALE

3.5.1 CONTESTO APPLICATIVO E COSTRUZIONE DEL LATTICE

Prendiamo in considerazione un esempio pratico su cui applicare l'algoritmo di selezione delle viste candidate per la materializzazione per aumentare le prestazioni di risposta di un cubo descritto nel paragrafo 2.7.1.

Consideriamo il primo ambito d'applicazione del prototipo realizzato descritto nel paragrafo 3.3.1. All'interno dello schema dell'applicazione sono stati definiti 3 cubi: uno per le pratiche, uno per gli atti e uno per le attività. Le differenze tra i tre cubi, in termini di struttura, sono minime. Per l'applicazione dell'algoritmo consideriamo il terzo, quello degli atti. All'interno del cubo preso in esempio sono definite tre dimensioni:

- data di inizio, con i livelli: anno_inizio, trimestre_inizio e mese_inizio;
- data di fine, con i livelli: anno_fine, trimestre_fine e mese_fine;
- tipo, con i livelli: area, tema, attività.

Per semplicità non consideriamo la quarta dimensione Esito che contiene un unico livello (al contrario delle altre dimensioni che hanno più livelli) e che renderebbe complicato il disegno del lattice. Di seguito riportiamo il lattice che rappresenta la situazione presa in esame.

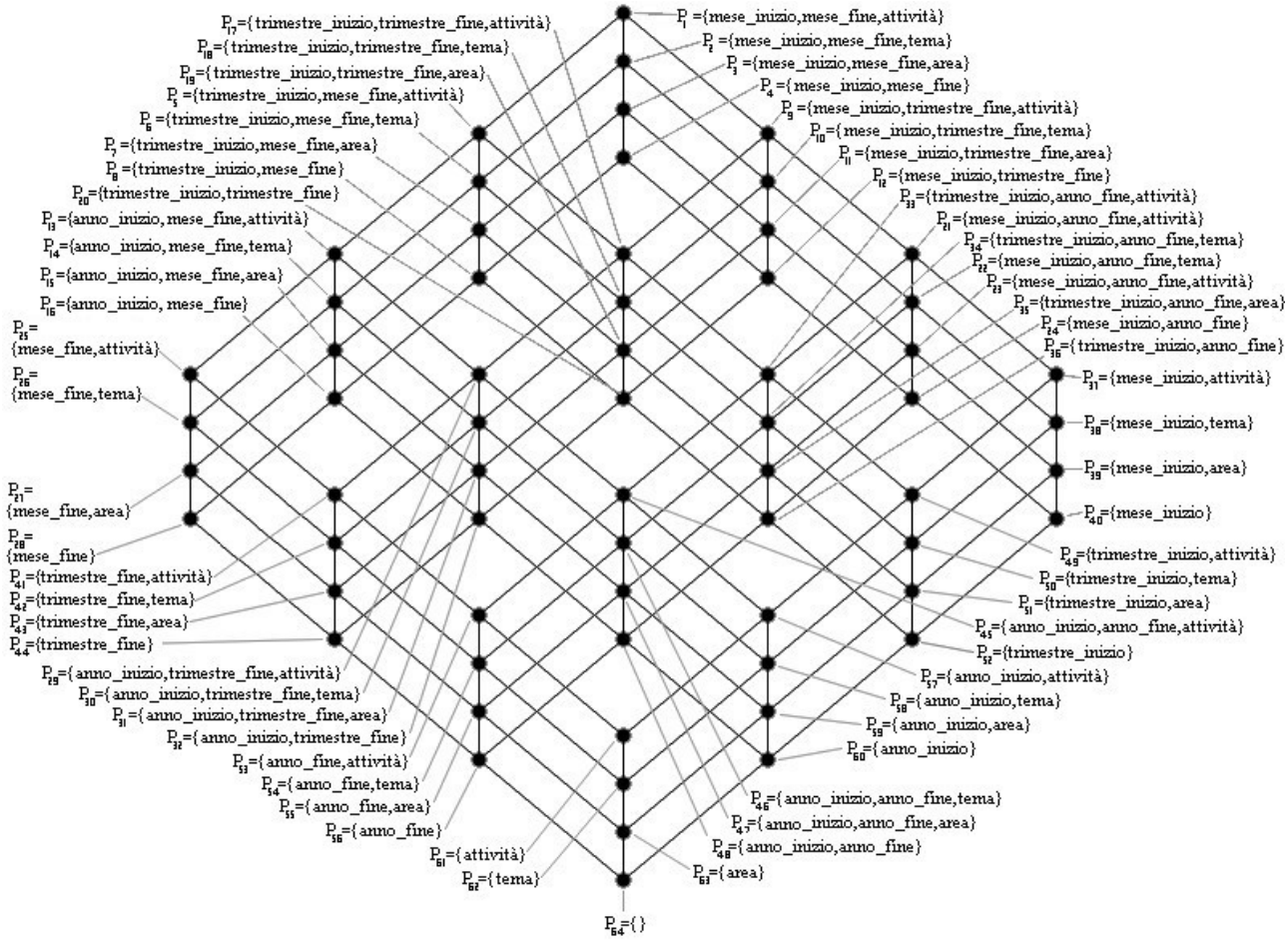


Figura 65 - Lattice del cubo preso in esame

Ricordiamo che nella notazione del lattice, una vista di nome Px={a, b, c} significa che la vista caratteristica ha come attributi caratteristici a, b, c. Nell'esempio in considerazione, la vista P1 ha come attributi caratteristici {mese_inizio, mese_fine, attività}, quindi rappresenta la fact table originale, senza pre-aggregazioni calcolate, mentre la vista P64 non ha attributi caratteristici e rappresenta le *misure* completamente aggregate (conterrà un solo record).

3.5.2 INTERROGAZIONI CARATTERISTICHE PER L'APPLICAZIONE

Consideriamo le seguenti interrogazioni come caratteristiche per l'applicazione.

```
Q1:   SELECT { [Tempo inizio].[MESE_INIZIO].Members ON COLUMNS,
      Crossjoin({[Tipo].[ATTIVITA].Members}}, {[Measures].Members}) ON ROWS
      FROM [Attivita]

Q2:   SELECT { [Tempo fine].[MESE_FINE].Members} ON COLUMNS,
      Crossjoin({[Tipo].[ATTIVITA].Members}}, {[Measures].Members}) ON ROWS
      FROM [Attivita]

Q3:   SELECT {[Tipo].[TEMA].Members } ON COLUMNS,
      {[Measures].Members} ON ROWS
      FROM [Attivita]
```

```
Q4:   SELECT Crossjoin({[Tempo inizio].[ANNO_INIZIO].Members},
      {[Measures].Members}) ON COLUMNS,
      {[Tipo].[AREA].Members } ON ROWS
      FROM [Attivita]

Q5:   SELECT Crossjoin({[Tempo fine].[ANNO_FINE].Members},
      {[Measures].Members}) ON COLUMNS,
      {[Tipo].[AREA].Members} ON ROWS
      FROM [Attivita]
```

Query	*Attributi caratteristici*	*Vista*
Q1	{mese_inizio,attività}	P37
Q2	{mese_fine,attività}	P25
Q3	{tema}	P62
Q4	{anno_inizio,area}	P59
Q5	{anno_fine,area}	P55

Tabella 17- Attributi caratteristici delle interrogazioni scelte

Riportiamo qui di seguito i risultati delle interrogazioni scelte come caratteristiche per l'applicazione calcolati su dati d'esempio.

					Tempo inizio										
					Tutte										
					2003									2004	
Tipo				Measures	T2			T3			T4			T1	
(All)	AREA	TEMA	ATTIVITA	Measure	4	5	6	7	8	9	10	11	12	1	2
Tutte	Acqua	SCARICHI ACQUE SUP.	ISTRUTTORIA DI IMPORTAZIONE ACQUA	N_ATTIVITA	7	8	9	13	8	4	7	10	7	5	
				DURATA_MEDIA (gg)	129.00	182.87	152.22	132.38	161.00	78.50	243.29	131.50	165.00	65.00	131
				DURATA_MAX (gg)	261	563	300	390	267	146	860	231	284	105	
	Aria	ARIA EMISSIONE IN ATMOSFERA	ISTRUTTORIA ARIA	N_ATTIVITA											
				DURATA_MEDIA (gg)											
				DURATA_MAX (gg)											
			ISTRUTTORIA DI IMPORTAZIONE ARIA	N_ATTIVITA	8	9	6	6	3	8	7	17	8	16	
				DURATA_MEDIA (gg)	150.00	150.00	150.00	150.00	150.00	150.00	150.00	150.00	150.00	150.00	150
				DURATA_MAX (gg)	150	150	150	150	150	150	150	150	150	150	

Figura 66 - Risultato dell'interrogazione Q1

					Tempo fine									
					Tutte									
					2005							2006		
Tipo				Measures	T2	T3			T4			T1		
(All)	AREA	TEMA	ATTIVITA	Measure	6	7	8	9	10	11	12	1	2	3
Tutte	Acqua	SCARICHI ACQUE SUP.	INVIO AUTORIZZAZIONE	N_ATTIVITA										1
				DURATA_MEDIA (gg)										11.00
				DURATA_MAX (gg)										11
			ISTRUTTORIA DI IMPORTAZIONE ACQUA	N_ATTIVITA		1	3	12	2	7	2	17	9	6
				DURATA_MEDIA (gg)		181.00	186.00	190.42	278.50	172.00	373.00	212.00	205.56	206.50
				DURATA_MAX (gg)		181	463	414	290	347	382	540	860	339
	Aria	ARIA EMISSIONE IN ATMOSFERA	COMUNICAZIONE AVVIO PROCEDIMENTO	N_ATTIVITA										2
				DURATA_MEDIA (gg)										2.50
				DURATA_MAX (gg)										4
			ISTRUTTORIA ARIA	N_ATTIVITA										3
				DURATA_MEDIA (gg)										150.00
				DURATA_MAX (gg)										150
			ISTRUTTORIA CONTROLLO	N_ATTIVITA										1
				DURATA_MEDIA (gg)										3.00
				DURATA_MAX (gg)										3
			ISTRUTTORIA DI IMPORTAZIONE ARIA	N_ATTIVITA	15	10		11	9	2	2	1	9	2
				DURATA_MEDIA (gg)	150.00	150.00		150.00	150.00	150.00	150.00	150.00	150.00	150.00
				DURATA_MAX (gg)	150	150		150	150	150	150	150	150	150
			MESSA A REGIME	N_ATTIVITA						1				
				DURATA_MEDIA (gg)						150.00				
				DURATA_MAX (gg)						150				
			MESSA IN ESERCIZIO	N_ATTIVITA					1					
				DURATA_MEDIA (gg)					150.00					
				DURATA_MAX (gg)					150					
			RICEZIONE ANALISI	N_ATTIVITA								1		
				DURATA_MEDIA (gg)								150.00		
				DURATA_MAX (gg)								150		
			RICEZIONE DOCUMENTO	N_ATTIVITA				1						3
				DURATA_MEDIA (gg)				150.00						150.00
				DURATA_MAX (gg)				150						150
			RICHIESTA BOLLI	N_ATTIVITA										1
				DURATA_MEDIA (gg)										12.00
				DURATA_MAX (gg)										12
			RICHIESTA INTEGRAZIONE	N_ATTIVITA										1
				DURATA_MEDIA (gg)										49.00
				DURATA_MAX (gg)										49
			RICHIESTA PARERE ARPAM	N_ATTIVITA						1				1
				DURATA_MEDIA (gg)						54.00				57.00
				DURATA_MAX (gg)						54				57
			RICHIESTA PARERE COMUNE	N_ATTIVITA									1	2
				DURATA_MEDIA (gg)									14.00	252.00
				DURATA_MAX (gg)									14	442
	Rifiuti	RIFIUTI ART. 27/28	CONFERENZA SERVIZI	N_ATTIVITA								1		1
				DURATA_MEDIA (gg)								38.00		7.00
				DURATA_MAX (gg)								38		7
		RIFIUTI ART.27	CONFERENZA SERVIZI	N_ATTIVITA										1
				DURATA_MEDIA (gg)										23.00
				DURATA_MAX (gg)										23
			ISTRUTTORIA 27	N_ATTIVITA										1
				DURATA_MEDIA (gg)										1.00
				DURATA_MAX (gg)										1
		RIFIUTI ART.28	RICHIESTA PARERE ARPA	N_ATTIVITA										2
				DURATA_MEDIA (gg)										47.00
				DURATA_MAX (gg)										58
		RIFIUTI ART.33	COMUNICAZIONE AVVIO PROCEDIMENTO	N_ATTIVITA										1
				DURATA_MEDIA (gg)										16.00
				DURATA_MAX (gg)										16
			RICHIESTA INTEGRAZIONE	N_ATTIVITA										3
				DURATA_MEDIA (gg)										27.00
				DURATA_MAX (gg)										43

Figura 67 - Risultato dell'interrogazione Q2

	Tipo						
	Tutte						
Measures	Acqua		Aria	Rifiuti			
Measure	+SCARICHI ACQUE SUP.	+SCARICHI ACQUE URBANE	+ARIA EMISSIONE IN ATMOSFERA	+RIFIUTI ART. 27/28	+RIFIUTI ART.27	+RIFIUTI ART.28	+RIFIUTI ART.33
N_ATTIVITA	387	5	916	3	3	111	197
DURATA_MEDIA (gg)	163.69	10.60	148.49	17.67	44.67	143.14	147.45
DURATA_MAX (gg)	860	20	442	38	110	169	150

Figura 68 - Risultato dell'interrogazione Q3

			Tipo		
Tempo inizio		Measures	Tutte		
(All)	ANNO_INIZIO	Measure	Acqua	Aria	Rifiuti
Tutte	1989	N_ATTIVITA		7	
		DURATA_MEDIA (gg)		150.00	
		DURATA_MAX (gg)		150	
	1990	N_ATTIVITA		5	
		DURATA_MEDIA (gg)		150.00	
		DURATA_MAX (gg)		150	
	1991	N_ATTIVITA		9	1
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1992	N_ATTIVITA		6	
		DURATA_MEDIA (gg)		150.00	
		DURATA_MAX (gg)		150	
	1993	N_ATTIVITA		17	1
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1994	N_ATTIVITA		8	
		DURATA_MEDIA (gg)		150.00	
		DURATA_MAX (gg)		150	
	1995	N_ATTIVITA		25	1
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1996	N_ATTIVITA		18	7
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1997	N_ATTIVITA		72	30
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1998	N_ATTIVITA		54	161
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1999	N_ATTIVITA		46	26
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	2000	N_ATTIVITA		39	25
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	2001	N_ATTIVITA	22	44	22
		DURATA_MEDIA (gg)	262.77	150.00	150.00
		DURATA_MAX (gg)	693	150	150
	2002	N_ATTIVITA	129	70	21
		DURATA_MEDIA (gg)	180.91	150.00	150.00
		DURATA_MAX (gg)	739	150	150
	2003	N_ATTIVITA	96	100	2
		DURATA_MEDIA (gg)	147.05	150.00	150.00
		DURATA_MAX (gg)	860	150	150
	2004	N_ATTIVITA	74	120	
		DURATA_MEDIA (gg)	158.77	151.97	
		DURATA_MAX (gg)	608	386	
	2005	N_ATTIVITA	46	240	3
		DURATA_MEDIA (gg)	165.37	149.35	105.67
		DURATA_MAX (gg)	379	442	169
	2006	N_ATTIVITA	24	36	14
		DURATA_MEDIA (gg)	27.42	109.36	18.29
		DURATA_MAX (gg)	150	150	58

			Tipo		
Tempo fine		Measures	Tutte		
(All)	ANNO_FINE	Measure	Acqua	Aria	Rifiuti
Tutte	1989	N_ATTIVITA		7	
		DURATA_MEDIA (gg)		150.00	
		DURATA_MAX (gg)		150	
	1990	N_ATTIVITA		5	
		DURATA_MEDIA (gg)		150.00	
		DURATA_MAX (gg)		150	
	1991	N_ATTIVITA		9	1
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1992	N_ATTIVITA		6	
		DURATA_MEDIA (gg)		150.00	
		DURATA_MAX (gg)		150	
	1993	N_ATTIVITA		17	1
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1994	N_ATTIVITA		8	
		DURATA_MEDIA (gg)		150.00	
		DURATA_MAX (gg)		150	
	1995	N_ATTIVITA		25	1
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1996	N_ATTIVITA		18	7
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1997	N_ATTIVITA		72	30
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1998	N_ATTIVITA		54	161
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	1999	N_ATTIVITA		46	26
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	2000	N_ATTIVITA		39	25
		DURATA_MEDIA (gg)		150.00	150.00
		DURATA_MAX (gg)		150	150
	2001	N_ATTIVITA	3	44	22
		DURATA_MEDIA (gg)	76.67	150.00	150.00
		DURATA_MAX (gg)	150	150	150
	2002	N_ATTIVITA	72	70	21
		DURATA_MEDIA (gg)	111.92	150.00	150.00
		DURATA_MAX (gg)	318	150	150
	2003	N_ATTIVITA	127	100	2
		DURATA_MEDIA (gg)	179.80	150.00	150.00
		DURATA_MAX (gg)	693	150	150
	2004	N_ATTIVITA	94	119	
		DURATA_MEDIA (gg)	161.89	150.00	
		DURATA_MAX (gg)	739	150	
	2005	N_ATTIVITA	45	240	
		DURATA_MEDIA (gg)	206.58	149.12	
		DURATA_MAX (gg)	463	386	
	2006	N_ATTIVITA	51	37	17
		DURATA_MEDIA (gg)	152.24	118.35	33.71
		DURATA_MAX (gg)	860	442	169

Figura 69 - Risultati delle interrogazioni Q4 e Q5

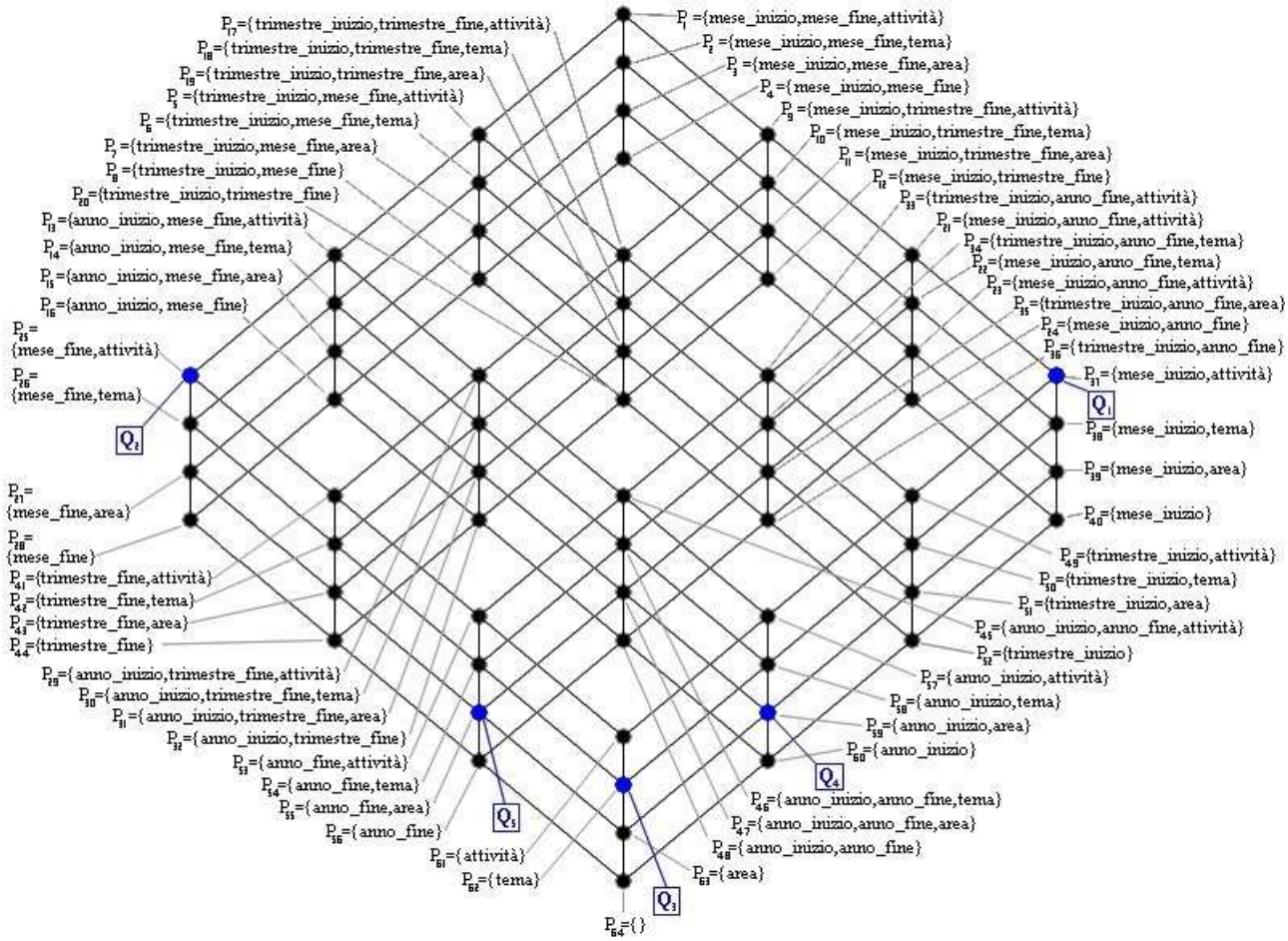


Figura 70 - Viste corrispondenti alle interrogazioni caratteristiche

3.5.3 DIPENDENZE FUNZIONALI DEL DATABASE MULTIDIMENSIONALE

Una volta identificato l'insieme delle viste corrispondenti alle interrogazioni caratteristiche per l'applicazione, l'unica altra informazione necessaria per applicare l'algoritmo di selezione delle viste sono le dipendenze funzionali tra gli attributi delle dimensioni coinvolte.

Queste ultime sono ricavabili dalla definizione del cubo, grazie all'ordine di definizione dei livelli. Riportiamo qui di seguito l'insieme delle dipendenze funzionali direttamente ricavabili dallo schema:

```
FDTCDB = { mese_inizio → trimestre_inizio,
           trimestre_inizio → anno_inizio,
           mese_fine → trimestre_fine,
           trimestre_fine → anno fine,
           attività → tema, tema → area}
```

A questo insieme vanno aggiunte tutte le dipendenze funzionali che si ottengono calcolando la chiusura transitiva dell'insieme precedente.

```
FDTCDB = { mese_inizio → trimestre_inizio,
           trimestre_inizio → anno_inizio,
           mese_fine → trimestre_fine,
```

```
            trimestre_fine → anno fine,
            attività → tema,
            tema → area,
            mese_inizio → anno_inizio,
            mese_fine → anno_fine,
            attività → area}
```

3.5.4 APPLICAZIONE DELL'ALGORITMO

A questo punto abbiamo tutte le informazioni necessarie per applicare l'algoritmo di selezione delle viste.

$L := \{P_{25}, P_{37}, P_{55}, P_{59}, P_{62}\}$
$lastViews := \{P_{25}, P_{37}, P_{55}, P_{59}, P_{62}\}$
$newViews := \{\emptyset\}$

Passo 1:

$v_i=P_{25}$ $v_j=P_{37}$
v_i+v_j={mese_inizio, mese_fine, attività}
$v_i \oplus v_j$={mese_inizio, mese_fine, attività}
{mese_inizio, mese_fine, attività}$=P_1 \notin L$ $newViews=\{P_1\}$

$v_i=P_{25}$ $v_j=P_{55}$
v_i+v_j={mese_fine, attività, anno_fine, area}
$v_i \oplus v_j$={mese_fine, attività}
{mese_fine, attività}$=P_{25} \in L$

$v_i=P_{25}$ $v_j=P_{59}$
v_i+v_j={mese_fine, attività, anno_inizio, area}
$v_i \oplus v_j$={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}$=P_{13} \notin L$ $newViews=\{P_1, P_{13}\}$

$v_i=P_{25}$ $v_j=P_{62}$
v_i+v_j={mese_fine, attività, tema}
$v_i \oplus v_j$={mese_fine, attività}
{mese_fine, attività}$=P_{25} \in L$

$v_i=P_{37}$ $v_j=P_{55}$
v_i+v_j={mese_inizio, attività, anno_fine, area}
$v_i \oplus v_j$={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}$=P_{21} \notin L$
$newViews=\{P_1, P_{13}, P_{21}\}$

$v_i=P_{37}$ $v_j=P_{59}$
v_i+v_j={mese_inizio, attività, anno_inizio, area}
$v_i \oplus v_j$={mese_inizio, attività}
{mese_inizio, attività}$=P_{37} \in L$

$v_i=P_{37}$ $v_j=P_{62}$
v_i+v_j={mese_inizio, attività, tema}
$v_i \oplus v_j$={mese_inizio, attività}
{mese_inizio, attività}$= P_{37} \in L$

$v_i=P_{55}$ $v_j=P_{59}$
v_i+v_j={anno_fine, anno_inizio, area}
$v_i \oplus v_j$={anno_fine, anno_inizio, area}

{anno_fine, anno_inizio, area}=$P_{45} \notin L$

newViews={$P_1, P_{13}, P_{21}, P_{47}$}

$v_i=P_{55}$ $v_j=P_{62}$

v_i+v_j={anno_fine, area, tema}

$v_i \oplus v_j$={anno_fine, tema}

{anno_fine, tema}=$P_{54} \notin L$

newViews={$P_1, P_{13}, P_{21}, P_{47}, P_{54}$}

$v_i=P_{59}$ $v_j=P_{62}$

v_i+v_j={anno_inizio, area, tema}

$v_i \oplus v_j$={anno_inizio, tema}

{anno_inizio, tema}=$P_{58} \notin L$

newViews={$P_1, P_{13}, P_{21}, P_{47}, P_{54}, P_{58}$}

L := {P_1, P_{13}, P_{21}, P_{25}, P_{37}, P_{47}, P_{54}, P_{55}, P_{58}, P_{59}, P_{62}}

lastViews := {P_1, P_{13}, P_{21}, P_{47}, P_{54}, P_{58}}

newViews := {∅}

Passo 2:

$v_i=P_1$ $v_j=P_{13}$

v_i+v_j={mese_inizio, mese_fine, attività, anno_inizio}

$v_i \oplus v_j$={mese_inizio, mese_fine, attività}

{mese_inizio, mese_fine, attività}=$P_1 \in L$

$v_i=P_1$ $v_j=P_{21}$

v_i+v_j={mese_inizio, mese_fine, attività, anno_fine}

$v_i \oplus v_j$={mese_inizio, mese_fine, attività}

{mese_inizio, mese_fine, attività}=$P_1 \in L$

$v_i=P_1$ $v_j=P_{25}$

v_i+v_j={mese_inizio, mese_fine, attività}

$v_i \oplus v_j$={mese_inizio, mese_fine, attività}

{mese_inizio, mese_fine, attività}=$P_1 \in L$

$v_i=P_1$ $v_j=P_{37}$

v_i+v_j={mese_inizio, mese_fine, attività}

$v_i \oplus v_j$={mese_inizio, mese_fine, attività}

{mese_inizio, mese_fine, attività}=$P_1 \in L$

$v_i=P_1$ $v_j=P_{47}$

v_i+v_j={mese_inizio, mese_fine, attività, anno_inizio, anno_fine, area}

$v_i \oplus v_j$={mese_inizio, mese_fine, attività}

{mese_inizio, mese_fine, attività}=$P_1 \in L$

$v_i=P_1$ $v_j=P_{54}$

v_i+v_j={mese_inizio, mese_fine, attività, anno_fine, tema}

$v_i \oplus v_j$={mese_inizio, mese_fine, attività}

{mese_inizio, mese_fine, attività}=$P_1 \in L$

$v_i=P_1$ $v_j=P_{55}$

v_i+v_j={mese_inizio, mese_fine, attività, anno_fine, area}

$v_i \oplus v_j$={mese_inizio, mese_fine, attività}

{mese_inizio, mese_fine, attività}=$P_1 \in L$

$v_i=P_1$ $v_j=P_{58}$

v_i+v_j={mese_inizio, mese_fine, attività, anno_inizio, tema}

$v_i \oplus v_j$={mese_inizio, mese_fine, attività}
{mese_inizio, mese_fine, attività}=$P_1 \in L$
$v_i=P_1$ $v_j=P_{59}$
v_i+v_j={mese_inizio, mese_fine, attività, anno_inizio, area}
$v_i \oplus v_j$={mese_inizio, mese_fine, attività}
{mese_inizio, mese_fine, attività}=$P_1 \in L$
$v_i=P_1$ $v_j=P_{62}$
v_i+v_j={mese_inizio, mese_fine, attività, tema}
$v_i \oplus v_j$={mese_inizio, mese_fine, attività}
{mese_inizio, mese_fine, attività}=$P_1 \in L$
$v_i=P_{13}$ $v_j=P_{21}$
v_i+v_j={anno_inizio, mese_fine, attività, mese_inizio,
anno_fine}
$v_i \oplus v_j$={mese_inizio, mese_fine, attività}
{mese_inizio, mese_fine, attività}=$P_1 \in L$
$v_i=P_{13}$ $v_j=P_{25}$
v_i+v_j={anno_inizio, mese_fine, attività}
$v_i \oplus v_j$={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}=$P_{13} \in L$
$v_i=P_{13}$ $v_j=P_{37}$
v_i+v_j={anno_inizio, mese_fine, attività, mese_inizio}
$v_i \oplus v_j$={mese_inizio, mese_fine, attività}
{mese_inizio, mese_fine, attività}=$P_1 \in L$
$v_i=P_{13}$ $v_j=P_{47}$
v_i+v_j={anno_inizio, mese_fine, attività, anno_fine, area}
$v_i \oplus v_j$={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}=$P_{13} \in L$
$v_i=P_{13}$ $v_j=P_{54}$
v_i+v_j={anno_inizio, mese_fine, attività, anno_fine, tema}
$v_i \oplus v_j$={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}=$P_{13} \in L$
$v_i=P_{13}$ $v_j=P_{55}$
v_i+v_j={anno_inizio, mese_fine, attività, anno_fine, area}
$v_i \oplus v_j$={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}=$P_{13} \in L$
$v_i=P_{13}$ $v_j=P_{58}$
v_i+v_j={anno_inizio, mese_fine, attività, tema}
$v_i \oplus v_j$={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}=$P_{13} \in L$
$v_i=P_{13}$ $v_j=P_{59}$
v_i+v_j={anno_inizio, mese_fine, attività, area}
$v_i \oplus v_j$={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}=$P_{13} \in L$
$v_i=P_{13}$ $v_j=P_{62}$
v_i+v_j={anno_inizio, mese_fine, attività, tema}
$v_i \oplus v_j$={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}=$P_{13} \in L$
$v_i=P_{21}$ $v_j=P_{25}$

v_i+v_j={mese_inizio, anno_fine, attività, mese_fine}
$v_i \oplus v_j$={mese_inizio, mese_fine, attività}
{mese_inizio, mese_fine, attività}=$P_1 \in L$

$v_i=P_{21}$ $v_j=P_{37}$
v_i+v_j={mese_inizio, anno_fine, attività}
$v_i \oplus v_j$={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}=$P_{21} \in L$

$v_i=P_{21}$ $v_j=P_{47}$
v_i+v_j={mese_inizio, anno_fine, attività, anno_inizio, area}
$v_i \oplus v_j$={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}=$P_{21} \in L$

$v_i=P_{21}$ $v_j=P_{54}$
v_i+v_j={mese_inizio, anno_fine, attività, tema}
$v_i \oplus v_j$={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}=$P_{21} \in L$

$v_i=P_{21}$ $v_j=P_{55}$
v_i+v_j={mese_inizio, anno_fine, attività, area}
$v_i \oplus v_j$={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}=$P_{21} \in L$

$v_i=P_{21}$ $v_j=P_{58}$
v_i+v_j={mese_inizio, anno_fine, attività, anno_inizio, tema}
$v_i \oplus v_j$={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}=$P_{21} \in L$

$v_i=P_{21}$ $v_j=P_{59}$
v_i+v_j={mese_inizio, anno_fine, attività, anno_inizio, area}
$v_i \oplus v_j$={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}=$P_{21} \in L$

$v_i=P_{21}$ $v_j=P_{62}$
v_i+v_j={mese_inizio, anno_fine, attività, tema}
$v_i \oplus v_j$={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}=$P_{21} \in L$

$v_i=P_{47}$ $v_j=P_{25}$
v_i+v_j={anno_inizio, anno_fine, area, mese_fine, attività}
$v_i \oplus v_j$={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}=$P_{13} \in L$

$v_i=P_{47}$ $v_j=P_{37}$
v_i+v_j={anno_inizio, anno_fine, area, mese_inizio, attività}
$v_i \oplus v_j$={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}=$P_{21} \in L$

$v_i=P_{47}$ $v_j=P_{54}$
v_i+v_j={anno_inizio, anno_fine, area, tema}
$v_i \oplus v_j$={anno_inizio, anno_fine, tema}
{anno_inizio, anno_fine, tema}=$P_{46} \notin L$ newViews={P_{46}}

$v_i=P_{47}$ $v_j=P_{55}$
v_i+v_j={anno_inizio, anno_fine, area}
$v_i \oplus v_j$={anno_inizio, anno_fine, area}
{anno_inizio, anno_fine, area}=$P_{47} \in L$

$v_i=P_{47}$ $v_j=P_{58}$

v_i+v_j={anno_inizio, anno_fine, area, tema}
$v_i \oplus v_j$={anno_inizio, anno_fine, tema}
{anno_inizio, anno_fine, tema}=$P_{46} \notin L$ newViews={P_{46}}
$v_i=P_{47}$ $v_j=P_{59}$
v_i+v_j={anno_inizio, anno_fine, area}
$v_i \oplus v_j$={anno_inizio, anno_fine, area}
{anno_inizio, anno_fine, area}=$P_{47} \in L$
$v_i=P_{47}$ $v_j=P_{62}$
v_i+v_j={anno_inizio, anno_fine, area, tema}
$v_i \oplus v_j$={anno_inizio, anno_fine, tema}
{anno_inizio, anno_fine, tema}=$P_{46} \notin L$ newViews={P_{46}}
$v_i=P_{54}$ $v_j=P_{25}$
v_i+v_j={anno_fine, tema, mese_fine, attività}
$v_i \oplus v_j$={mese_fine, attività}
{mese_fine, attività}=$P_{25} \in L$
$v_i=P_{54}$ $v_j=P_{37}$
v_i+v_j={anno_fine, tema, mese_inizio, attività}
$v_i \oplus v_j$={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}=$P_{21} \in L$
$v_i=P_{54}$ $v_j=P_{55}$
v_i+v_j={anno_fine, tema, area}
$v_i \oplus v_j$={anno_fine, tema}
{anno_fine, tema}=$P_{54} \in L$
$v_i=P_{54}$ $v_j=P_{58}$
v_i+v_j={anno_fine, tema, anno_inizio}
$v_i \oplus v_j$={anno_inizio, anno_fine, tema}
{anno_inizio, anno_fine, tema}=$P_{46} \notin L$ newViews={P_{46}}
$v_i=P_{54}$ $v_j=P_{59}$
v_i+v_j={anno_fine, tema, anno_inizio, area}
$v_i \oplus v_j$={anno_inizio, anno_fine, tema}
{anno_inizio, anno_fine, tema}=$P_{46} \notin L$ newViews={P_{46}}
$v_i=P_{54}$ $v_j=P_{62}$
v_i+v_j={anno_fine, tema}
$v_i \oplus v_j$={anno_fine, tema}
{anno_fine, tema}=$P_{54} \in L$
$v_i=P_{58}$ $v_j=P_{25}$
v_i+v_j={anno_inizio, tema, mese_fine, attività}
$v_i \oplus v_j$={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}=$P_{13} \in L$
$v_i=P_{58}$ $v_j=P_{37}$
v_i+v_j={anno_inizio, tema, mese_inizio, attività}
$v_i \oplus v_j$={mese_inizio, attività}
{mese_inizio, attività}=$P_{37} \in L$
$v_i=P_{58}$ $v_j=P_{55}$
v_i+v_j={anno_inizio, tema, anno_fine, area}
$v_i \oplus v_j$={anno_inizio, anno_fine, tema}
{anno_inizio, anno_fine, tema}=$P_{46} \notin L$ newViews={P_{46}}
$v_i=P_{58}$ $v_j=P_{59}$

v_i+v_j={anno_inizio, tema, area}
v_i⊕v_j={anno_inizio, tema}
{anno_inizio, tema}=P_{58}∈L
v_i=P_{58} v_j=P_{62}
v_i+v_j={ anno_inizio, tema}
v_i⊕v_j={anno_inizio, tema}
{anno_inizio, tema}=P_{58}∈L

L := {P_1, P_{13}, P_{21}, P_{25}, P_{37}, P_{46}, P_{47}, P_{54}, P_{55}, P_{58}, P_{59}, P_{62}}
lastViews := {P_{46}}
newViews := {∅}

Passo 3:
v_i=P_{46} v_j=P_1
v_i+v_j={anno_inizio, anno_fine, tema, mese_inizio, mese_fine, attività}
v_i⊕v_j={mese_inizio, mese_fine, attività}
{mese_inizio, mese_fine, attività}=P_1∈L
v_i=P_{46} v_j=P_{13}
v_i+v_j={anno_inizio, anno_fine, tema, mese_fine, attività}
v_i⊕v_j={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}=P_{13}∈L
v_i=P_{46} v_j=P_{21}
v_i+v_j={anno_inizio, anno_fine, tema, mese_inizio, attività}
v_i⊕v_j={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}=P_{21}∈L
v_i=P_{46} v_j=P_{25}
v_i+v_j={anno_inizio, anno_fine, tema, mese_fine, attività}
v_i⊕v_j={anno_inizio, mese_fine, attività}
{anno_inizio, mese_fine, attività}=P_{13}∈L
v_i=P_{46} v_j=P_{37}
v_i+v_j={anno_inizio, anno_fine, tema, mese_inizio, attività}
v_i⊕v_j={mese_inizio, anno_fine, attività}
{mese_inizio, anno_fine, attività}=P_{21}∈L
v_i=P_{46} v_j=P_{47}
v_i+v_j={anno_inizio, anno_fine, tema, area}
v_i⊕v_j={anno_inizio, anno_fine, tema}
{anno_inizio, anno_fine, tema}=P_{46}∈L
v_i=P_{46} v_j=P_{54}
v_i+v_j={anno_inizio, anno_fine, tema}
v_i⊕v_j={anno_inizio, anno_fine, tema}
{anno_inizio, anno_fine, tema}=P_{46}∈L
v_i=P_{46} v_j=P_{55}
v_i+v_j={anno_inizio, anno_fine, tema, area}
v_i⊕v_j={anno_inizio, anno_fine, tema}
{anno_inizio, anno_fine, tema}=P_{46}∈L
v_i=P_{46} v_j=P_{58}
v_i+v_j={anno_inizio, anno_fine, tema}
v_i⊕v_j={anno_inizio, anno_fine, tema}

```
        {anno_inizio, anno_fine, tema}=P46∈L
    vi=P46 vj=P59
        vi+vj={anno_inizio, anno_fine, tema, area}
        vi⊕vj={anno_inizio, anno_fine, tema}
        {anno_inizio, anno_fine, tema}=P46∈L
    vi=P46 vj=P62
        vi+vj={anno_inizio, anno_fine, tema}
        vi⊕vj={anno_inizio, anno_fine, tema}
        {anno_inizio, anno_fine, tema}=P46∈L

L := {P1, P13, P21, P25, P37, P46, P47, P54, P55, P58, P59, P62}
lastViews := {∅}
newViews := {∅}

Ritorna {P1, P13, P21, P25, P37, P46, P47, P54, P55, P58, P59, P62}
```

Riportiamo qui di seguito il lattice che rappresenta il cubo con evidenziate le viste selezionate dall'algoritmo. Nell'immagine sono state evidenziate in blu le viste corrispondenti alle interrogazioni caratteristiche dell'applicazione, in verde le viste selezionate dal primo passo dell'algoritmo e in rosso le viste selezionate dal secondo passo dell'algoritmo.

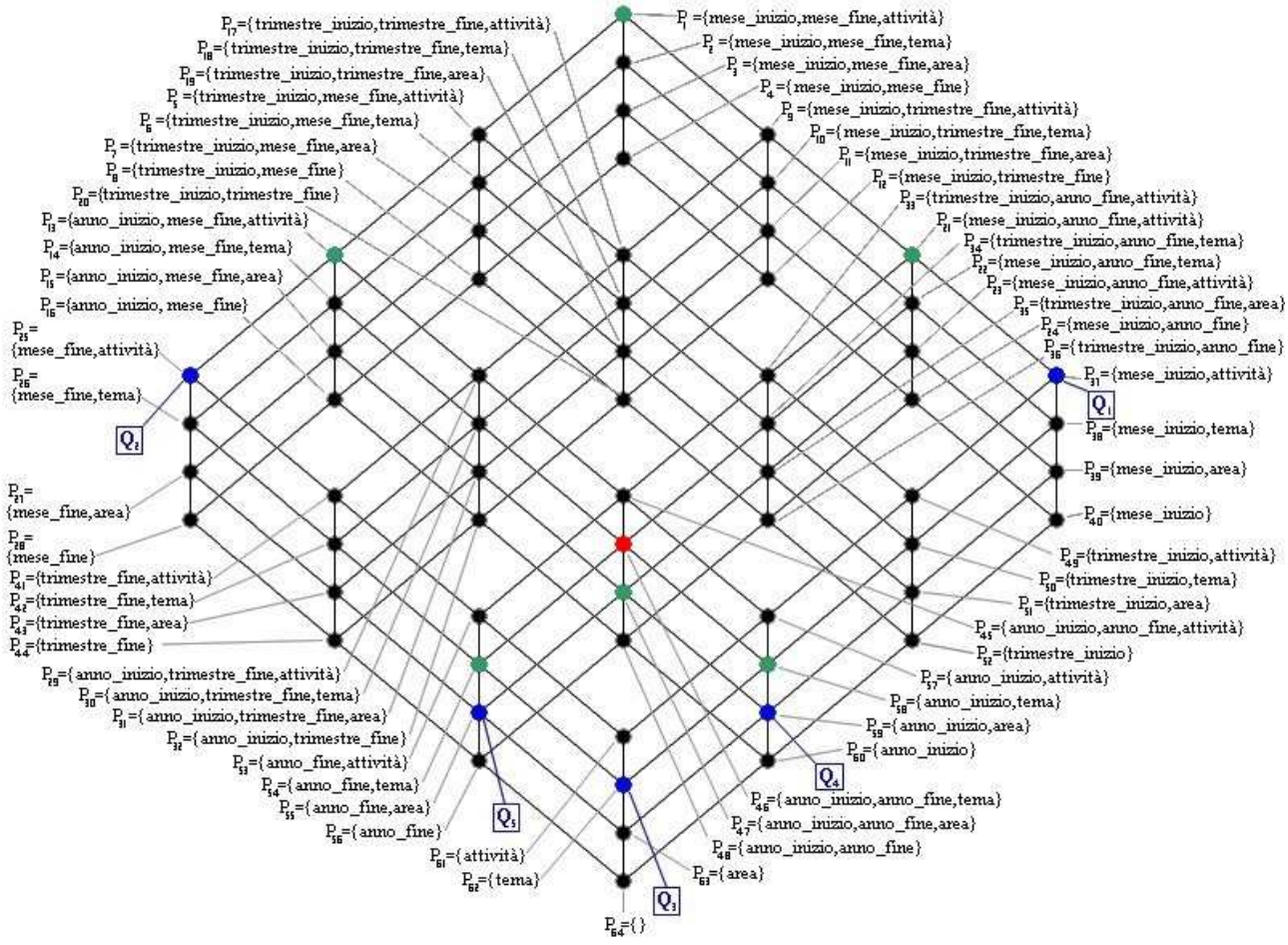


Figura 71 - Viste selezionate per la materializzazione dall'algoritmo

Dall'insieme delle viste selezionate dall'algoritmo possiamo eliminare la vista P1 che corrisponde ai dati che sono già contenuti nella fact table e quindi non necessita di essere materializzata. Consideriamo l'insieme delle viste selezionate:

$\{P_{13}, P_{21}, P_{25}, P_{37}, P_{46}, P_{47}, P_{54}, P_{55}, P_{58}, P_{59}, P_{62}\}$

Per ridurre la cardinalità dell'insieme, conoscendo la cardinalità della fact table e delle tabelle dimensionali, si può utilizzare il criterio euristico illustrato nel paragrafo 2.7.3. Nel caso in esame, trattandosi di dimensioni che aumentano la cardinalità molto lentamente rispetto alla fact table (le dimensioni tempo aumentano di una tupla al mese, mentre la dimensione tipo rimane pressoché costante), vanno considerati con particolare attenzione ai risultati di questa tecnica. Infatti, se oggi potrebbe non essere conveniente materializzare una vista perché contiene pochi record rispetto alle tabelle dimensionali, potrebbe non esserlo in futuro vista la crescita più veloce della fact table.

4. CONCLUSIONI E LAVORO FUTURO

Durante il periodo di 6 mesi svolto presso l'azienda "Quix srl" di Soliera (MO) si sono approfondite diverse tematiche che riguardano i sistemi di data warehouse, partendo da uno studio dei prodotti disponibili sul mercato per arrivare a un'analisi approfondita di alcuni strumenti open-source e allo sviluppo di un prototipo.

In particolare durante la tesi si sono approfondite tematiche che riguardano la definizione degli schemi preposti a contenere i dati da interrogare, il linguaggio per estrarre i dati in formato multidimensionale e le tecniche di alimentazione richieste da alcuni campi di applicazione. Si sono inoltre analizzate tecniche per migliorare le prestazioni del sistema quali un algoritmo di selezione delle viste candidate a essere materializzate e l'applicazione di tecniche di frammentazione.

Il lavoro svolto durante il periodo di stage ha portato alla realizzazione di un'applicazione Web, basata sul framework Jakarta Struts, e che utilizza il server OLAP open-source Mondrian per offrire servizi di reporting basati su interrogazioni multidimensionali. All'interno dell'applicazione è possibile definire tre tipi diversi di elementi ("tabelle", "grafici" e "report") e viene data la possibilità di combinare più elementi all'interno della stessa schermata ("pagine").

All'interno dell'applicazione sono state definite due diverse procedure di alimentazione create per due diversi campi applicativi. La prima, più classica, alimenta il cubo a partire dai dati operazionali di un'applicazione esistente, mentre il secondo richiede un complesso sistema di regole che permette di estrarre dai file di log di accesso di Tomcat le informazioni sulle azioni compiute dagli utenti.

Le note maggiormente positive dell'applicazione realizzata riguardano:

- le prestazioni dell'architettura che, in presenza di grandi moli di dati e a patto che questi ultimi siano adeguatamente strutturati e indicizzati, risponde in tempi più che accettabili;
- il vantaggio che comporta il sistema di cache messo a disposizione dal server OLAP Mondrian che permette di ridurre sensibilmente i tempi di attesa in risposta per le interrogazioni successive alla prima;
- l'elevata configurabilità di Mondrian che permette di avere un controllo molto preciso sia sul lato prestazionale con i parametri della cache, sia sul recupero dei dati (tramite i file di catalogo permette di definire anche strutture non standard) sia per quel che riguarda l'estensibilità, assicurata dalle numerose interfacce implementabili per ampliare le funzionalità di base;
- la creazione di cruscotti che possono racchiudere più elementi multidimensionali tra loro indipendenti che risultano molto gradevoli dal punto di vista grafico e che riescono a racchiudere un gran numero di informazioni mantenendo tempi di attesa accettabili;
- la possibilità fornita dalla funzione di "Modeller" di costruire e memorizzare interrogazioni in modo visuale a partire da una interrogazione di default e procedendo per modifica.

Durante lo sviluppo dell'applicazione si sono anche evidenziati alcuni aspetti negativi legati al tool JPivot di visualizzazione dei componenti e alla difficoltà di esprimere interrogazioni complesse.

In particolare:

- Il componente di visualizzazione delle interrogazioni utilizza un'architettura basata sul framework di WCF (Web Component Framework) che è una implementazione di JSF (Java Server Faces). Questa architettura è completamente diversa dal framework Struts utilizzato per il resto dell'applicazione. Oltre ai problemi di convivenza dei due framework, bisogna considerare che WCF crea qualche difficoltà nella gestione dei suoi componenti salvati all'interno della sessione utente.
- La visualizzazione dei componenti grafici del tool JPivot viene realizzata partendo da un codice XML generato da ogni singolo componente grafico che viene trasformato attraverso un foglio di stile XSL in codice HTML. Questa scelta, che può apparire interessante, in realtà pone dei limiti alla personalizzazione della visualizzazione dei componenti. Infatti, per esempio, non è possibile cambiare l'ordine di visualizzazione di alcune proprietà all'interno dello stesso componente senza modificare il codice del componente stesso che genera il codice XML.
- Il componente grafico JPivot visualizza solo interrogazioni con al più 2 dimensioni. Questo limite, che in prima analisi appare di natura grafica (lo schermo ha 2 dimensioni), in realtà è dovuto alla mancanza di una funzione che, dato un risultato n-dimensionale lo visualizzi a due dimensioni. Abbiamo visto, infatti, che il linguaggio MDX è in grado di esprimere interrogazioni che possono coinvolgere fino a 128 assi diversi. Il limite del componente grafico impone di effettuare operazioni di "CrossJoin" delle dimensioni in modo da potere utilizzare due soli assi al motore OLAP prima di eseguire l'interrogazione. Questa soluzione, oltre all'onere del carico delle operazioni di "CrossJoin" sul server OLAP, piuttosto che a livello di componente finale, ha due importanti conseguenze:
    - la necessità, in caso di interrogazioni a 3 o più dimensioni, di richiamare in causa il server OLAP in caso di cambiamento di asse di qualche dimensione che non sia la completa inversione degli assi;
    - un carico computazionale maggiore per il server OLAP che si trova a operare con pochi assi ma con una cardinalità molto elevata (il prodotto cartesiano delle singole cardinalità delle dimensioni coinvolte).
- Un altro problema legato al componente grafico riguarda la sezione di "Modeller" delle interrogazioni che viene abilitata per i soli utenti con diritti di amministrazione per un comportamento non ideale del componente quando si trova a modificare una interrogazione. Infatti, quando un utente esegue una operazione di "drill-down" su un membro, invece di aggiungere solamente il predicato "membro.children" all'interrogazione, il componete tende a modificare l'interrogazione aggiungendo l'elenco esplicito dei membri. Questo comportamento, che a breve termine non ha conseguenze, può, nel caso di interrogazioni salvate, comportare la non visualizzazione di nuovi elementi della *gerarchia* che siano stati inseriti in seguito. Per questo motivo, l'amministratore dovrà, prima di salvare l'interrogazione, considerare se una dimensione è statica o dinamica e eventualmente modificare il codice MDX di conseguenza.
- Scrivere interrogazioni MDX per un report di media complessità diventa abbastanza complicato e richiede una profonda conoscenza del linguaggio e un po' di esperienza nella scrittura di report, rende il linguaggio adatto solo per utenti esperti.

- La mancanza in MDX di alcune funzioni molto utili in SQL, quali ad esempio now(), motivata dalla diversa struttura dei dati, richiede una maggiore manutenzione delle interrogazioni che coinvolgono periodi temporali basati sul giorno attuale.

Alcune altre considerazioni riguardano l'applicabilità delle tecniche descritte nel capitolo 2 ad alcuni contesti applicativi. Prendiamo, ad esempio, in considerazione l'algoritmo di selezione delle viste da materializzare. Questo algoritmo presuppone di sapere a priori quali sono le interrogazioni caratteristiche per l'applicazione. L'individuazione di questo insieme e i risultati prodotti dall'algoritmo vanno considerati con attenzione nel caso (come quello del prototipo sviluppato) in cui le informazioni visualizzate non siano statiche ma navigabili dagli utenti.

Da notare, però, come l'applicazione di algoritmi dello stesso tipo di quello presentato all'interno della tesi siano citati nelle roadmap di diversi sistemi OLAP (come, ad esempio, in quella di Mondrian) tra le funzioni da implementare nelle future versioni dei sistemi, a testimonianza dell'importanza dell'individuazione di un insieme di viste da materializzare.

Funzioni che sono state lasciate a futuri sviluppi dell'applicazione sono:

- diritti di visibilità dei singoli elementi in base a utenti o gruppi di utenti;
- modifica dei cataloghi di definizione delle sorgenti dati via Web;
- definizione di procedure che periodicamente memorizzino automaticamente i risultati dell'esecuzione di alcune interrogazioni.

5. BREVE GLOSSARIO

BI Acronimo di Business Intelligence. Un settore dell'information technology che include applicazioni di reporting e di analisi.

BPEL Acronimo di Business Process Execution Language. Un sistema standard usato per orchestrare workflow attraveso servizi multipli. I web service sono chiamati BPEL4WS o WSBPEL.

CMS Arconimo di Content Management System.

Data Mart (DM) Insieme di dati organizzati per supportare specifiche esigenze di un gruppo di utenti (es. dipartimento) che viene alimentato da processi di estrazione e trasformazione utilizzando i dati del DWH.

Data Warehouse (DWH) Insieme di dati tematici, integrati, temporali, permanenti finalizzato al supporto dei processi decisionali. Viene alimentato attraverso processi di alimentazione e trasformazione utilizzando dati contenuti nei database operazionali ed eventualmente dati esterni.

DCL Data Control Language permette di gestire gli utenti e i permessi

DDL Data Definition Language permette di creare e cancellare database o di modificarne la struttura

Dimensioni Insieme di attributi, generalmente di tipo testo, che definiscono e danno un significato alle *misure* (es. Tempo, Prodotto, Cliente, Mercato, Promozione, Contratti, Fornitore, Transazioni, ecc.)

DML Data Manipulation Language permette di inserire, cancellare e modificare i dati

Drill Funzionalità OLAP che consente di visualizzare dati a diversi livelli di dettaglio, "navigando" attraverso le gerarchie. Si parla di drill-up quando l'operazione provoca un'aggregazione delle informazioni (es. da quantità venduta per modello a quantità venduta per marca), drill-down quando succede il contrario.

DSS Sistemi per il supporto alle decisioni (Decision Support Systems)

ETL Il termine ETL (Extract, Transform, Load) si riferisce al processo di caricamento dei dati in un data warehouse.

Gerarchie Naturale correlazione tra attributi appartenenti alla stessa dimensione, dipendente dall'organizzazione e dalle specifiche esigenze applicative (ad esempio, la gerarchia Tempo è formata dai livelli Anno, Trimestre, Mese e Giorno).

HOLAP Utilizza tabelle relazionali per memorizzare i dati e le tabelle multidimensionali per le aggregazioni "speculative".

HTML Acronimo di Hyper Text Markup Language. Un linguaggio usato per creare pagine web.

IBM WebSphere™ Un portale proprietario di IBM che supporta lo standard JSR-168.

J2EE Acronimo di Java 2 Enterprise Edition. Uno standard indipendente dalla piattaforma per sviluppare software multi-livello.

Java™	Un linguaggio indipendente dalla piattaforma per scrivere software.
JBoss™	Una compagnia che fornisce tecnologie open source.
JBoss™ Portal	Il portale open source di JBoss che supporta lo standard JSR-168.
Jetspeed	Il portale open source di Apache foundation che supporta lo standard JSR-168.
JSR-168	Le specifiche dello standard per realizzare interfacce utente da integrare in un portale.
JSR-170	Content Repository for Java Technology API. Uno standard per implementare Content Management Systems.
JSR-94	Java Rule Engine API. Le specifiche dello standard per realizzare motori per regole basati su java.
Kerberos	Un protocollo standard di rete per l'autenticazione.
KPI	Acronimo di Key Performance Indicator. Una metrica per le performance.
LDAP	Acronimo di Lightweight Directory Access Protocol. Un protocollo standard per accedere alle proprietà di una risorsa.
MDX	MDX è un linguaggio d'interrogazione utilizzato per definire, utilizzare e recuperare dati da oggetti multidimensionali.
Metadata	Informazioni utilizzate per descrivere la struttura e il contenuto dei dati.
Misure o Fatti	Valori, generalmente numerici, utilizzati dagli utenti per la misurazione del loro business (es. Quantità venduta, Ricavo, Costo, Presenza/Assenza di un evento, ecc.).
Modello multidimensionale	Organizzazione delle informazioni orientata alle funzioni di reporting e analisi realizzata per consentire un efficace utilizzo degli strumenti automatici di query, reporting ed analisi.
MOLAP	è la tipologia più utilizzata e ci si riferisce ad essa comunemente con il termine OLAP. Utilizza un database di riepilogo che ha uno specifico motore per l'analisi multidimensionale e crea le "dimensioni" con un misto di dettaglio ed aggregazioni.
MS Excel™ Add-In	Un programma Windows che esegue all'interno di MS Excel
Notification Tray	Comunemente chiamato anche "System Tray" è l'area della task bar di windows che visualizza i messaggi di allerta e i servizi.
OLAP	Letteralmente On Line Analytical Processing, è l'insieme delle funzionalità utente che consentono di rendere "dinamica" la visualizzazione delle informazioni attraverso alcune funzionalità utente denominate drill, slice and dice, pivoting
OracleAS Portal™	Portale proprietario di Oracle che supporta lo standard JSR-168
Portal	Una applicazione web-based per l'aggregazione di contenuti provenienti da diversi sistemi in una o più pagine multifunzione.
Portlet	Un componente di un portale che fornisce contenuti per il portale e provenienti da altri sistemi.

Query Caratteristiche	Insieme delle interrogazioni che più frequentemente vengono sottoposte a un sistema e che, quindi, modellano il comportamento della maggior parte degli utenti.
ROLAP	Lavora direttamente con database relazionali; i dati e le tabelle delle dimensioni sono memorizzati come tabelle relazionali e nuove tabelle sono create per memorizzare le informazioni di aggregazione.
Slice and dice o pivoting	Funzionalità OLAP che consente di ristrutturare le informazioni in modo da renderne più efficace la visualizzazione: creazione di master-detail e rotazione degli assi delle rappresentazioni a matrice.
SQL	Structured Query Language è un linguaggio creato per l'accesso a informazioni memorizzate nei database.
Staging Area	E' un'area in cui i dati provenienti dai sistemi operazionali sono temporaneamente memorizzati per essere sottoposti a processi di "pulizia", trasformazione e aggregazione allo scopo di alimentare correttamente il Data Warehouse
Weblogic Portal™	Un portale proprietario della BEA Corporation che supporta lo standard JSR-168.
WSRP	Acronimo di Web Service for Remote Portlet. Una specific ache permette a un portale di chiamare un'altro portale per ricevere le informazioni da visualizzare.
XML	Acronimo di eXtensible Markup Language. Un linguaggio standard per creare documenti che contengono dati leggibili dalle macchine, comunemente usato per trasferire e trasformare dati.
XPDL	Acronimo di XML Process Definition Language. Un linguaggio standard per descrivere dei processi in XML.
XSL	Acronimo di XML Stylesheet Language. Uno standard per la trasformazione di un documento XML in un altro formato.

6. BIBLIOGRAFIA RAGIONATA

DATABASE

[1] P. Ciaccia, D. Maio: "Lezioni di BASI DI DATI", Progetto Leonardo - ed. Esculapio – Bologna

[2] http://www.techonthenet.com/oracle/functions/ come guida alle funzioni utilizzabili all'interno di query SQL nei DBMS Oracle.

[3] http://dev.mysql.com/doc/refman/5.1/en/index.html come guida alle funzioni utilizzabili all'interno di query SQL in MySQL 5.1.

DATAWAREHOUSE

[4] "Materialized View Selection in a Multidimensional Database" Elena Baralis, Stefano Paraboschi, Ernest Teniente per l'algoritmo di selezione delle viste candidate a essere materializzate.

[5] http://www.olap.it/ per la parte di descrizione generale degli strumenti OLAP.

[6] Dimitri Theodoratos, Mokrane Bouzeghoub "A General Framework for the View Selection Problem for Data Warehouse Design and Evolution" Third ACM International Workshop on Data Warehousing and OLAP, November 10, 2000, Washington, DC

[7] "Data Warehouse from Architecture to Implementation" di Barry Devlin, Addison Wesley, 1997

[8] http://jpivot.sourceforge.net/wcf/index.html per la guida a WCF (Web Component Framework) e per il capitolo dedicato all'argomento.

[9] http://jpivot.sourceforge.net/wcf/tags/wcf-tags-en.html#form per la guida ai tag di WCF.

[10] http://jpivot.sourceforge.net/ per la guida a JPivot e per il capitolo dedicato all'argomento.

[11] http://jpivot.sourceforge.net/tags/jpivot-tags-en.html per la guida ai tags di JPivot

[12] http://mondrian.sourceforge.net/ per la guida al OLAP server Mondrian e per i capitoli dedicati alla sua architettura, alla sua configurazione, alle sue prestazioni e alla scrittura dei file di definizione degli schemi e delle tabelle degli aggregati.

[13] Mondrian Schema Eclipse Plugin (http://jpivot.sourceforge.net/mondrian-schema/index.html) come strumento per la scrittura dei file di catalogo di Mondrian attraverso un'interfaccia grafica all'interno di Eclipse.

[14] http://www.pentaho.org/ come guida alla piattaforma di Business Intelligence Pentaho e per il capitolo dedicato all'argomento.

[15] "Data Warehouse - Teoria e pratica della progettazione" di: Matteo Golfarelli, Stefano Rizzi, Mcgraw-hill

MDX

[16] "MDX Solutions With Microsoft SQL Server Analysis Services 2005 and Hyperion Essbase" di: George Spofford , Sivakumar Harinath, Christopher Webb, Dylan Hai Huang - John Wiley and Sons Ltd.

[17] "Multidimensional Expressions (MDX) Reference" Microsoft MSDN (http://msdn2.microsoft.com/en-us/library/ms145506.aspx) come guida alle funzioni utilizzabili all'interno di interrogazioni MDX.

[18] "MDX Data Definition Statements" Microsoft MSDN (http://msdn2.microsoft.com/en-us/library/ms144926.aspx) come guida al linguaggio DDL di MDX

ALTRI

[19] "Sviluppare applicazioni J2EE con Jakarta Struts" di Alfredo Larotonda (http://www.mokabyte.it/2004/01/jstruts-1.htm) per il capitolo dedicato al framework Jakarta Struts.

[20] http://tomcat.apache.org/tomcat-5.0-doc/config/valve.html come guida alla configurazione dei log di accesso di Tomcat.