UNIVERSITÀ DEGLI STUDI DI NAPOLI "FEDERICO II"

FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Riscrittura di interrogazioni con viste in sistemi per la gestione di Data Warehouse

Tesi di Laurea Sperimentale in Basi di Dati e Sistemi Informativi

Relatore:
Ch.mo Prof. Adriano Peron

Correlatore:
Ch.mo Prof. Albero Finzi

Candidato:
Francesco Saverio Ferrara
Matr. N97/11

ANNO ACCADEMICO 2009-2010

Alla mia famiglia

Indice

Elenco delle figure

Capitolo 1

Introduzione

Le moderne tecnologie informatiche hanno permesso di collezionare molti più dati di quanto la mente umana possa assimilarne. La quantità di dati digitalizzati cresce vertiginosamente; da qui nasce il bisogno di nuove tecnologie che permettano la loro elaborazione in modo da estrarre da essi informazioni utili.
In particolare per le aziende, lo studio dei dati digitali permette di ottenere vantaggi competitivi sul mercato. In questo contesto si collocano le applicazioni della Business Intelligence, dedicate al supporto decisionale.

Il supporto informativo utilizzato per i processi della Business Intelligence è il *Data Warehouse*. Un sistema per la sua gestione differisce dai sistemi utilizzati per i tradizionali database operazionali per l'utilizzo di tecnologie e strutture dati specializzate.
Un esempio di tale sistema è SADAS, un *sistema per la gestione di basi di dati* (*DBMS*) specializzato nell'interrogazione di archivi statici di elevate dimensioni, frutto della ricerca svolta dall'azienda *Advanced Systems* dagli anni '80. Basato sul modello dei dati relazionale, SADAS utilizza un metodo di archiviazione dei dati colonnare (*column-based*) e altre strutture di indicizzazione specializzate. Queste caratteristiche permettono di avere, in ambienti di Data Warehousing, performance migliori rispetto ai DBMS tradizionali.

Una tecnica comunemente utilizzata dagli analisti dei dati è di creare viste materializzate allo scopo di velocizzare il calcolo della risposta a interrogazioni (*query*) frequentemente poste alla base di dati.
Tale tecnica è senza dubbio più efficace se affiancata da un sistema di ottimizzazione delle interrogazioni in grado di utilizzare automaticamente le viste materializzate. Tale sistema deve affrontare il problema di rispondere a query utilizzando le viste, cioè quello di trovare metodi efficaci per elaborare la risposta a una query utilizzando un insieme di viste materializzate in precedenza definite sulla base di dati, piuttosto che accedere alle relazioni del database. Un approccio utilizzato per risolvere tale problema riguarda la *riscrittura dell'interrogazione con viste materializzate* (*view-based query rewriting*).

Nel presente lavoro si introdurrà un modulo dell'ottimizzatore di SADAS, per la riscrittura di interrogazioni con viste materializzate. Questo componente sarà integrato nella struttura dell'ottimizzatore, come una nuova fase "di riscrittura", posta tra la fase di *semplificazione* e quella di *trasformazione logica*.
L'intera parte implementativa sarà fatta utilizzando linguaggio C++, in modo da sfruttare il codice preesistente, e agevolare l'integrazione.

Per la riscrittura si utilizzerà un innovativo approccio basato su regole, nel quale non si punta a un algoritmo di riscrittura generico, ma ad avere una "regola" per ogni tipologia di query che si vuole riscrivere.
Il componente sarà affiancato da un modulo software per la selezione automatica delle viste. Quest'ultimo, per generare script SQL concernenti la creazione di viste materializzate, si servirà dello stesso insieme di regole definite per il modulo di riscrittura.
Avendo in comune la parte di codice riguardante le regole, si creerà uno stretto legame tra i due moduli, che permetterà di individuare le interrogazioni costose da eseguire in termini di tempo, e di creare per esse delle viste materializzate utilizzabili dal sistema di riscrittura.

Capitolo 2

I Data Warehouse nella Business Intelligence

Un *Data Warehouse* è un database contenente una grande quantità di informazioni, utilizzata per supportare le decisioni imprenditoriali delle organizzazioni.
Un sistema per la gestione di Data Warehouse differisce dai sistemi utilizzati per i tradizionali database operazionali per l'utilizzo di tecnologie e strutture dati specializzate.

In questo capitolo si introduce il Data Warehouse come supporto informativo per i processi della Business Intelligence, e si presentano le principali tecniche implementative utilizzate.

2.1 La Business Intelligence

La *Business Intelligence* (BI) è stata definita come il processo di "trasformazione di dati e informazioni in conoscenza" (si veda[1]). Produce riflessioni atte a consentire ai responsabili aziendali di operare decisioni consapevoli e informate, oltre che a stabilire, modificare e trasformare le strategie e i processi di business in modo tale da trarne vantaggi competitivi, migliorare le performance operative e la profittabilità, e, più in generale, raggiungere gli obiettivi prefissati.

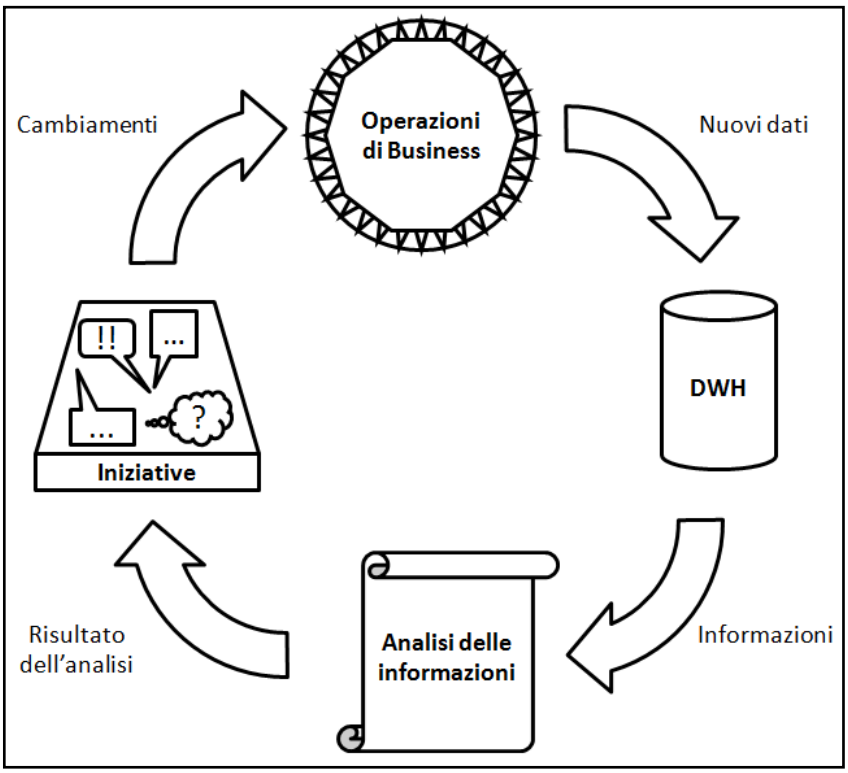

Figura 1: Ciclo di vita della Business Intelligence

Usando una moderna e sintetica definizione, riportata in [2], si può definire la Business Intelligence come l'insieme dei *processi* per *raccogliere* e *analizzare* le *informazioni* riguardanti il business aziendale.

È interessante analizzare in dettaglio le parole chiave di questa definizione:

- Processi: si riferisce a un particolare insieme di processi aziendali come, ad esempio, la produzione o la gestione contabile; questo comporta il coinvolgimento di tecnologie, strumenti, e figure professionali.
- Raccogliere: una parte di questi processi ha come obiettivo quello di reperire le informazioni riguardanti il business aziendale, le quali spesso si trovano in diversi sistemi informativi e sono difformi per struttura e tecnologia.
- Analizzare: un'altra parte dei processi ha l'obiettivo di permettere, attraverso strumenti dedicati, l'analisi delle informazioni raccolte.

- Informazioni: la definizione fa riferimento a informazioni aziendali, e non a dati. Queste due termini, spesso usati in modo intercambiabile, fanno qui riferimento a differenti concetti: l'informazione è un qualcosa di immediatamente fruibile per il ragionamento, mentre il dato ha bisogno di essere contestualizzato, completato, e trasformato, per poi diventare informazione. Un lampante esempio è quello di una fattura emessa per un cliente. Questa, isolata dal contesto delle altre fatture emesse, non porta ad alcun ragionamento, e rappresenta un dato; la stessa fattura, vista insieme con le altre, può essere utile all'azienda per compiere ragionamenti di valutazione o confronto.

La principale finalità dei sistemi di BI, come schematizzato nella Figura 1, è quella di essere un supporto alle decisioni e alle iniziative aziendali. Per questo motivo spesso si fa riferimento ad essi come *sistemi di supporto alle decisioni* (*Decision Support Systems*). Questi sistemi forniscono vari tipi di supporto per le decisioni:

- Report dei dati: è il livello di supporto più basso, in cui si generano dei semplici report informativi per i manager, senza effettuare particolari elaborazioni sulle informazioni presentate.
- Analisi dei dati: effettua delle analisi statistiche sui dati al fine di scoprire delle proprietà potenzialmente interessanti.
- Data Mining: utilizza tecniche sofisticate al fine di scoprire nuova informazione dai dati dell'organizzazione.

I sistemi di supporto alle decisioni si appoggiano dunque a basi di dati, o comunque a basi di conoscenza, per la gestione dei dati. Nella quasi totalità dei casi, i dati sono gestiti da un *Data Base Management System* (*DBMS*), cioè un sistema software che fornisce gli strumenti per definire la base di dati, le strutture per memorizzare i dati, e metodi per accedere in modo agevole alle informazioni memorizzate. In questo ambiente i DBMS sono specializzati nella gestione di Data Warehouse: il sistema informativo alla base dei processi della Business Intelligence (Figura 2).

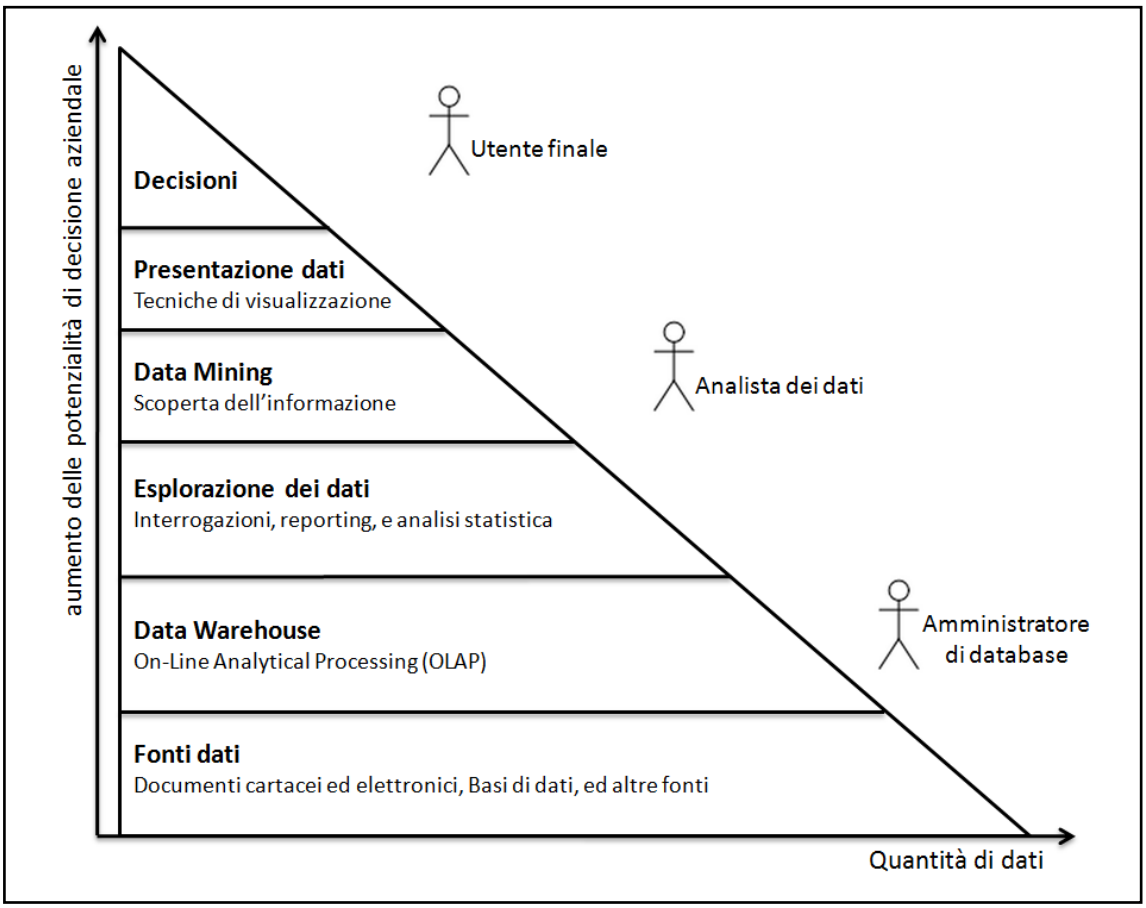

Figura 2: Data Warehouse nella Business Intelligence

2.2 Data Warehouse

Un *Data Warehouse* (*DHW*), letteralmente *magazzino dati*, è un database utile a conservare grandi quantità di informazioni, a volte suddivise in unità logiche più piccole chiamate *Data Mart*. I DWH sono alimentati da altre fonti informative: ad esempio in un'azienda queste fonti possono essere i dati dei sistemi *ERP* (*Enterprise Resource Planning*) o di altri sistemi gestionali. Esso contiene le informazioni storiche di un'organizzazione, immagazzinate in strutture dati che favoriscono analisi efficienti e reporting dei dati.

Bill Inmon fu il primo a parlare esplicitamente di Data Warehouse, e lo definì come: "*Una raccolta di dati integrata, orientata al soggetto, che varia nel tempo e non volatile di supporto ai processi decisionali*" ([3]).
In questa definizione vi sono delle parole chiave interessanti da esaminare:

- Raccolta di dati integrata: un Data Warehouse contiene dati diversi, ma che sono tra loro correlati, poiché provengono da più sistemi transazionali e altre fonti dati.
- Orientata al soggetto: i dati sono archiviati nel DWH in modo che possano essere facilmente letti o elaborati dagli utenti. A differenza dei database operazionali, l'obiettivo non è più quello di minimizzare la ridondanza mediante la normalizzazione, ma quello di immagazzinare dati che abbiano una struttura in grado di favorire la produzione di informazioni. Per raggiungere quest'obiettivo spesso si crea un Data Mart per ogni categoria di utenti.
- Variante nel tempo: in un database operazionale i dati corrispondono sempre a una situazione costantemente aggiornata, e spesso non forniscono un quadro storico della loro evoluzione. Contrariamente i dati archiviati all'interno di un DWH hanno un orizzonte temporale molto più esteso. Essendo uno strumento di supporto alle decisioni si pone l'accento sulla natura storica dei dati, e spesso le informazioni contenute al suo interno sono aggiornate a una data antecedente a quella in cui l'utente interroga il sistema.
- Non volatile: i dati memorizzati in un DWH sono stabili: si utilizzano principalmente per eseguire interrogazioni, e difficilmente sono soggetti a modifiche.

La creazione di un sistema di Data Warehousing prevede una fase di integrazione dei dati, svolta dagli strumenti ETL (acronimo di Extract-Transform-Load), i quali estraggono, trasformano, e caricano i dati nel Data Warehouse centrale oppure in un Data Mart. La trasformazione dei dati è fondamentale per avere un sistema coerente.

In seguito il DWH creato è utilizzato per presentare all'utente finale le analisi effettuate sui dati. Quest'ultimo dispone spesso di un insieme di strumenti utili a facilitare la produzione di informazioni, come interfacce grafiche intuitive per la presentazione dei risultati, generatori automatici di interrogazioni e report, e sistemi di analisi dati più complessi.

Come mostrato in Figura 3, una possibile architettura per un sistema di Data Warehousing prevede quattro livelli:

- Livello delle sorgenti. Il sistema utilizza fonti di dati eterogenei che tipicamente fanno parte dell'ambiente di produzione dell'organizzazione.
- Livello dell'alimentazione. I dati delle sorgenti sono estratti, ripuliti per eliminare le inconsistenze, integrati secondo uno schema comune, e successivamente caricati nel Data Warehouse. Come già detto questo compito è lasciato agli strumenti ETL.
- Livello del Warehouse. Il DWH è il contenitore che raccoglie tutte le informazioni. Esso può essere direttamente consultato dalle applicazioni di analisi dati, o può essere usato per alimentare i Data Mart, i quali ne costituiscono una replica parziale.
- Livello di analisi. Le applicazioni analitiche permettono la consultazione efficiente dei dati integrati.

Nell'architettura proposta i Data Mart sono alimentati direttamente dal Data Warehouse primario: in questo caso si parla di Data Mart *dipendenti*. In altre architetture non vi è un DWH primario e i Data Mart sono alimentati direttamente dalle sorgenti: in questo caso sono detti *indipendenti*.

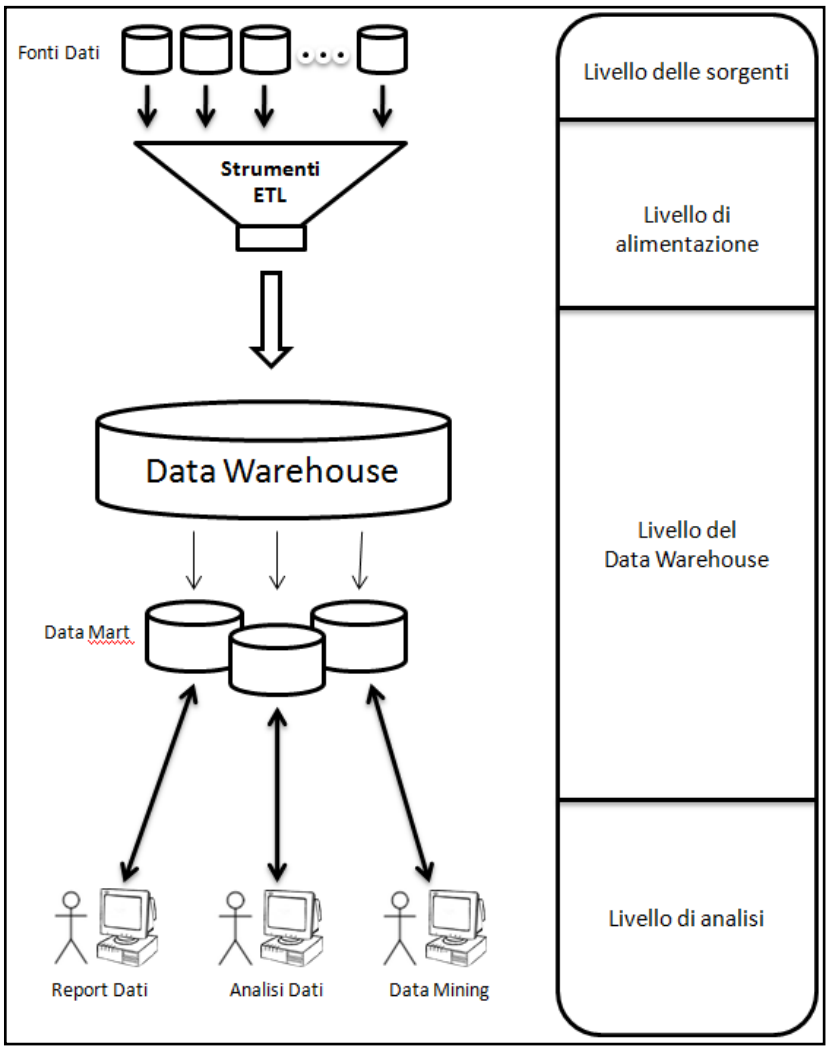

Figura 3: Possibile architettura per un sistema di Data Warehousing

2.2.1 On Line Analytical Processing

Un DWH si distingue da una tradizione base di dati operazionale per varie ragioni. La prima riguarda le prestazioni del sistema. Ad uno strumento di supporto alle decisioni sono spesso poste interrogazioni complesse, e dunque un buon sistema dovrebbe sfruttare la natura statica delle informazioni prevedendo speciali strutture dati in modo da ottimizzare le performance. Una seconda ragione è che spesso le organizzazioni

hanno l'esigenza di gestire una grande quantità di dati, la quale è difficilmente trattabile con strumenti classici, come database operazionali, ottimizzati per l'interrogazione basata su transazione di piccole quantità di dati. Inoltre, come già detto nella precedente sezione, i sistemi di supporto alle decisioni devono mantenere i dati storici provenienti da più fonti, e integrarli in maniera coerente.

Per queste ragioni si differenziano i sistemi classici che utilizzano database operazionali (OLTP, On-Line Transaction Processing) dai sistemi di supporto alle decisioni che utilizzano i Data Warehouse (OLAP, On-Line Analytical Processing).

Con il termine OLAP, coniato da E. F. Codd in[4], ci si riferisce all'insieme di sistemi e tecniche software per l'analisi interattiva e efficiente di grandi quantità di dati. In seguito al lavoro di E. F. Codd, N. Pendse caratterizzò i sistemi OLAP in [5] introducendo il modello FASMI (Fast Analysis of Shared Multidimensional Information):

- Fast: per permettere a un analista dei dati di lavorare in modo efficiente, un sistema OLAP deve calcolare il risultato molto velocemente (in genere pochi secondi);
- Analytical: fornire funzioni raffinate per analizzare i dati, minimizzando lo sforzo richiesto nella programmazione del sistema;
- Shared: un sistema OLAP è condiviso da più categorie di utenti. Esso deve dunque fornire meccanismi di sicurezza e di controllo degli accessi adeguati;
- Multidimensional: il sistema deve fornire una visione multidimensionale dei dati, perché, per i manager di aziende e organizzazioni, è il modo più logico di organizzare le informazioni;
- Information: i sistemi OLAP memorizzano le informazioni provenienti da più fonti dati; un'importante caratteristica è la quantità delle informazioni gestibile dal sistema.

Il punto più importante è senza dubbio la visione multidimensionale dei dati. I manager sono interessati nell'analizzare dei *fatti* riguardanti l'attività dell'organizzazione: l'esempio classico è quello delle vendite. Ogni fatto è descritto da un insieme di attributi numerici detti *misure*: nell'ambito delle vendite, le misure possono essere "quantità", "incasso", e "costo". Per analizzare i dati in modo agevole, è conveniente introdurre dimensioni, le quali danno la visione multidimensionale prima citata. Per il fatto "vendita", le dimensioni possono essere "prodotto", "giorno", e "negozio".

La Figura 4 mostra la visione multidimensionale per l'esempio delle vendite. La struttura risultante è un *ipercubo*, e ogni sua cella corrisponde a una particolare istanza del fatto (*evento*). È usato il termine ipercubo poiché possono esserci più di tre dimensioni.

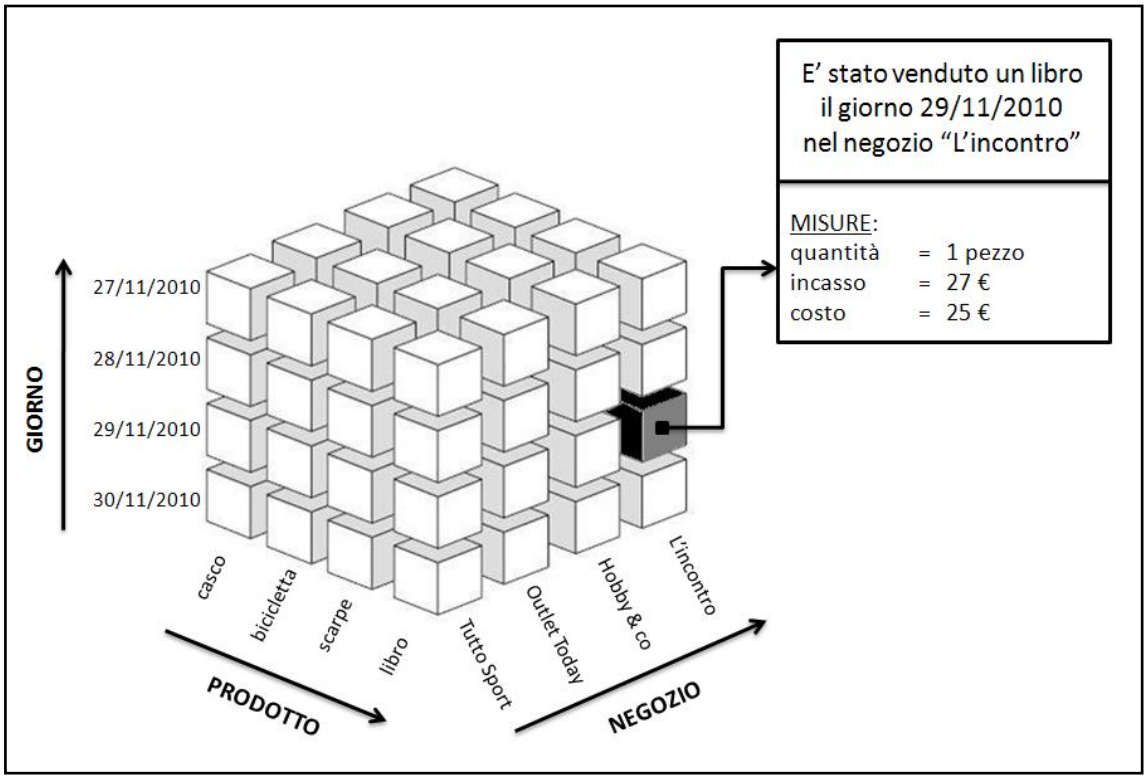

Figura 4: Cubo per l'esempio delle vendite

La struttura a *ipercubo*, detta in gergo semplicemente *cubo*, permette di gestire le interrogazioni OLAP in modo intuitivo. Un analista dei dati può ad esempio chiedere l'operazione di *slicing*, che consiste nel fissare il valore di una dimensione ed esaminare così solamente un "fetta" del cubo. Se si fissa il valore di più di una dimensione, allora si ha l'operazione chiamata *dice*. Un'altra operazione comune è quella di *rollup*, che consiste nell'aggregare i dati delle celle per ottenere delle informazioni di sintesi. L'operazione di *drill-down* è opposta a quella di rollup: permette di espandere i dati di sintesi, in dati a granularità più fine.

Per la modellazione concettuale di fatti, misure, e dimensioni, sono stati proposti vari formalismi nel mondo accademico. Il Dimensional Fact Model (DFM) introdotto in [2] è riuscito a valicare i confini dell'ambito accademico e a essere utilizzato in diverse realtà aziendali.
Lo schema in Figura 5, detto *schema di fatto*, usa questo formalismo per rappresentare l'esempio delle vendite. La tabella centrale "VENDITA" rappresenta il fatto, e al suo interno sono elencante le misure "quantità", "incasso", e "costo". Ogni istanza del fatto, ovvero ogni vendita, è univocamente individuata fissando il valore delle dimensioni "giorno", "prodotto", e "negozio". Queste sono rappresentate nello schema dai cerchi bianchi direttamente connessi con la tabella del fatto, mentre gli altri cerchi individuano delle gerarchie associate alle dimensioni. Le gerarchie sono usate come coordinate di analisi; ad esempio la gerarchia giorno-mese-anno permette di analizzare l'andamento delle vendite giornaliero, mensile, e annuale.

Alcuni server OLAP utilizzano strutture dati multidimensionali specializzate per implementare l'ipercubo. Si fa riferimento a queste implementazione con il termine *MOLAP* (*Multidimensional OLAP*). Un noto prodotto di questo tipo è PALO, presentato dettagliatamente in [6].

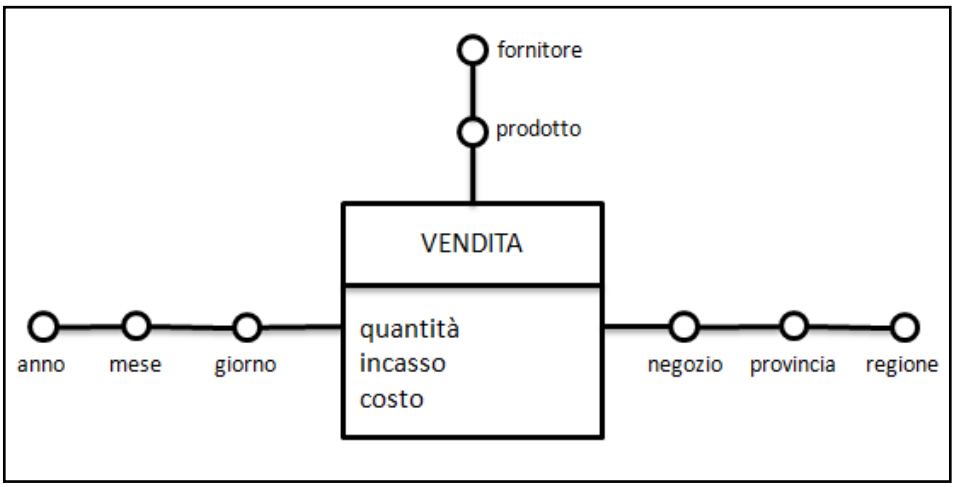

Figura 5: Schema concettuale per l'esempio delle vendite

Altre implementazioni OLAP invece fanno uso della tradizionale tecnologia relazionale, e modellano l'ipercubo utilizzando uno speciale schema relazionale detto *schema a stella*. Tali implementazioni sono dette *ROLAP* (*Relational OLAP*), e sono quelle analizzate nel presente lavoro di tesi. Per un'introduzione alla tecnologia e al modello dei dati relazionale si può consultare [7] e [8].
Le soluzioni *HOLAP* (*Hybrid OLAP*) sono invece un ibrido: utilizzano sia le classiche tabelle relazionali, sia le strutture multidimensionali.

2.2.2 Relational OLAP

Questo lavoro riguarda in particolare i sistemi ROLAP, nei quali la modellazione del cubo è fatta con lo *schema a stella*. Tale schema possiede una tabella principale, chiamata *tabella dei fatti* (in inglese "*fact table*"), connessa con una o più tabelle delle *dimensioni*.

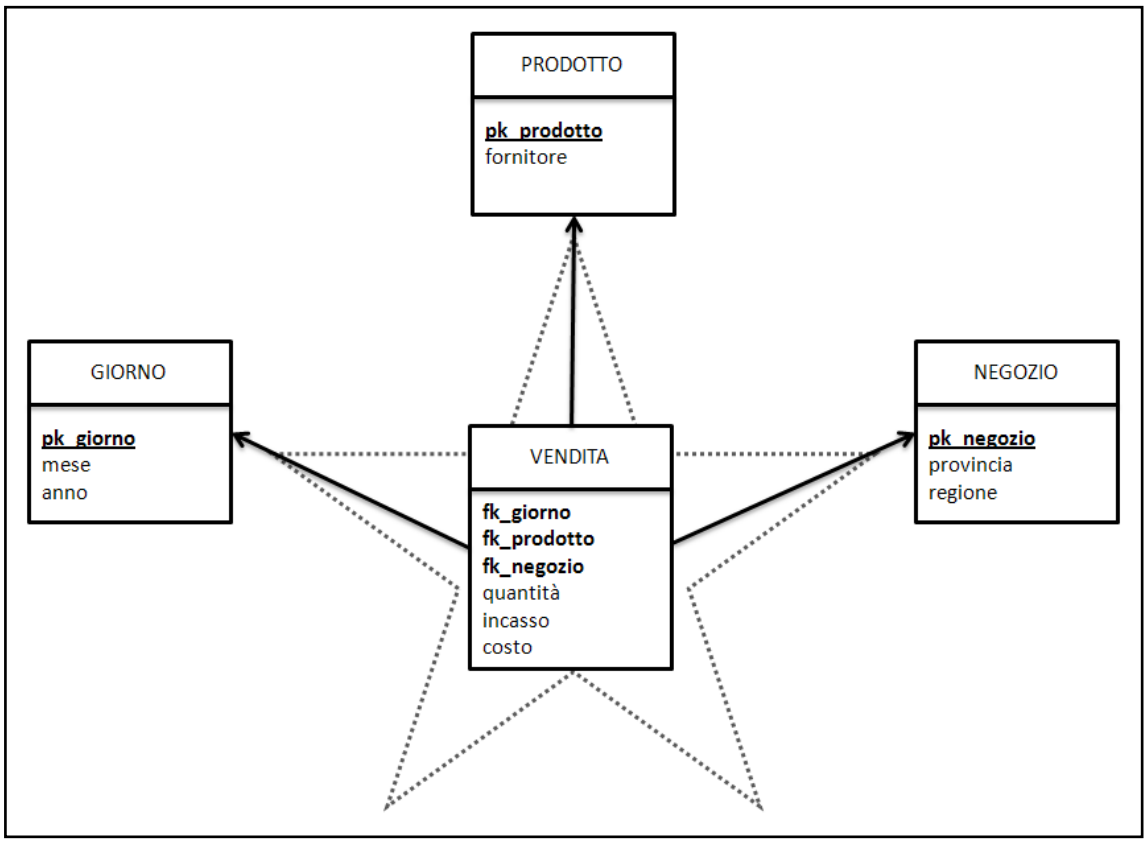

Figura 6: Schema a stella per l'esempio delle vendite

Nella Figura 6 è raffigurato lo schema a stella per l'esempio delle vendite. La tabella dei fatti è VENDITA, e le tabelle concernenti le dimensioni sono GIORNO, PRODOTTO, e NEGOZIO.

Ogni vendita è rappresentata da una tupla della tabella dei fatti, ed è univocamente determinata dalla terna <*fk_giorno, fk_prodotto, fk_negozio*>. Ogni tabella dimensionale possiede una chiave primaria in relazione uno-a-molti con la relativa chiave esterna contenuta nella tabella dei fatti.

Una variante dello schema a stella è lo *schema a fiocco di neve*. In questo particolare tipo di schema una o più dimensioni sono normalizzate in modo da diminuire la quantità di dati ridondanti. Ad esempio si può normalizzare la dimensione "negozio" dello

schema di Figura 6 in modo da normalizzare i dati riguardanti la provincia concernente il negozio: lo schema risultante è mostrato in Figura 7.

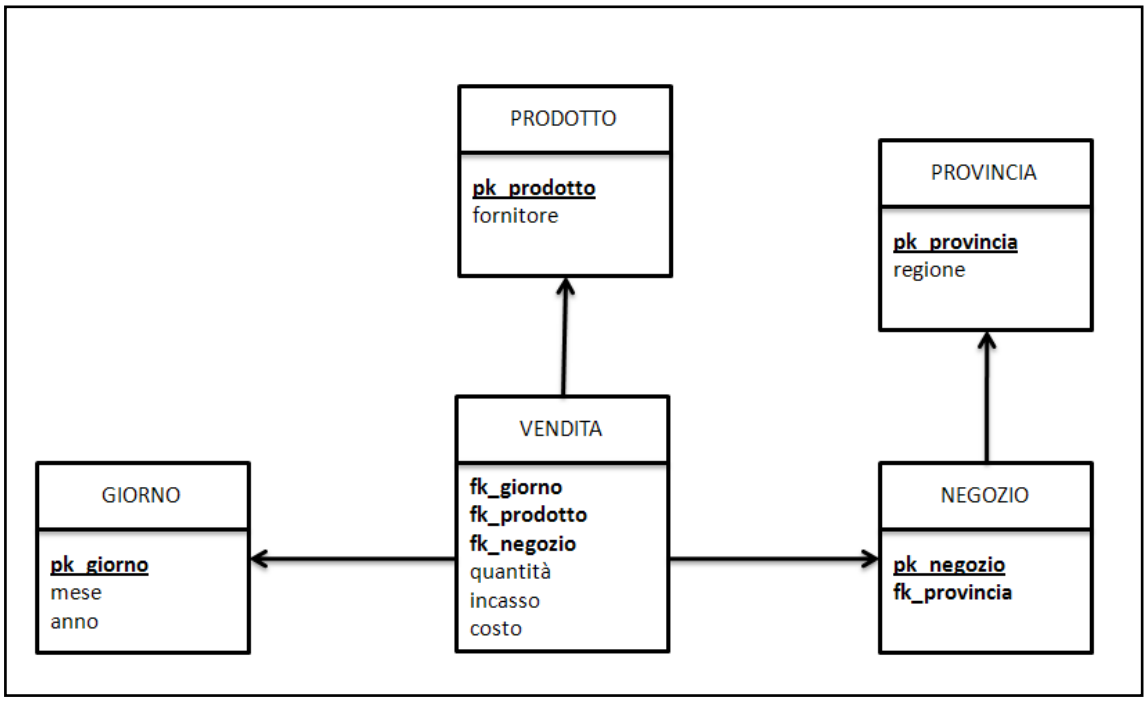

Figura 7: Schema a fiocco di neve per l'esempio delle vendite

In alcuni casi vi è la presenza di più tabelle dei fatti che possono condividere una o più dimensioni. Questa seconda variante dello schema a stella è chiamata "*constellation schema*".

2.3 Tecniche di implementazione per Data Warehouse

Attualmente i principali sistemi per il Data Warehousing sono di tipo ROLAP, poiché i fornitori possono avvantaggiarsi della conoscenza acquisita negli ultimi anni sulla tecnologia relazionale. Alcune tecniche di implementazione di questi sistemi sono quelle utilizzate nei DBMS relazionali classici, ampiamente discusse in [9]; altre tecniche invece sono proprie dell'ambiente DWH.

La prima differenza rispetto ai sistemi operazionali classici è dovuta alla staticità dei dati. I dati storici raramente sono sottoposti a modifiche: ciò comporta una maggiore semplicità di progettazione del DBMS. In tale situazione non si fronteggiano le possibili anomalie dovute agli aggiornamenti e tanto meno si ricorre a strumenti complessi per gestire l'integrità referenziale o per bloccare record a cui possono accedere altri utenti in fase di aggiornamento.

Le altre differenze sono dovute alla natura delle interrogazioni poste alla base di dati. Per ottimizzare le operazioni di *slicing*, *dice*, *rollup*, e *drill-down* in precedenza citate, si utilizzano particolari tipi di indici oltre al classico indice invertito.
Un primo tipo di indice è chiamato *bitmap index*. Supponiamo che i record siano memorizzati maniera consecutiva, e siano numerati in modo sequenziale dal numero *identificativo del record* (detto IDR). Un bitmap index rappresenta un modo alternativo di rappresentare gli IDR in un indice di tipo <valore, lista IDR>. Esso associa a ogni valore dell'attributo indicizzato un vettore di bit al posto della lista IDR, con il seguente significato: il vettore ha l'i-esimo bit settato a "1" se e solo se il relativo valore dell'attributo è presente nell'i-esimo record.
Tale tipo di indice sembra sprecare molto spazio su memoria di massa; ciò non è vero poiché i vettori di bit sono facilmente comprimibili. Inoltre i moderni processori sono particolarmente efficienti nell'eseguire operazioni logiche sui bit, e dunque un indice bitmap permette di ottimizzare interrogazioni con un predicato di partizione nella clausola WHERE del tipo "attributo = valore":

```
SELECT …
FROM <tabella>
WHERE <predicato di partizione> AND …
```

L'indice di giunzione, chiamato *Join Index*, è un altro tipo di indice utilizzato in sistemi di Data Warehousing. Lo scopo di questa struttura è di precalcolare la giunzione di tabelle, e memorizzarla in forma compatta per usi futuri. Supponiamo di voler creare un indice di giunzione tra la chiave primaria di una dimensione "D", e la corrispondente chiave esterna contenuta nella tabella dei fatti "F". L'indice risultante associa ogni

record della dimensione al relativo record della fact table, in una collezione di coppie <RID di D, RID si F>.

Le strutture appena introdotte sono utilizzate in maniera massiccia per l'ottimizzazione delle interrogazioni dette "star queries". Tale tipo di query chiede un equi-join tra la tabella dei fatti e una o più tabelle dimensionali, con possibili restrizioni, e il risultato può essere successivamente raggruppato e aggregato:

```
SELECT A,…,B,[COUNT(*),SUM(C),…,MAX(D)]
FROM T
[WHERE (condizioni su A,…,B)]
GROUP BY A,…,B
[HAVING (condizioni)]
[ORDER BY (A,…,B)]
```

Nel calcolare il piano di accesso per le star query, inizialmente si utilizzano gli indici bitmap per filtrare gli identificativi dei record che non soddisfano condizioni locali alle tabelle. Il risultato è memorizzato in una struttura dati locale usata in seguito insieme agli indici di giunzione per eseguire l'operazione di join in modo efficiente. A questo punto si accede al disco per leggere i record relativi agli identificativi selezionati, ed eventualmente sono svolte le operazioni di raggruppamento e aggregazione.

Un'altra ottimizzazione possibile è la creazione di viste materializzate, in altre parole delle vere e proprie tabelle memorizzate su disco, in modo da memorizzare in esse i risultati (o comunque dei risultati intermedi) delle interrogazioni eseguite frequentemente e che sono onerose in termini di tempo.
In alcuni casi la creazione delle viste è fatta da strumenti automatici che basandosi sulle statistiche di utilizzo del sistema, come ad esempio un file di log, selezionano le viste da materializzate allo scopo di migliorare le prestazioni complessive del sistema.
Per sfruttare a pieno le viste materializzate nella base di dati, il sistema di Data Warehousing dovrebbe essere in grado di trasformare in modo automatico le interrogazioni che potrebbero trarre beneficio dall'uso delle viste: tale operazione è detta *riscrittura delle interrogazioni con viste*, o semplicemente "*View Rewriting*".

2.3.1 Riscrittura delle interrogazioni con viste materializzate

Nei tradizionali sistemi per la gestione di basi di dati, le viste sono utilizzate per semplificare la scrittura di query complesse e per scrivere interrogazioni che non possono essere scritte senza l'utilizzo di viste. In ambiente DWH sono ampiamente utilizzate le viste materializzare per velocizzare la risposta del sistema nel caso di interrogazioni onerose eseguite frequentemente.

Come già detto, le interrogazioni in sistemi OLAP coinvolgono spesso operazioni di raggruppamento e aggregazione su grandi quantità di dati. In tale situazione un analista dei dati crea delle viste materializzate per memorizzare su disco i risultati più utilizzati, per poi poterli riutilizzare nel rispondere efficientemente a interrogazioni costose da eseguire in termini di tempo.
L'utilizzo delle viste materializzate può essere esplicito nella definizione della query, oppure il DMBS è capace di individuare automaticamente quando una certa interrogazione può essere riscritta rimpiazzando parte della sua definizione con un'espressione che coinvolge le viste.

Nella letteratura scientifica, tra i lavori al tema "Rispondere alle interrogazioni utilizzando le viste", sono stati proposti vari metodi per effettuare il *View Rewriting*, alcuni dei quali sono discussi nel capitolo 5.

Capitolo 3

Il Sistema per la gestione di basi di dati SADAS

SADAS, acronimo di "Sistema per l'Analisi dei Dati Statistici", è un DBMS relazionale specializzato nell'interrogazione di archivi statici di elevate dimensioni.
Un archivio è detto statico quando raramente è soggetto a operazioni di aggiornamento. Ciò significa che una volta raccolti, i dati saranno utilizzati solo per eseguire operazioni di analisi. Questa caratteristica permette al sistema SADAS di non dover affrontare i problemi che i tradizionali DBMS relazionali risolvono assicurando le cosiddette proprietà ACIDE.

Frutto della ricerca svolta dall'azienda *Advanced Systems* negli anni '80, SADAS si basa su un metodo di archiviazione dei dati colonnare (*column-based*) e su altre strutture di indicizzazione specializzate. Queste caratteristiche permettono di avere, in ambienti di Data Warehousing, performance migliori rispetto ai DBMS tradizionali.

In questo capitolo è presentata l'architettura del sistema per la gestione di basi di dati SADAS, e l'organizzazione fisica dei dati. Si mette, inoltre, in evidenza il ruolo dell'ottimizzatore del DBMS, in quanto è il modulo principalmente coinvolto in questo lavoro di tesi.

3.1 Architettura classica di un DBMS

Un sistema per la gestione di basi di dati, detto *DBMS* (*Database Management System*), è un software che permette la gestione di basi di dati[1]. Tale sistema permette di eseguire operazioni di creazione e gestione dello schema del database, garantisce la consistenza dei dati, e gestisce l'accesso a essi.

Le informazioni presenti nella base di dati sono organizzate dal DBMS usando un modello logico di rappresentazione dei dati. Il modello preso in considerazione in questo lavoro è quello *relazionale*, basato sull'algebra relazionale e sul concetto di relazione (o tabella).
Un DBMS che utilizza il modello dei dati relazionale è anche detto *RDMBS* (*Relational DBMS*). Per un'introduzione a tale modello dei dati si può far riferimento a [8].

Un sistema per la gestione di basi di dati espone all'utente diversi livelli di astrazione per accedere e gestire il database. Il *livello fisico* è il livello più basso, ed è strettamente dipendente dal modo in cui i dati sono realmente memorizzati su disco. A questo livello si possono effettuare operazioni per la gestione del database, e si definiscono caratteristiche utili per l'ottimizzazione delle prestazioni (come ad esempio la creazione di un indice).
Il *livello logico* è più astratto del *livello fisico* e descrive la struttura logica e le relazioni tra i dati; a questo livello sono definite le tabelle, gli attributi, i campi chiave, e altre regole d'integrità. Gli utenti che accedono al database devono conoscere i dettagli di questo livello.

[1] Una *base di dati* è definita in [8], come "una collezione di insiemi omogenei di dati, memorizzati in memoria di massa, fra i quali sono definite delle relazioni".

Per interagire con la base di dati, si utilizza normalmente il linguaggio *SQL*, acronimo di "*Structured Query Language*". Esso è stato progettato specificatamente per la gestione dei dati in sistemi di tipo relazionale, ed è composto dai seguenti gruppi di comandi:

- DDL (Data Definition Language): contiene i comandi utilizzati per definire la struttura (o schema) della base di dati.
- DML (Data Manipulation Language): contiene i comandi utilizzati per accedere ai dati, sia in lettura, sia in scrittura.
- DCL (Data Control Language): contiene i comandi utilizzare per gestire i permessi di accesso al database. Siccome diversi utenti e applicazioni possono condividere la stessa base di dati, è necessaria l'esistenza di un meccanismo di sicurezza, al fine di evitare accessi non autorizzati alle informazioni.
- TCL (Transaction Control): contiene i comandi utilizzati per la gestione delle transazioni.

Per eseguire un'*interrogazione*, o *query*, sulla base di dati, si usa il comando SELECT, facente parte del gruppo DML. Esso permette di descrivere l'informazione chiesta al DMBS, il quale fornisce la risposta in modo efficiente e ottimizzato.
Una query inizia sempre con la clausola SELECT, seguita da un'eventuale lista di attributi da includere nel risultato, e successivamente vi sono altre clausole, alcune delle quali sono opzionali. La struttura di questo "statement" è del tipo:

```
SELECT <lista di attributi>
FROM <lista di tabelle>
[ WHERE <condizioni sugli attribute delle tabelle> ]
[ GROUP BY <criterio di raggruppamento>
  [ HAVING <condizioni sui gruppi> ] ]
[ ORDER BY <criterio di ordinamento> ]
```

Le clausole racchiuse dalle parentesi quadre sono opzionali. La clausola FROM indica le tabelle dalle quali prendere le informazioni; esse possono eventualmente essere unite con operazioni di giunzione. La clausola WHERE indica delle condizioni di selezione sui valori degli attributi contenuti nelle tabelle specificate nella precedente clausola. Per

eseguire dei raggruppamenti si utilizza la clausola GROUP BY; dopo di essa può essere specificata la clausola HAVING, per indicare delle condizioni sui gruppi formati. Infine la clausola ORDER BY è utilizzata per specificare un criterio di ordinamento dei dati nella risposta.
Per una più estesa introduzione al linguaggio SQL e del comando SELECT, si può far riferimento a [7].

3.1.1 Moduli funzionali di un DBMS relazionale

In Figura 8 è mostrato un modello semplificato dell'architettura di un DBMS relazionale. I moduli funzionali principali sono chiamati "*macchina logica"* e "*macchina fisica"*.

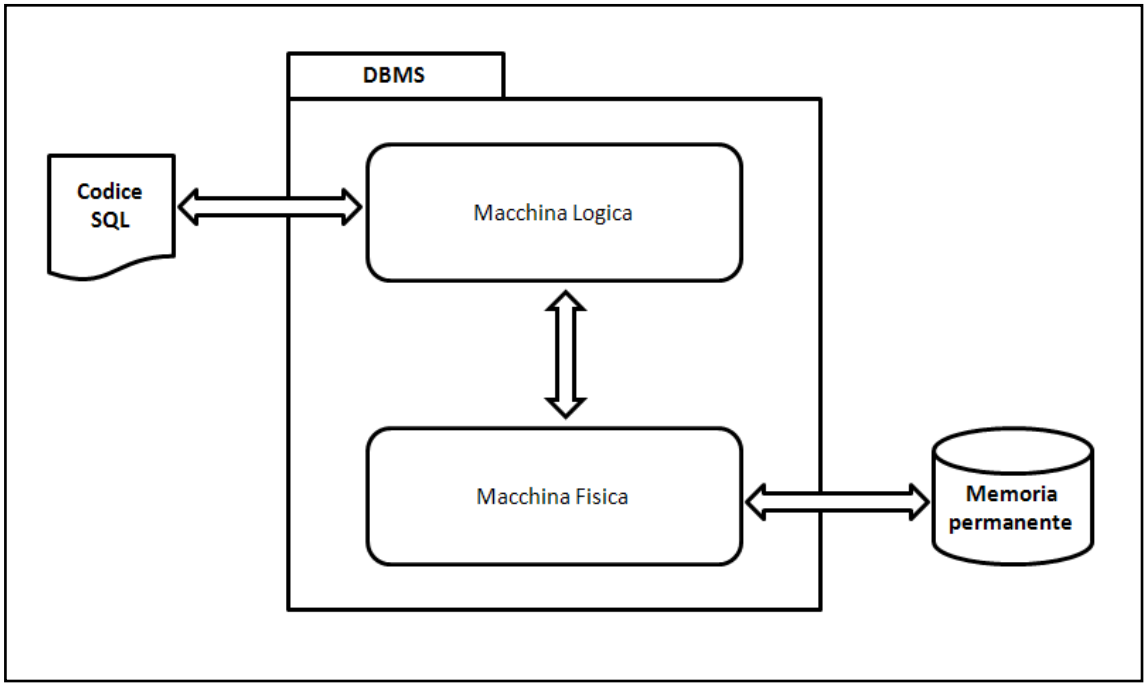

Figura 8: Moduli funzionali di un DBMS relazionale

I comandi SQL sono gestiti dal *processore SQL* contenuto nella *macchina logica*. Il suo compito è di controllare le autorizzazioni degli utenti, gestire il catalogo dei metadati, interpretare tutti i comandi SQL, e infine di tradurre le interrogazioni in piani di accesso. Quest'ultima struttura è data in input alla *macchina fisica*, la quale ha il compito di gestire la memoria relazionale.

La *macchina fisica* è composta dai seguenti sottomoduli funzionali:

- Gestore della memoria permanente: permette di vedere la memoria come un insieme di pagine con grandezza fissata o priori.
- Gestore del buffer: si occupa del trasferimento delle pagine dalla memoria temporanea a quella permanente e viceversa.
- Gestore dell'affidabilità: protegge il sistema da eventuali malfunzionamenti, permettendo cosi di preservare i dati a fronte di problemi hardware o software.
- Gestore della concorrenza: gestisce gli accessi concorrenti.
- Gestore delle strutture di memorizzazione e dei metodi di accesso: astrae i dati dalle strutture usate per la loro archiviazione, offrendo agli altri moduli, la possibilità di vedere i dati organizzati in collezioni di record e indici. Fornisce, inoltre, i metodi di accesso a essi come ad esempio una struttura indice oppure in modo sequenziale.

In [9] sono descritte in modo dettagliato le principali tecniche d'implementazione utilizzate dai DBMS relazionali. Nel seguito di questo capitolo la discussione è incentrata esclusivamente sulla struttura del sistema SADAS.

3.2 Architettura di SADAS

Dallo schema complessivo dell'architettura di SADAS, presentato in Figura 9, si nota che essa è costituita da diversi strati, ognuno dei quali destinato a una specifica funzionalità.

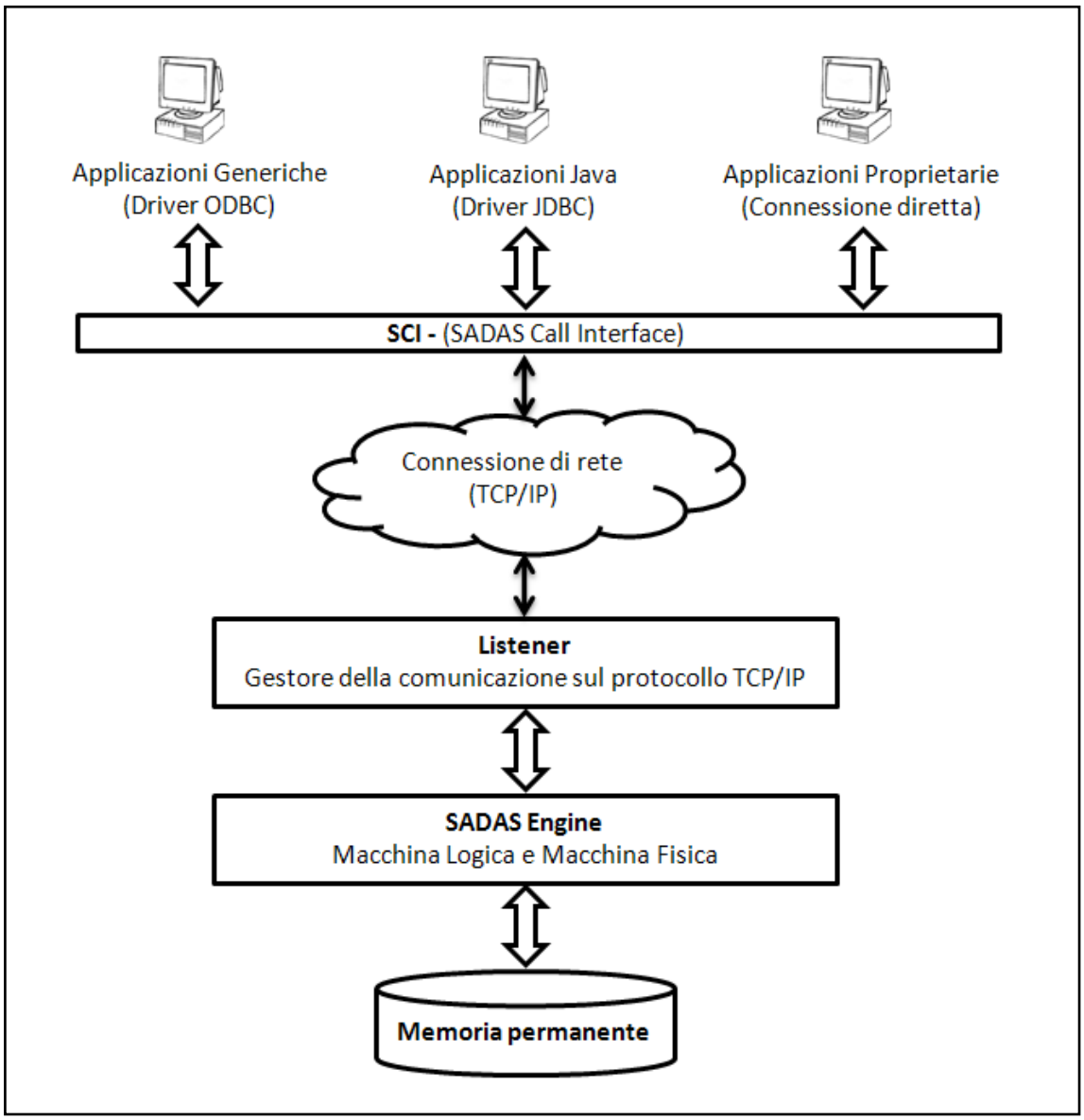

Figura 9: Architettura del sistema SADAS

Lo strato che è direttamente a contatto con le applicazioni utente, è lo strato "SCI" (SADAS Call Interface). Esso permette la comunicazione tra le applicazioni e il motore di SADAS, ed è implementato sotto forma di libreria software utilizzabile direttamente, come nel caso delle applicazioni proprietarie di Advanced Systems, oppure mediante driver ODBC o JDBC.

Lo strato direttamente collegato a SCI è quello del "*Listener"*. La comunicazione tra loro è basata sul protocollo di rete TCP/IP, definendo dunque un'applicazione client-server[2]. Il *Listener* interpreta dunque i messaggi in arrivo, e li inoltra allo strato "*SADAS Engine"*, il quale è il modulo centrale del sistema SADAS. L'ultimo strato è rappresentato dalla memoria permanente, utilizzata dal sistema per archiviare i dati.

Nei prossimi due paragrafi sono esaminati in dettaglio il "Listener" e "SADAS Engine".

3.3 Il modulo "Listener"

Il modulo "*Listener*" gestisce le richieste provenienti dai vari client, si occupa poi di inoltrarle ai livelli sottostanti, e attende l'esito della computazione per poi fornire ai client la risposta per tali richieste.
Il *Listener* è di tipo multi-thread[3]; il thread principale attende richieste di connessione da parte di processi client utilizzando il protocollo TCP/IP, e per ognuna di esse crea un thread separato per gestire la comunicazione con il client specifico.

Essendo SADAS un sistema per la gestione di basi di dati, può capitare che la richiesta del client preveda una risposta di dimensione notevole. Per questo motivo la risposta è paginata, ossia divisa in diverse pagine di dati, ognuna restituita al client dietro un'esplicita richiesta. Questo meccanismo consente di evitare blocchi del software dovuti alla trasmissione di grosse quantità di dati.
Il thread creato dal *Listener* deve essere dunque in grado di conservare lo stato della comunicazione con il client, e lo stato dell'elaborazione della richiesta da parte di SADAS Engine.

[2] Paradigma di computazione nel quale un'applicazione "client" fruisce dei servizi offerti da un'applicazione "server", connettendosi ad essa utilizzando un protocollo di rete.
[3] Un software multi-thread è composto da più thread, o filoni di esecuzione, eseguiti in modo concorrente.

Dopo aver creato il thread per la gestione della singola richiesta del client, il Listener ritorna subito in ascolto di nuove richieste di connessione, realizzando una forma parallelismo *inter-query* dovuta all'elaborazione contemporanea di più thread.

3.4 Il modulo "SADAS Engine"

Il modulo "*SADAS Engine*", in breve *Engine*, è il cuore del sistema per la gestione di basi di dati SADAS. La sua struttura, mostrata in Figura 10, rispetta l'architettura utilizzata per i tradizionali DBMS relazionali, ed è dunque composta dai sotto-moduli "Macchina Logica" e "Macchina Fisica".

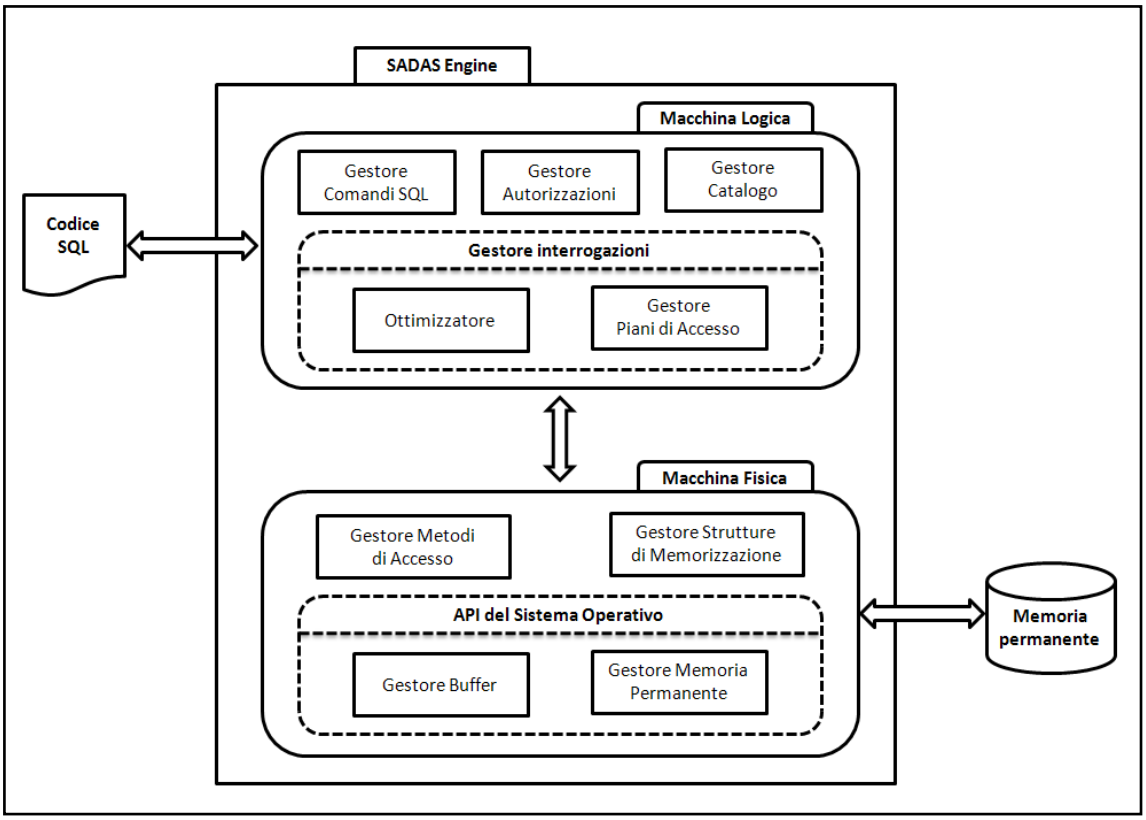

Figura 10: Il modulo SADAS Engine

3.4.1 La macchina logica

La *macchina logica* ha il compito di trasformare un comando SQL dato in input, in un piano d'accesso per la *macchina fisica*. Quest'ultimo identifica una strategia di esecuzione di un'interrogazione sulla base di dati: è formato da un albero, dove ogni nodo rappresenta un metodo di accesso o un *operatore fisico*, cioè un algoritmo che realizza un operatore dell'algebra relazionale.

I moduli funzionali della macchina logica sono: "*gestore del catalogo*", "*gestore delle autorizzazioni*", "*gestore dei comandi SQL*", e "*gestore delle interrogazioni*".

3.4.1.1 Gestore del catalogo

Il *catalogo* contiene le informazioni sui vari database mantenuti da SADAS, ed è amministrato dal *gestore del catalogo*. Per ognuno di essi fornisce informazioni riguardanti lo schema, e per ogni relazione conserva le informazioni quantitative, statistiche, e di stato utilizzate dagli altri moduli per ottimizzare i tempi d'interrogazione.

3.4.1.2 Gestore delle autorizzazioni

Il *gestore delle autorizzazioni* implementa il controllo degli accessi per SADAS. La politica utilizzata è sostanzialmente di tipo RBAC[4], dove a ogni funzionalità del sistema è associato uno dei seguenti ruoli:

- Ruolo "Select": consente di eseguire comandi SQL di tipo SELECT.
- Ruolo "Describe": consente l'acquisizione d'informazioni generali sullo stato del database e delle relative tabelle.
- Ruolo "Catalog": consente di gestire le informazioni contenute nel catalogo.

[4] RBAC, acronimo di "Role-Based Access Control", è un tipo particolare di controllo degli accessi il quale associa i permessi agli utenti in base al ruolo che essi ricoprono.

- Ruolo “Load”: consente di eseguire le attività caricamento dei dati nel database.
- Ruolo “Security”: consente di creare, aggiornare, e cancellare utenti e ruoli.

La definizione dei ruoli è comune a tutti i database, mentre la definizione degli utenti riguarda lo specifico database.
Oltre ad associare i ruoli agli utenti, per ognuno di essi è possibile abilitare o disabilitare l’accesso ai dati di ogni singola tabella. Volendo una granularità più fine, si può associare una “stringa di parzializzazione” all’utente, in modo da permettergli un accesso parziale ai dati di una tabella; ad esempio, in un archivio vendite di un magazzino, si può limitare l’accesso ai dati esclusivamente a un particolare prodotto.

3.4.1.3 Gestore dei comandi SQL e delle interrogazioni

Il *gestore dei comandi SQL* è il modulo che accetta direttamente lo script SQL in input, lo esamina per capire a quale sottoinsieme di SQL appartiene, e in seguito interagisce con il gestore delle autorizzazioni per verificare i permessi dell’utente.
Nel caso il comando SQL appartenga al gruppo DML, DCL, o DDL, il controllo passa al *gestore delle interrogazioni*. Quest’ultimo è diviso in due ulteriori sotto-moduli: l’ottimizzatore e il gestore dei piani di accesso.

Il sotto-modulo “*ottimizzatore”* ha il compito di determinare un piano di accesso per l’interrogazione in input. Il suo funzionamento è descritto in dettaglio nel capitolo 3.

Una volta determinato il piano di accesso, esso è dato in input al sotto-modulo “gestore dei piani di accesso”, il quale lo esegue utilizzando le funzionalità fornite dalla *macchina fisica*.
L’esecuzione di un piano di accesso in SADAS è effettuata utilizzando gli iteratori[5]; dunque le tuple appartenenti alla risposta sono prodotte su richiesta, permettendo il

[5] Nel contesto delle basi di dati, un iteratore o cursore, è un oggetto che fornisce un modo agevole per visitare gli elementi contenuti nella risposta di un’interrogazione.

meccanismo di paginazione della risposta. Non sempre utilizzare un iteratore evita di creare relazioni temporanee, e talvolta c'è necessità di materializzare i risultati intermedi.

3.4.2 La macchina fisica

La *macchina fisica* ha il compito di gestire la memoria permanente e volatile, esportando i metodi per l'accesso e per la memorizzazione. I moduli della macchina fisica sono: "gestore dei metodi di accesso", "gestore delle strutture di memorizzazione", "gestore del buffer", e il "gestore della memoria permanente".

Non esiste un'implementazione proprietaria in SADAS per tutti i moduli della macchina fisica: infatti, per il gestore del buffer e per il gestore della memoria permanente, ci si affida alle funzionalità del sistema operativo. Tali moduli implementando dunque un wrapper[6] per interfacciarsi con le chiamate di sistema del sistema operativo, che permettono: di accedere ai file in memoria di massa, di creare e sincronizzare i thread, e di gestire la memoria volatile.
La macchina fisica di SADAS, quando possibile, esegue operazioni in parallelo avvalendosi delle funzionalità multithreading del sistema operativo e utilizzando la libreria OpenMP (Open specifications for Multi Processing, [10]).

Tradizionalmente i DBMS sono basati su una tecnologia orientata alle righe (row-oriented); la relativa macchina fisica memorizza una tabella del database per righe in un insieme di pagine fisiche. Questa soluzione è pensata per applicazioni OLTP, dove l'accesso ai dati spesso riguarda un singolo record o comunque un piccolo insieme di record.

[6] Un wrapper è un componente software che permette l'interoperabilità tra moduli software aventi interfacce differenti. Si può anche definire un wrapper allo scopo di diminuire l'accoppiamento tra i vari moduli software.

Per migliorare le performance delle interrogazioni di tipo OLTP, le quali riguardano spesso grandi quantità di dati, si può adottare un partizionamento verticale dei dati in cui le tabelle sono memorizzate per colonne.
I sistemi che utilizzano questo tipo di partizionamento, tra cui SADAS, sono anche chiamati DMBS colonnari (*column-oriented DBMS*).

3.4.2.1 Partizionamento verticale dei dati

Memorizzare una tabella per colonne, porta vantaggi per le operazioni di: restrizione (*clausola WHERE*), raggruppamento (*clausola GROUP BY*), ordinamento (*clausola ORDER BY*), e giunzione.

Il sistema SADAS adotta un processo di "polverizzazione" della tabella che prevede come prima operazione la creazione di un file, detto *file CLN*, per ogni colonna. In Figura 11 si mostra un esempio in cui, per una tabella con quattro colonne, si creano quattro file CLN separati.

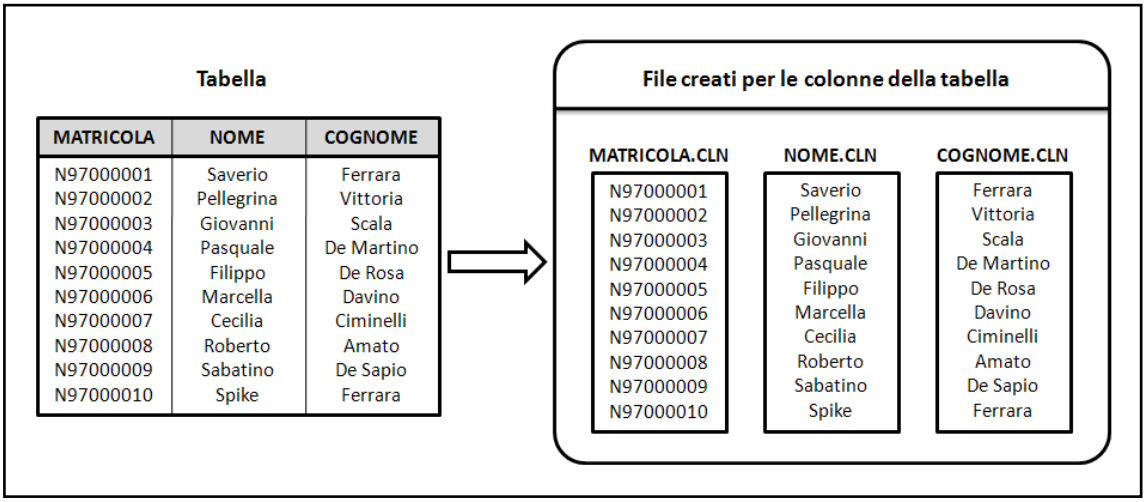

Figura 11: Partizionamento verticale dei dati

Al fine di velocizzare le successive operazioni di interrogazione, durante la creazione dei file CLN, si rende costante la lunghezza dei valori riguardanti una particolare colonna. A ogni elemento del file CLN è associato un identificativo del record (detto IDR), che coincide sempre con il numero progressivo del record.

In seguito alla creazione dei file CLN, per ogni attributo si creano altre strutture che possono essere di tipo tradizionale, come ad esempio i B+Alberi, oppure strutture sviluppate all'interno dell'azienda Advanced Systems.
Le tecniche utilizzate e le scelte implementative riguardanti queste strutture sono esaminate in [11].

Capitolo 4

L’ottimizzatore di SADAS

La *macchina logica* ha il compito di trasformare un comando SQL dato in input, in un piano d’accesso per la *macchina fisica*; l’ottimizzatore in SADAS è un sottomodulo del “gestore delle interrogazioni” incluso nella *macchina logica*.

Il gestore delle interrogazioni ha il compito di esaminare ed eseguire gli script SQL appartenenti ai gruppi DML, DCL, o DDL. In particolare per gli script SQL di tipo DML, il gestore delle interrogazioni si serve dell’*ottimizzatore* per individuare il piano di accesso, cioè una strategia di esecuzione dell’interrogazione. Il piano d’accesso è in seguito dato in input al “gestore dei piani di accesso”, il quale lo esegue servendosi delle funzionalità fornite dalla *macchina fisica*.

In SADAS lo script SQL esaminato dall’ottimizzatore è descritto da un’istanza della classe TSDSSQLStatement, creata in precedenza dal parser SQL. In questo capitolo si esamina il funzionamento degli algoritmi usati nell’ottimizzatore e nel parser SQL, entrambi scritti utilizzando il linguaggio C++. Particolare attenzione si è posta per la descrizione delle strutture dati utilizzate, le quali sono alla base del funzionamento di SADAS Engine.

4.1 Il parser SQL

Il *parser* SQL di SADAS, essendo costruito seguendo la teoria dei linguaggi formali ([12]), richiede quattro fasi: analisi lessicale, analisi sintattica, analisi semantica (la quale include le azioni semantiche), e analisi pragmatica (o esecuzione).

Tradizionalmente con il termine "parser" ci si riferisce esclusivamente alla fase di analisi sintattica; in questa trattazione useremo il termine "parser" per riferirci a tutte le quattro fasi, ciascuna delle quali ha all'interno di SADAS un modulo separato (Figura 12).

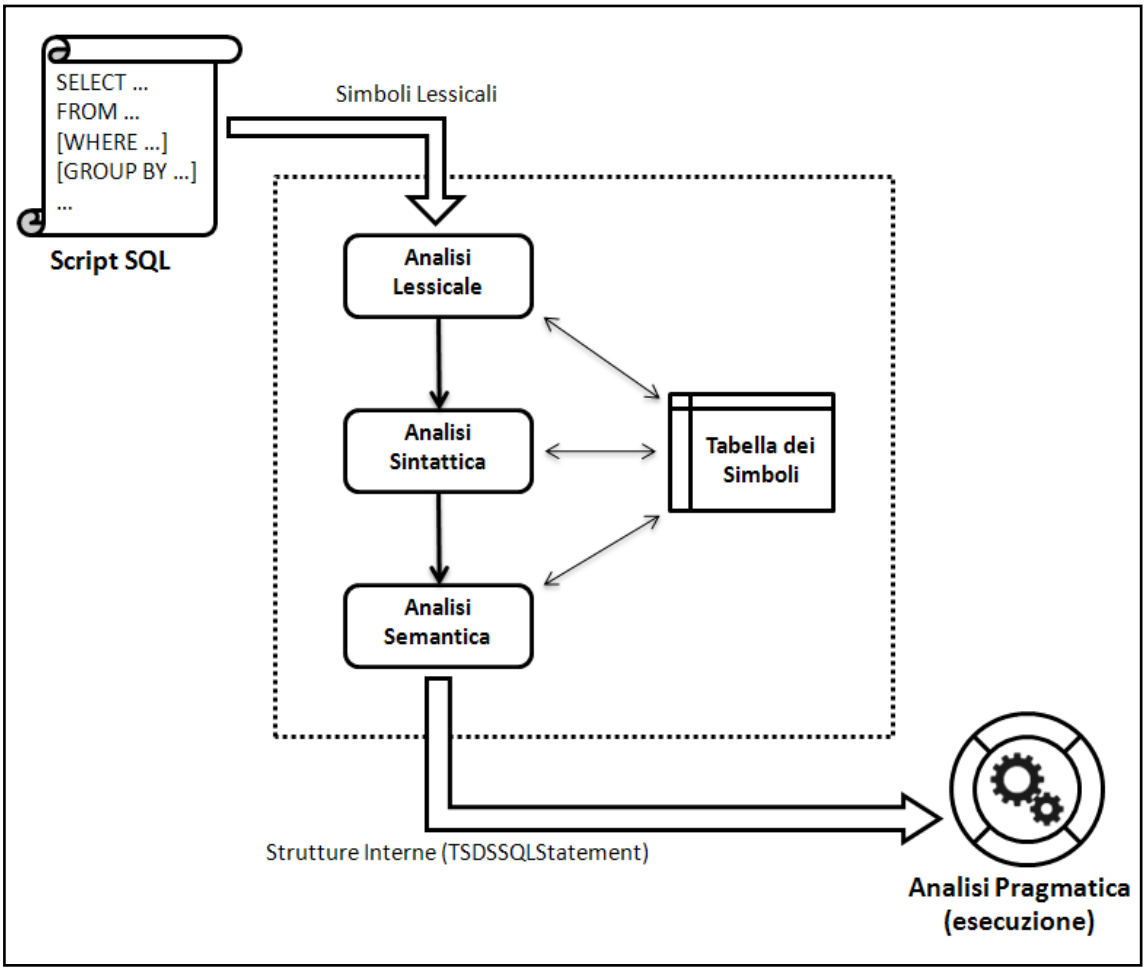

Figura 12: Struttura del parser SQL di SADAS

Ogni script SQL inviato SADAS Engine è dunque elaborato in sequenza dai moduli del parser. La prima fase è di analisi lessicale, la quale trasforma lo script SQL sorgente in una serie di simboli (detti *tokens*), utili per la successiva fase di analisi sintattica. Questi simboli sono raggruppati nelle seguenti categorie:

- *Parole chiave*: sono quelle fissate dallo standard SQL, hanno un significato preciso, e devono essere utilizzate solo nel contesto sintattico giusto;
- *Identificativi*: sono parole che fanno riferimento a una colonna, a una tabella, o a un alias. I nomi delle colonne e delle tabelle sono accessibili dal catalogo del database;
- *Costanti, commenti, e caratteri speciali*.

Il modulo di analisi lessicale usa un *automa a stati finiti* per individuare i simboli costituenti l'interrogazione SQL; successivamente tutti i tokens incontrati sono memorizzati nella *tabella dei simboli*, in modo da renderli accessibili alle fasi successive di analisi sintattica e analisi semantica.

La seconda fase è di analisi sintattica, la quale controlla, utilizzando i tokens riconosciuti dal modulo di analisi lessicale, se lo script SQL rispetta la grammatica usata nel dialetto SQL adottato in SADAS. Questa è una fase molto complessa, non solo dal punto di vista intrinseco degli algoritmi usati nel parser, ma riguardo lo sviluppo degli algoritmi stessi. Per questo motivo si sono utilizzati YACC e GNU BISON, dei generatori di parser open source reperibili in [13], per generare il codice sorgente in linguaggio C del modulo di analisi sintattica. Il codice sorgente generato implementa un automa a pila (*push-down automata*), e rappresenta la soluzione classica nelle tecniche di compilazione con *linguaggi non contestuali* (detti *context-free* nella classificazione di Chomsky).

Il risultato della fase di analisi sintattica è un albero sintattico equivalente allo script SQL in input, rappresentato da un'istanza della classe TSDSSQLStatement (Figura Z). L'intera struttura è costruita dalle azioni semantiche effettuate durante la fase successiva, detta di analisi semantica. In realtà la fase di analisi sintattica e quella di analisi semantica sono attivate simultaneamente: per ogni token individuato dall'analisi sintattica, è eseguita un'appropriata azione semantica, in modo da generare o gestire le strutture interne del parser utilizzando le informazioni estratte dal modulo di analisi sintattica.

Oltre ad occuparsi delle strutture interne di TSDSSQLStatement, il modulo di analisi semantica ha il compito di verificare la correttezza semantica dello script SQL. A tale scopo si eseguono i seguenti controlli di uso comune nell'analisi semantica:

- Analisi della compatibilità dei tipi concernenti gli attributi specificati nelle varie clausole;
- Analisi della compatibilità dei tipi dei parametri attuali delle funzioni, con i relativi tipi dei parametri formali;
- Verifica del corretto utilizzo degli operatori utilizzati nelle condizioni;
- Altre verifiche specifiche del dialetto SQL adottato in SADAS.

La correttezza esecuzione di queste prime tre fasi determina la correttezza grammaticale e semantica del codice SQL dato in input al sistema SADAS Engine. L'output del parser è, dunque, un'istanza della classe TSDSSQLStatement, la quale rappresenta l'interrogazione all'interno di SADAS.

Quest'oggetto, se relativo a uno script SQL di tipo DML, è dato in input all'*ottimizzatore*, che ha il compito di individuare il piano di accesso, cioè una strategia di esecuzione dell'interrogazione. Il piano d'accesso è poi dato in input al "gestore dei piani di accesso", il quale lo esegue servendosi delle funzionalità fornite dalla *macchina fisica*.

4.1.1 Struttura della classe TSDSSQLStatement

Un'istanza della classe TSDSSQLStatement, generata dal parser SQL di SADAS, rappresenta, all'interno di SADAS Engine, l'interrogazione da eseguire.

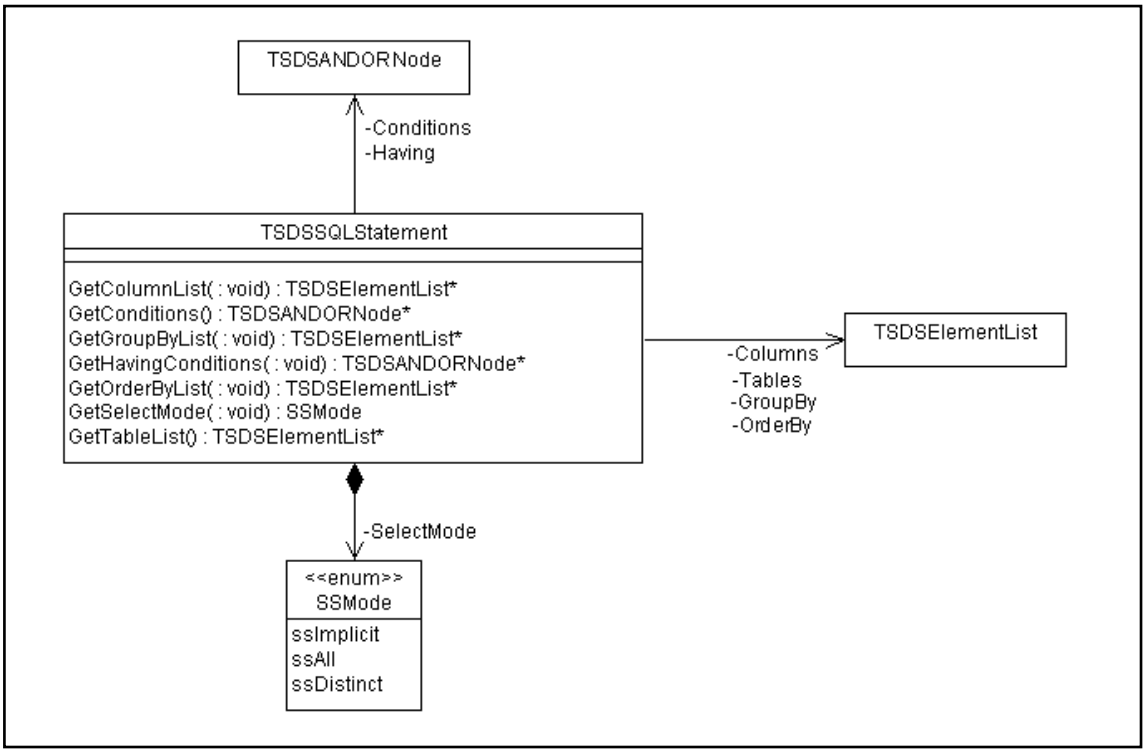

Figura 13: Parte dei metodi offerti dalla classe TSDSSQLStatement

La struttura interna della classe, accessibile attraverso i metodi illustrati in Figura 13, contiene i riferimenti a tutti gli elementi specificati nell'interrogazione SQL analizzata.

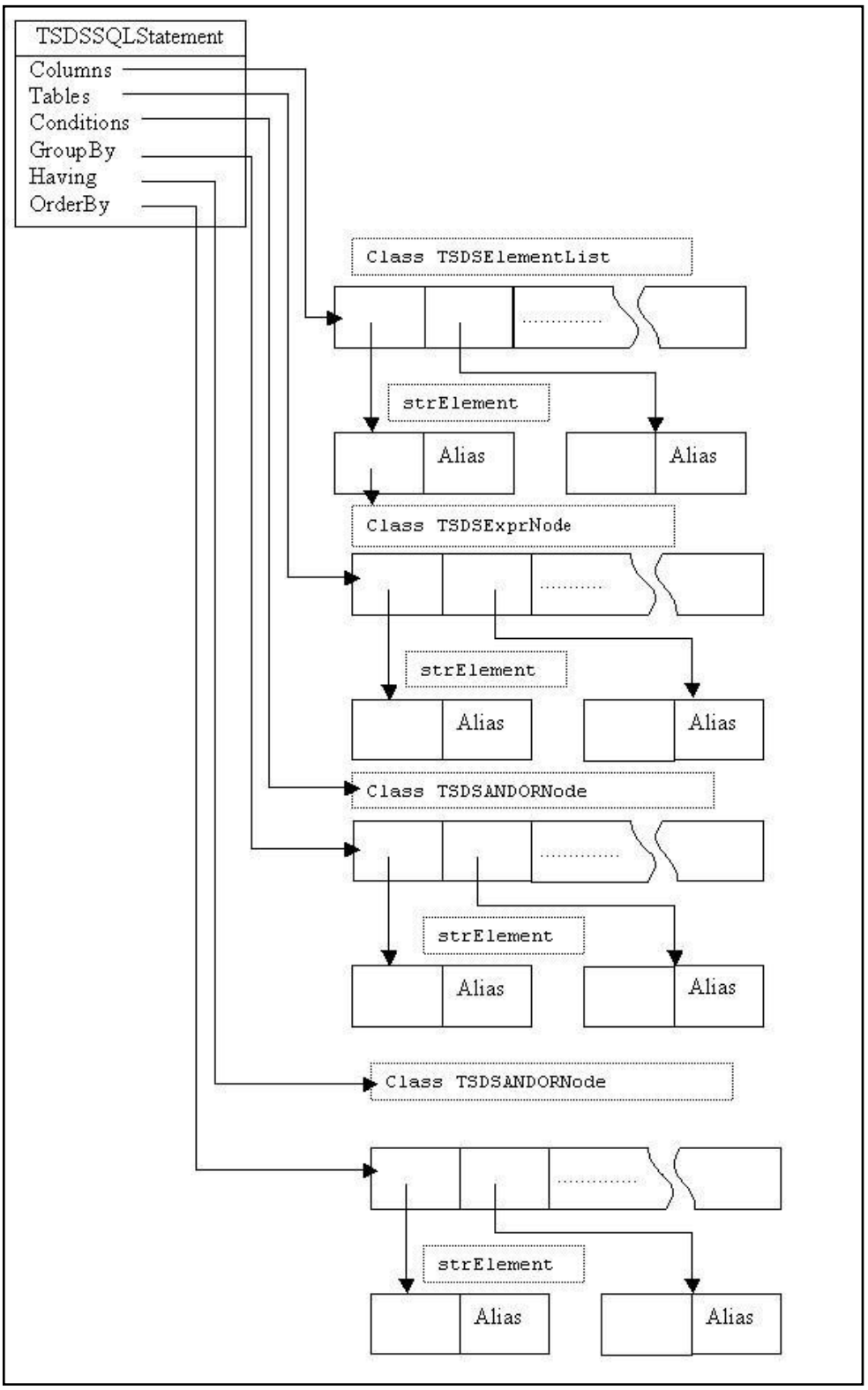

Figura 14: Struttura degli attributi della classe TSDSSQLStatement

In Figura 14, sono raffigurati i seguenti attributi privati:

- *SelectMode*: indica se la clausola *SELECT* dell'interrogazione è specificata con "*ALL*" oppure con "*DISTINCT*". Questa proprietà *SelectMode* è sempre specificata;
- *Columns*: lista contenente gli attributi specificati nella clausola *SELECT*;
- *Tables*: lista contenente i riferimenti alle tabelle specificate nell'interrogazione;
- *Conditions*: puntatore all'albero delle condizioni per la clausola *WHERE*;
- *GroupBy*: puntatore alla lista degli attributi di raggruppamento;
- *Having*: puntatore all'albero della condizione per la clausola *HAVING*;
- *OrderBy*: puntatore alla lista degli attributi di raggruppamento.

Le proprietà Conditions e Having sono puntatori a una struttura ad albero, mentre le proprietà Columns, Tables, GroupBy, e OrderBy sono liste di tipo TSDSElementList. Ogni elemento di questa lista è di tipo strElement, e a sua volta contiene un nodo di tipo TSDSExprNode. Quest'ultimo è di particolare importanza all'interno del sistema poiché rappresenta un'*espressione* di tipo aritmetico. Inoltre la classe TSDSExprNode specializza la classe TSDSNNode, la quale implementa un albero n-ario[7]: l'intera gerarchia è illustrata in Figura 15.

Un nodo dell'albero n-ario è istanza della classe TSDSNNode, e contiene l'informazione sul tipo di operazione implementata (logica, aritmetica, oppure una condizione). La classe esporta i metodi generali per la gestione dell'albero, e le funzioni per ottenere informazioni sul tipo di nodo. Il parser SQL si SADAS non utilizza mai direttamente questa classe, ma solo tramite i tipi che da essa derivano: TSDSExprNode e TSDSANDORNode.

[7] Un albero n-ario è un albero i cui nodi hanno, al più, grado n; per albero si intende un grafo connesso, non diretto, ed aciclico.

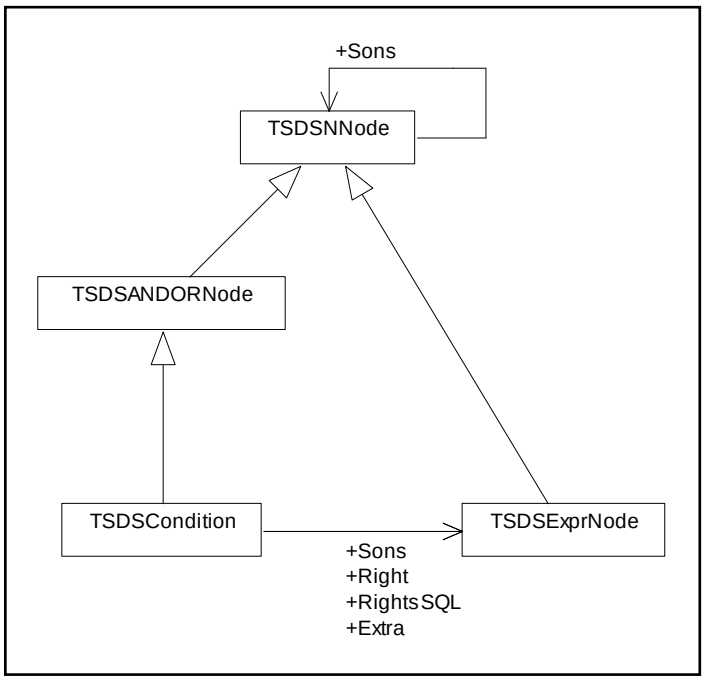

Figura 15: Gerarchia delle classi per i nodi dell'albero

La classe TSDSANDORNode specializza la classe TSDSNNode in modo da implementare un albero logico AND/OR, utile a rappresentare le operazioni logiche che si trovano nelle condizioni dello script SQL. Questo è il tipo di albero utilizzato per le proprietà Conditions e Having della classe TSDSSQLStatement.

La classe TSDSCondition, sottoclasse della classe TSDSANDORNode, implementa la struttura del nodo per una condizione di arietà unaria, binaria, o, in alcuni casi, ternaria (come nel caso dell'operatore *between*).

L'albero finale contiene una combinazione di nodi *TSDSANDORNode*, *TSDSCondition* e *TSDSExprNode*. Al livello superiore, l'albero può contenere nodi di tipo *TSDSANDORNode* (perché il livello superiore delle condizioni è quello logico); da un certo livello in giù l'albero ha nodi *TSDSCondition* (i figli tale tipo di nodo sono di tipo *TSDSExprNode*, cioè espressioni aritmetiche); al più basso livello si trovano le foglie,

sempre di tipo TSDSExprNode, con i riferimenti agli attributi delle tabelle e a valori costanti.

4.2 L'ottimizzazione dell'interrogazione

L'ottimizzatore è il modulo del DBMS che, data in input un'interrogazione, ha il compito di individuare per essa un "piano di accesso": un albero di operatori fisici, scelti tra quelli messi a disposizione dalla macchina fisica, il quale descrive le operazioni da eseguire per risolvere la query.

In generale esistono diverse strategie alternative per eseguire un'interrogazione, cosa tanto più vera quanto più un'interrogazione è complessa. Il problema di individuare il piano di accesso più conveniente per eseguire un'interrogazione, ha una complessità di tempo superpolinomiale e rientra dunque nella categoria dei problemi detti "difficili". Questo è dovuto a diversi fattori, tra cui:

- un'interrogazione può essere trasformata in più alberi di operatori fisici a essa equivalenti;
- gli operatori dell'algebra relazionale possono essere implementati da diversi operatori fisici;
- gli operatori fisici possono utilizzare diversi metodi di accesso, relativi a strutture fisiche alternative.

Nella pratica si utilizzano procedure euristiche con l'obiettivo di trovare rapidamente una soluzione approssimata al problema, esplorando, in modo controllato, lo spazio delle possibili soluzioni.

Si possono distinguere due tipi di ottimizzazione: *statica* o *dinamica*. Nel caso di accesso interattivo alla base dati l'ottimizzazione è sempre dinamica, cioè il piano di accesso è generato nel momento in cui si esegue l'interrogazione. Questo tipo di

ottimizzazione può utilizzare i dati statistici aggiornati e le strutture fisiche che in quel momento sono messe a disposizione dal sistema e registrate nel catalogo del database. Nel sistema SADAS Engine l'ottimizzazione è sempre di tipo dinamico.
Nel caso di interrogazioni effettuate da un programma scritto in un certo linguaggio di programmazione (*embedded queries*), l'ottimizzazione può essere sia di tipo dinamico, sia di tipo statico. Nel primo caso l'ottimizzazione è fatta in modo analogo al caso in cui si accede interattivamente alla base di dati; nel secondo caso l'ottimizzazione, e quindi la creazione del piano di accesso, è fatta una sola volta al momento dell'analisi dell'interrogazione da parte del precompilatore.

Un altro criterio di classificazione degli ottimizzatori è il modo in cui generano il piano di accesso; in genere si opera secondo due principi: il principio dei costi (detto "cost-based") e il principio delle regole (detto "rule-based").

Un ottimizzatore basato su regole è in genere veloce poiché applica all'interrogazione una lista di regole per generare un piano di accesso. Tale tipo di ottimizzatore, non utilizzando informazioni statistiche relative ai costi di accesso alle strutture dati, può scegliere di utilizzare strutture dati o metodi di accesso costosi da un punto di vista computazionale, che sarebbero banalmente evitati da un ottimizzatore basato sui costi. Quest'ultimo tipo di ottimizzatore in genere impiega più tempo per la costruzione del piano di accesso.

Nella pratica gli ottimizzatori utilizzati nei DBMS commerciali sono di tipo ibrido: si permette agli ottimizzatori basati su regole di utilizzare informazioni statistiche per migliorare una parte del piano di accesso, oppure si permette agli ottimizzatori basati sui costi di utilizzare regole per generare più piani di accesso, per poi scegliere quello di costo stimato minimo.

Anche il modulo di ottimizzazione in SADAS Engine è di tipo ibrido, e, essendo basato principalmente su regole, opera in varie fasi.

Nella prima fase, detta "*fase di semplificazione"*, si prende in input l'interrogazione SQL descritta da un oggetto di tipo TSDSSQLStatement, e si eseguono su di esso delle operazioni di semplificazione. Queste operazioni hanno lo scopo di semplificare gli alberi costruiti, e ottenere così strutture dati di dimensioni ridotte. In alcuni casi questa fase elimina espressioni complesse, a vantaggio del tempo impiegato nell'esecuzione. In casi particolari questa fase di semplificazione può individuare dei predicati dell'interrogazione valutati sempre a "false": questo ci assicura che il risultato della *query* sia vuoto.

In seguito alla *fase di semplificazione*, si compie una "*fase di trasformazione logica"* per ottenere un albero, equivalente all'interrogazione in input, in cui ogni nodo è relativo a un operatore dell'algebra relazionale (Figura 16).

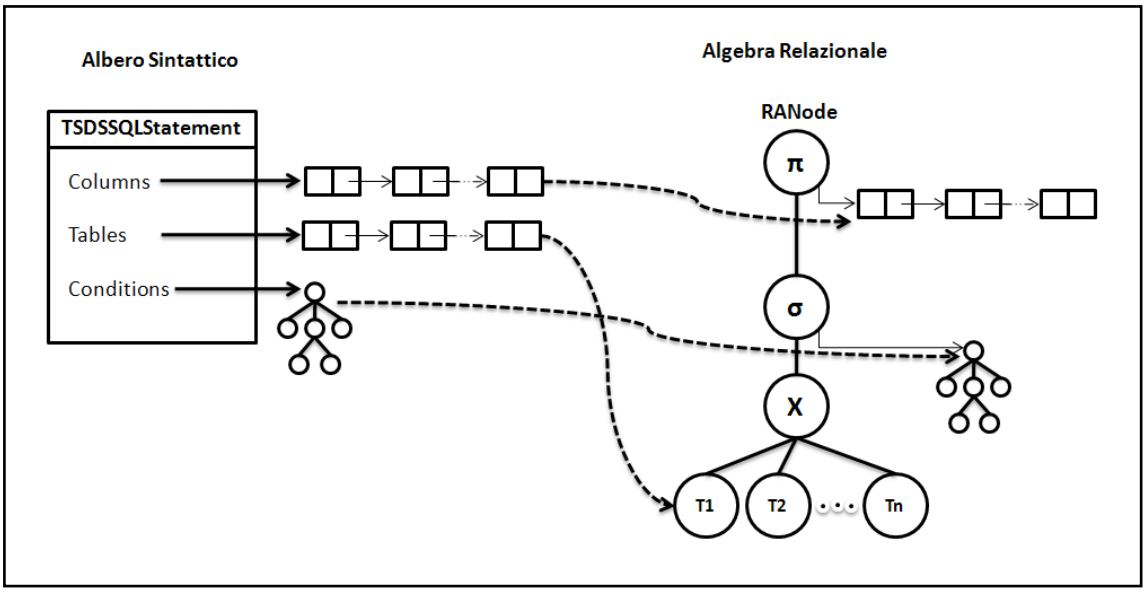

Figura 16: Trasformazione logica

La struttura ottenuta è successivamente trasformata e ottimizzata nelle fasi successive. L'ultima fase, detta "*fase di ottimizzazione fisica"*, prende in input l'ultima forma dell'espressione dell'algebra relazionale, e genera il piano di accesso, utilizzando le definizioni degli operatori fisici messi a disposizione dalla macchina fisica.

Il piano d'accesso è dopo dato in input al "gestore dei piani di accesso", il quale lo esegue servendosi delle funzionalità fornite dalla *macchina fisica*.

Capitolo 5

Rispondere alle interrogazioni utilizzando le viste

Il problema di rispondere a query utilizzando le viste è quello di trovare metodi efficaci per elaborare la risposta ad una query utilizzando un insieme di viste materializzate in precedenza definite sulla base di dati, piuttosto che accedere alle relazioni del database. Il problema è rilevante per una vasta gamma di applicazione riguardanti la gestione dei dati. Nell'ottimizzazione delle query, trovare una riscrittura di una query usando un insieme di viste materializzate, può risultare in un piano di esecuzione più efficiente. Per supportare la separazione delle viste logiche e fisiche di dati, uno storage schema può essere descritto usando le viste sullo schema logico. Di conseguenza, per risolvere il problema di rispondere alle query utilizzando le viste c'è bisogno di trovare un piano di esecuzione per la query da eseguire. Infine, il problema si pone in sistemi di integrazione dati, in cui le fonti di dati possono essere descritte come viste precalcolate su uno *schema intermedio* (*mediated schema*).
In questo capitolo si esaminano alcuni lavori sul problema, e si sintetizzano i numerosi e vari contributi in un quadro coerente. Inoltre si descrivono le diverse applicazioni del problema e alcuni degli algoritmi proposti per risolverlo.

5.1 Introduzione al problema

Il problema di rispondere a query usando le viste (*answering queries using views*), anche conosciuto come riscrittura di query usando le viste (*rewriting queries using views*), ha avuto molto successo per via della sua rilevanza in una vasta gamma di

problemi di gestione dei dati: ottimizzazione delle query, l'integrazione dati, e altre applicazioni quali la manutenzione dell'indipendenza fisica dei dati e la progettazione di Data Warehouse. Come formulato in [14], il problema è il seguente:

"Supponiamo di avere una query *Q* su uno schema di database, e una serie di definizioni di viste $V_1,\ldots,V_n$ sullo stesso schema. È possibile rispondere alla query *Q* utilizzando *solo* le risposte delle viste $V_1,\ldots,V_n$? In alternativa, qual è l'insieme massimo di tuple prodotte da *Q* che si possono ottenere dalle viste? Se siamo in grado di accedere sia alle viste, sia alle relazioni del database, qual è il piano di esecuzione più conveniente per rispondere alla query *Q*?".

La prima classe di applicazioni in cui incontriamo il problema di rispondere a query utilizzando le viste è l'*ottimizzazione delle query*. In questo contesto, calcolare la risposta una query utilizzando una vista precedentemente materializzata può accelerarne l'elaborazione, poiché una parte del calcolo necessario potrebbe essere già stato fatto durante il calcolo della vista. Queste ottimizzazioni sono particolarmente rilevanti in applicazioni di supporto alle decisioni, quando le viste e le query contengono operazioni di raggruppamento e aggregazione.

Una seconda classe di applicazioni in cui il problema si pone è l'*integrazione dati* (*data integration*). Sistemi di integrazione dati forniscono un'interfaccia uniforme (sulla quale è possibile eseguire query) per una moltitudine di fonti di dati autonome, che possono risiedere all'interno di un'azienda o sul World-Wide Web. I sistemi di integrazione dati automaticamente individuano le fonti dati rilevanti per una query, interagiscono con ognuna isolatamente, e successivamente combinano i dati provenienti da fonti multiple. In questa situazione gli utenti non pongono le query facendo riferimento agli schemi delle fonti dati, ma in termini di uno *schema intermedio*. Quest'ultimo è un insieme di relazioni progettato per una specifica applicazione di integrazione dei dati, e contiene gli aspetti salienti del dominio specifico. Le tuple delle relazioni dello schema intermedio non sono effettivamente memorizzate. Invece, il sistema include una serie di

descrizioni delle fonti che forniscono una corrispondenza semantica tra le relazioni degli schemi delle fonti dati e le relazioni dello schema intermedio.
In alcuni sistemi di integrazione dati si segue un approccio in cui il contenuto delle fonti è modellato con delle viste sullo schema intermedio. Come risultato, il problema di riformulare una query posta sullo schema intermedio in una query che fa riferimento agli schemi delle fonti, diventa il problema di rispondere a una query utilizzando le viste. Le soluzioni al problema di rispondere a query usando le viste divergono in questo contesto, perché il numero di viste (cioè le fonti) tende ad essere molto più grande, e le fonti non devono contenere le estensioni *complete* delle viste. Un'estensione di una vista non è altro che l'insieme di tuple da essa restituito.

Il problema di rispondere a query utilizzando le viste si pone in molte altre classi di applicazioni. Ad esempio nel contesto della progettazione dei database si possono definire delle viste per fornire un meccanismo atto a supportare l'indipendenza della *vista fisica* dei dati dalla *vista logica.* Questa indipendenza permette di modificare lo *schema fisico* (*storage schema*) senza cambiare il suo schema logico, e di modellare i tipi più complessi di indici. Un modo di approcciarsi a questo problema è descrivere lo *storage schema* come un insieme di viste sullo schema logico. In questa situazione, il calcolo del piano di esecuzione di una query (che, ovviamente, deve accedere alla memoria fisica) consiste nel capire come utilizzare le viste per rispondere alla richiesta. In un certo senso, il contesto dell'integrazione dei dati può essere visto come un caso estremo in cui c'è necessità di mantenere l'indipendenza fisica dei dati, dove il layout logico e il layout fisico sono stati definiti in anticipo.
Un'altra applicazione dove incontriamo nuovamente il problema di rispondere a query utilizzando le viste è la progettazione di Data Warehouse, dove si ha bisogno di scegliere una serie di viste (e indici sulle viste) da materializzare. Allo stesso modo, nella progettazione di siti web, la performance di un sito web può essere nettamente migliorata scegliendo un insieme di viste da materializzare. In entrambi questi problemi, il primo passo per determinare l'utilità della scelta delle viste è quello di garantire che esse siano sufficienti per rispondere alle interrogazioni che si aspetta di ricevere sulla

base di dati. Questo problema, ancora una volta, si traduce in un problema di riscrittura con le viste.
Infine, il problema di rispondere alle interrogazioni utilizzando le viste svolge un ruolo chiave nello sviluppo di metodi per il *semantic data caching* nei sistemi client-server. In questi lavori, i dati memorizzati nella cache del client sono semanticamente modellati come un insieme di viste, piuttosto che come una rappresentazione di livello fisico (insieme di pagine di dati o tuple). Quindi, decidere quali dati devono essere spediti dal server al fine di rispondere ad una data query, richiede di individuare quali parti della query possono essere risolte utilizzando le viste nella cache dei client.

Le numerose applicazioni del problema hanno stimolato la ricerca di algoritmi in diversi sistemi commerciali. Il problema è trattato differentemente in base al contesto operativo: principalmente *ottimizzazione delle query* oppure *integrazione dati*.
Nel caso di ottimizzazione delle query, l'attenzione è rivolta a produrre un piano di esecuzione di query che coinvolga le viste, e quindi lo sforzo è quello di estendere gli ottimizzatori di query in modo da sfruttare le viste. In questa situazione si punta al "*query rewriting*", cioè a riscrivere l'interrogazione posta sulla base di dati in modo da avere una riscrittura che sia *equivalente*, per avere in output un piano di esecuzione della query corretto. È importante notare che in questo caso, alcune delle viste contenute nel piano di esecuzione della query possono non contribuire alla correttezza logica del piano, ma solo a ridurre i costi di esecuzione.
Nel contesto dell'*integrazione dati* si parla invece di "*query answering*"; qui l'attenzione è rivolta a tradurre le query formulate sullo schema intermedio in query formulate sulle fonti di dati. Quindi, l'output dell'algoritmo è una query, e non un piano di esecuzione dell'interrogazione posta sullo schema logico. Poiché le fonti dati possono non coprire tutto il dominio, a volte ci si limita di una *query riscritta contenuta*, piuttosto che una riscrittura equivalente. Una *query riscritta contenuta* fornisce un sottoinsieme della risposta alla query, ma forse non l'intera risposta. Inoltre, i lavori sull'integrazione dati distinguono tra il caso in cui le viste sono *complete* (cioè contengono tutte le tuple

soddisfacenti la loro definizione) e il caso in cui esse possono essere *incomplete* (situazione comunemente incontrata quando si modella fonti di dati autonome).

Prima di iniziare, nel prossimo paragrafo, una discussione tecnica dettagliata del problema di rispondere a query usando le viste, si introduce il seguente schema per presentare alcune delle sue applicazioni attraverso degli esempi.

```
Studenti(nome_stud)
Area_ricerca(area_ricerca)
Atenei(ateneo)
Professori(nome_prof, area_ricerca)
Corsi(id_corso, titolo, ateneo)
Insegnamenti(nome_prof, id_corso, anno, num_corsisti, ateneo)
Frequenze(nome_stud, id_corso, anno)
Tesisti(nome_stud, nome_prof)
```

Si suppone che i professori, studenti, le aree di ricerca, e gli atenei siano identificati in modo univoco con i loro nomi, e che i corsi siano univocamente identificati dal loro numero. La relazione *Frequenze* associa gli studenti ai corsi seguiti, mentre la relazione *Tesisti* descrive da quale professore lo studente è seguito per il suo lavoro di tesi. Tale schema è ovviamente semplificato rispetto alla situazione reale di un ambiente universitario, ma in questo contesto è sufficiente per realizzare gli esempi.

5.1.1 Ottimizzazione delle query

La prima e più ovvia motivazione per considerare il problema di rispondere a query utilizzando le viste è l'ottimizzazione delle query. Se parte del calcolo necessario per rispondere a una query è già stato eseguito durante la materializzazione di una vista, allora può essere usata per velocizzare il calcolo dell’interrogazione.

Si consideri la seguente interrogazione, che chiede gli studenti dottorandi e i titoli dei corsi da loro seguiti associati a professori dell’area di ricerca “Basi di Dati” (negli esempi supponiamo che i corsi per studenti della laurea magistrale hanno *id* pari a 800 o

superiore, e i corsi per dottorandi hanno *id* pari a 1000 o superiore. Tutti gli altri corsi hanno *id* minore di 800):

```
SELECT Frequenze.nome_stud, Corsi.titolo
FROM Insegnamenti, Professori, Frequenze, Corsi
WHERE Professori.nome_prof = Insegnamenti.nome_prof
  AND Insegnamenti.id_corso = Frequenze.id_corso
  AND Insegnamenti.anno = Frequenze.anno
  AND Frequenze.id_corso = Corsi.id_corso
  AND Corso.id_corso ≥ 1000
  AND Professori.area_ricerca = "Basi di Dati"
```

Supponiamo che la seguente vista, contenente le informazioni relative ai corsi a livello di laurea magistrale o superiore, sia materializzata:

```
CREATE VIEW Laureandi AS
SELECT Frequenze.nome_stud, Corsi.titolo, Corsi.id_corso,
  Frequenze.anno
FROM Frequenze, Corsi
WHERE Frequenze.id_corso = Corsi.id_corso
  AND Corsi.id_corso ≥ 800
```

La vista *Laureandi* può essere utilizzata nel calcolo della query sopra come segue:

```
SELECT Laureandi.nome_stud, Laureandi.titolo
FROM Insegnamenti, Professori, Laureandi
WHERE Professori.nome_prof = Insegnamenti.nome_prof
  AND Insegnamenti.id_corso = Laureandi.id_corso
  AND Insegnamenti.anno = Laureandi.anno
  AND Laureandi.id_corso ≥ 1000
  AND Professori.area_ricerca = "Basi di Dati"
```

La valutazione della query riscritta sarà più conveniente perché la vista *Laureandi* ha già eseguito la giunzione tra *Frequenze* e *Corsi*, e ha già filtrato i corsi con id minore di 800 (che possono costituire buona parte della tabella *Corsi*). Si noti che la vista *Laureandi* è utile per risolvere la query, anche se non corrisponde *sintatticamente* a nessuna sottoparte della query originale.

Anche se la vista ha già calcolato parte della query, non è necessariamente vero che utilizzandola risulti in un piano di esecuzione più efficace, specialmente in presenza di indici disponibili sulle relazioni del database o sulle viste. Ad esempio, si supponga che le relazioni *Corsi* e *Frequenze* abbiano degli indici sull'attributo *id_corso*. In questo

caso, se la vista *Laureandi* non ha indici, allora valutare la query direttamente dalle relazioni del database potrebbe essere più conveniente. Quindi, la sfida non è solo quella di rilevare quando una vista è logicamente utilizzabile per rispondere a una query, ma anche dover prendere una decisione razionale basata sui costi.

5.1.2 Integrazione dati

Parte del lavoro al problema di rispondere a query usando le viste riguarda la sua applicabilità ai sistemi di *integrazione dati*. Esempi di tali lavori sono [15], [16], [17], [18], e [19]. Questi tipi di sistemi prevedono un'interfaccia di interrogazione *uniforme* per una moltitudine di fonti dati eterogenee e autonome. Esempi sono l'integrazione delle risorse a livello aziendale, l'interrogazione di fonti dati multiple sparse nel World-Wide Web, e l'integrazione di dati provenienti da esperimenti scientifici distribuiti. Le *fonti dati* in queste applicazioni possono essere database tradizionali, sistemi *legacy*, o anche file strutturati. L'obiettivo di un sistema di integrazione dati è di togliere l'onere all'utente di dover trovare le fonti dati rilevanti per una determinata query, interagire con ogni fonte in modo isolato, e manualmente combinare dati provenienti da diverse fonti.

Per fornire un'interfaccia uniforme, un sistema di integrazione dati espone all'utente uno *schema intermedio*. Tale schema, progettato manualmente per ogni particolare applicazione di integrazione dati, consiste in un insieme di relazioni *virtuali*, nel senso che esse non sono in realtà memorizzate da nessuna parte. Per essere in grado di rispondere a query, il sistema deve anche contenere una serie di *descrizioni delle fonti dati*. Una descrizione di una fonte dati specifica il contenuto della fonte, gli attributi che possono essere trovati in essa, e dei vincoli sui suoi contenuti.

Uno degli approcci per descrivere il contenuto di una fonte dati, adottato in diversi sistemi, è quello di descriverlo come una vista sullo schema intermedio (Figura 17).

Quest'approccio facilita l'aggiunta di nuove fonti di dati e la specifica dei vincoli sui contenuti delle fonti. Per rispondere a una query, un sistema di integrazione dati deve tradurre una query formulata sullo schema intermedio in una query che fa riferimento direttamente alle fonti dati. Poiché il contenuto delle fonti dati è descritto come viste sullo schema intermedio, il problema della traduzione è riportato al problema di rispondere a una query utilizzando le viste.

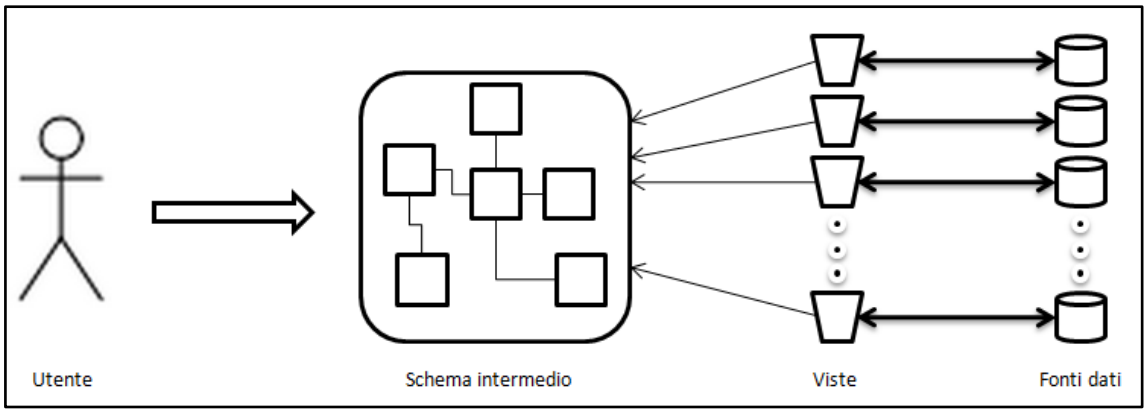

Figura 17: Viste definite sullo schema intermedio

Di seguito si illustra il problema con un esempio. Si supponga di avere due fonti dati; la prima fonte fornisce un elenco di tutti i corsi dal titolo "Sistemi Informativi" insegnati nell'ateneo *Corsi.ateneo* dal professore *Insegnamenti.nome_prof*. Questa fonte dati può essere descritta dalla seguente definizione di vista:

```
CREATE VIEW corsi_sisinfo AS
SELECT Corsi.titolo, Insegnamenti.nome_prof, Corsi.id_corso,
  Corsi.ateneo
FROM Insegnamenti, Corsi
WHERE Insegnamenti.id_corso = Corsi.id_corso
  AND Insegnamenti.ateneo = Corsi.ateneo
  AND Corsi.titolo = "Sistemi Informativi"
```

La seconda fonte dati contiene un elenco di tutti i corsi per dottorandi tenuti nell'ateneo "Federico II" di Napoli (F2), e può essere descritta dalla seguente vista:

```
CREATE VIEW corsi_f2 AS
SELECT Corsi.titolo, Insegnamenti.nome_prof, Corsi.id_corso,
  Corsi.ateneo
```

```
FROM Insegnamenti, Corsi
WHERE Insegnamenti.id_corso = Corsi.id_corso
  AND Corsi.ateneo = "F2"
  AND Insegnamenti.ateneo = "F2"
  AND Corsi.id_corso ≥ 1000
```

Se dovessimo chiedere al sistema di integrazione dati i corsi dal titolo "Sistemi Informativi" svolti presso l'ateneo Federico II di Napoli, esso sarebbe in grado di rispondere alla richiesta applicando una selezione della fonte dati descritta da *corsi_sisinfo*:

```
SELECT nome_prof
FROM corsi_sisinfo
WHERE ateneo = "F2"
```

D'altra parte, si supponga di chiedere al sistema tutti i corsi offerti dall'ateneo "F2", i quali non sono tutti contenuti nelle fonti dati disponibili. Siccome al momento ci sono solo due sorgenti dati, il sistema di integrazione dati non è in grado di trovare *tutte* le tuple per rispondere alla query. Si può tentare di trovare l'insieme massimo di tuple che soddisfano l'interrogazione basandosi esclusivamente sulle fonti dati disponibili. In particolare, il sistema può ottenere i corsi con titolo "Sistemi Informativi" dell'ateneo "F2" dalla fonte dati descritta dalla vista *corsi_sisinfo*, e i corsi per dottorandi ottenuti dalla fonte dati descritta da *coursi_f2*. Quindi, la query seguente fornisce l'insieme massimo di tuple che può essere ottenuto da queste due fonti dati:

```
SELECT titolo, id_corso
FROM corsi_sisinfo
WHERE ateneo = "F2"
  AND id_corso ≥ 800
UNION
SELECT titolo, id_corso
FROM corsi_f2
```

Si noti che non saranno presenti nella risposta i corsi che non sono per dottorandi e non sono intitolati "Sistemi Informativi". Ricordiamo che nel contesto di ottimizzazione delle query il focus è sulla ricerca di una riscrittura della query che sia *equivalente* alla query originale, qui invece si tenta di trovare un'espressione di query che fornisce le

risposte massimali utilizzando le viste, attraverso una riscrittura contenuta. Queste nozioni sono formalizzate nel paragrafo 5.2.

5.1.3 Altre applicazioni

Altri lavori al problema di rispondere a query usando le viste puntano all'obiettivo di mantenere l'indipendenza fisica dei dati in database relazionali (quali[19] e [20]), o database object-oriented (come quello descritto in[21]). Infatti uno dei principi alla base di questi sistemi è la separazione tra la vista logica dei dati (cioè le tabelle con i loro nomi di attributi) e la vista fisica dei dati (come i dati sono posizionati sul disco). Non considerando situazioni di partizionamento orizzontale o verticale delle relazioni in più file, svariati DBMS relazionali sono basati su una corrispondenza 1-a-1 tra le relazioni nello schema logico e i file in cui si trovano memorizzati i dati. In sistemi object-oriented, mantenere questa separazione è necessario perché lo schema logico contiene una ridondanza significativa, e non corrisponde a un buon layout fisico.
Mantenere l'indipendenza fisica dei dati è ancora più importante nelle applicazioni in cui il modello logico introdotto è definito su una rappresentazione fisica già esistente. Questo è il caso di applicazioni che operano con dati semi-strutturati, oppure in contesti di integrazione dati. In un certo senso l'integrazione dati può essere vista come un caso estremo in cui si vuole mantenere l'indipendenza fisica dei dati dove il layout logico e il layout fisico sono stati definiti in anticipo.
Ancora una volta incontriamo il problema di rispondere alle interrogazioni usando le viste in quanto, per mantenere l'indipendenza fisica dei dati, diversi lavori sul tema propongono di utilizzare le viste come un meccanismo per descrivere l'archiviazione dei dati.

Un altro campo dove si pone il problema di rispondere a query usando le viste è la progettazione di Data Warehouse (come in [22], [23], [24], e [25]) e il data caching semantico (come in [26]). Nella progettazione di Data Warehouse, quando si sceglie un

insieme di viste da materializzare, si ha bisogno di controllare di essere in grado di rispondere alle interrogazioni utilizzando le queste viste laddove è possibile. Nel contesto dal data caching semantico, si ha bisogno di verificare se i risultati di una query effettuata precedentemente possono essere utilizzati per una nuova query, oppure se il client ha bisogno di chiedere dati aggiuntivi al server. Inoltre materializzare le viste può accelerare notevolmente i tempi di risposta da siti web (come in [27]), e quindi si è nuovamente di fronte al problema di rispondere alle query utilizzando le viste.

5.2 Definizioni utili

In questo paragrafo si riporta parte della terminologia base utilizzata in [28], per introdurre formalmente il problema di rispondere alle query usando le viste. I concetti di *contenimento in una query* e di *equivalenza a una query* forniscono una base per il confronto semantico tra le query e le loro riscritture.

La maggior parte della discussione sarà incentrata sulla classe di query del tipo *selezione-proiezione-giunzione* su database relazionali. Una vista non è altro che una query alla quale è stato associato un nome, ed è detta materializzata se il suo risultato è memorizzato nella base di dati. Un'istanza di database è un'assegnazione di un'estensione (cioè un insieme di tuple) a ogni sua relazione. E' data per scontata la conoscenza del linguaggio SQL, ampiamente descritto in [7]. Si distinguono le query che coinvolgono predicati di confronto (per esempio, $\leq$, $<$, $=$) da quelle che non lo fanno. Si indica con *Q(D)* il risultato del calcolo della query *Q* sul database *D*. Quando si tratta con query *Q* che fanno riferimento a viste (ad esempio interrogazioni che hanno subito una riscrittura), allora *Q(D)* si riferisce al risultato del calcolo *Q* dopo che le viste sono state calcolate su *D*.

È importante notare che le definizioni di seguito si applicano sia a query che fanno riferimento esclusivamente alle relazioni del database, sia a query che possono far

riferimento a viste. Inoltre si suppone una semantica insiemistica (set di tuple), anche se le definizioni possono essere estese in modo diretto alla semantica “bag di tuple”.

5.2.1 Contenimento ed equivalenza

Le nozioni di *contenimento* e di *equivalenza* consentono il confronto tra le varie riformulazioni di una query e per provare la correttezza di una riscrittura di una query usando un determinato insieme di viste.

Definizione 5.1. *Contenimento ed equivalenza*: una query Q_1 è detta contenuta in una query Q_2, denotato con $Q_1 \sqsubseteq Q_2$, se per tutte le istanze del database D, l'insieme di tuple calcolato per Q_1 è un sottoinsieme di quello calcolato per Q_2, vale a dire, $Q_1(D) \subseteq Q_2(D)$. Le due interrogazioni sono detti equivalenti se $Q_1 \sqsubseteq Q_2$ e $Q_2 \sqsubseteq Q_1$.

In questo lavoro si considerano interrogazioni del tipo selezione-proiezione-giunzione, interrogazioni con predicati aritmetici di confronto, e interrogazioni con semantica “bag di tuple”.

5.2.2 Riscrittura equivalente e contenuta

Nel linguaggio SQL una query fa riferimento esclusivamente alle viste se tutte le relazioni indicate nella clausola FROM sono delle viste. Si definisce *riscrittura completa* una riscrittura che fa riferimento esclusivamente alle viste, e non alle relazioni dello schema.
Data una query Q e un insieme di viste $V_1,\ldots,V_m$, una riscrittura completa dell’interrogazione è una query Q' che si riferisce *solo* alle viste $V_1,\ldots,V_m$.

Nell’ottimizzazione delle interrogazioni si può anche essere interessati in riscritture che facciano riferimento sia a relazioni del database, sia a viste materializzate.

Concettualmente riscritture di questo tipo non introducono un nuovo grado di difficoltà, perché si può sempre simulare il caso precedente creando delle viste che rispecchiano esattamente ogni tabella del database.

Come accennato visto nel paragrafo 5.1, c'è bisogno di distinguere due tipi principali di riscritture: *riscritture equivalenti* e *riscritture massimamente-contenute*.
Le riscritture massimamente-contenute sono considerate nell'ambito dell'integrazione dati. Questo tipo di riscrittura varia a seconda del linguaggio di interrogazione utilizzato per la riscrittura, e di conseguenza la seguente definizione dipende dal particolare linguaggio:

Definizione 5.2. *Riscritture massimamente-contenute*: Sia Q una query, $V=\{V_1,\ldots,V_m\}$ un insieme di viste, e L un linguaggio di interrogazione. La query Q' è una riscrittura massimamente-contenuta di Q che utilizza le viste di V nel rispetto del linguaggio L se:

- Q' è una query scritta nel linguaggio L che fa riferimento solo alle viste in V,
- $Q' \sqsubseteq Q$, e
- non c'è un'altra riscrittura $Q'' \in L$, tale che:
  - $Q' \sqsubseteq Q'' \sqsubseteq Q$, e
  - Q'' non è equivalente a Q'.

Quando una riscrittura Q' è contenuta in Q, ma non è un massimamente-contenuta allora ci si riferisce a essa come una *riscrittura contenuta*.

Per l'ottimizzazione delle query invece si considerano esclusivamente le riscritture equivalenti.

Definizione 5.3. *Riscritture equivalenti*: Sia Q una query e $V=\{V_1,\ldots,V_m\}$ un insieme di viste. La query Q' è una riscrittura equivalente di Q che usa le viste V se:

- Q' è una riscrittura completa che fa riferimento solo alle viste in V, e
- Q' è equivalente a Q, cioè: $Q \sqsubseteq Q'$ e $Q' \sqsubseteq Q$.

Notiamo che gli algoritmi per il contenimento di query e l'equivalenza di query forniscono metodi per *testare* se una riscrittura candidata di una query è equivalente o contenuta. Si noti, inoltre, l'indipendenza dal particolare linguaggio di interrogazione delle definizioni date.

5.3 Condizioni per utilizzare una vista nella riscrittura

La domanda fondamentale che ci si pone quando si lavora al problema di rispondere alle query usando le viste, è quella di sapere quando una vista è utilizzabile per elaborare la risposta. In questo paragrafo si presentano degli esempi in modo da illustrare le condizioni che devono valere per le quali una vista è utilizzabile nel rispondere a una query, e in che modo la vista può essere utilizzata. Se non diversamente specificato, si suppone di operare con query di tipo selezione-proiezione-giunzione e con sotto semantica insiemistica (*set* di tuple).
Nel linguaggio SQL, per query di tipo *selezione-proiezione-giunzione*, anche dette query di tipo *select-project-join* (*SPJ*), si intendono interrogazioni costituite da un singolo blocco di tipo:

```
SELECT …
FROM …
[WHERE …]
```

Questo tipo di interrogazioni non possono contenere sotto-query, e contengono esclusivamente le clausole SELECT, FROM, e la clausola WHERE opzionale.

Detto in modo informale, una vista può essere utile nel rispondere a una query se l'insieme di relazioni che utilizza si sovrappone con quello della query da riscrivere, e alcuni attributi che seleziona sono attributi selezionati anche dalla query. Inoltre, se la query e la vista richiedono il soddisfacimento di predicati sugli attributi in comune, allora la vista deve applicare dei predicati pari o logicamente "più deboli" in modo da poter avere una riscrittura equivalente. Se invece la vista applica predicati logicamente "più forti", si può avere solamente una riscrittura contenuta. Un predicato P è detto più

debole di un predicato P' se l'insieme delle tuple che soddisfano il predicato P' è contenuto nell'insieme delle tuple che soddisfano il predicato P'.

Come esempio si consideri la seguente query, che chiede tuple del tipo <*nome_prof*, *nome_stud*, *anno*> nel quale lo studente nome_stud ha seguito un corso tenuto dal professore *nome_prof* durante l'anno 2008 o successivamente.

```
SELECT Tesisti.nome_prof, Tesisti.nome_stud, Frequenze.anno
FROM Frequenze, Insegnamenti, Tesisti
WHERE Frequenze.id_corso = Insegnamenti.id_corso
  AND Frequenze.anno = Insegnamenti.anno
  AND Tesisti.nome_prof = Insegnamenti.nome_prof
  AND Tesisti.nome_stud = Frequenze.nome_stud
  AND Frequenze.anno ≥ 2008
```

La seguente vista *View1* è utilizzabile perché applica le stesse condizioni di join per le relazioni *Frequenze* e *Insegnamenti*. Quindi possiamo usarla per rispondere all'interrogazione facendo la giunzione con la relazione *Tesisti*. Inoltre la vista *View1* seleziona i campi *Frequenze.nome_stud*, *Frequenze.anno*, e *Insegnamenti.nome_prof* necessari per la giunzione con la relazione *Tesisti* e per la selezione della clausola *SELECT* della query. Infine nella vista si applica il predicato *Frequenze.anno≥2007*, che è più debole rispetto al predicato della query *Frequenze.anno≥2008*, e siccome *View1* ha tra gli attributi di proiezione *Frequenze.anno*, il predicato più forte può essere applicato nella riscrittura.

```
CREATE VIEW View1 AS
SELECT Frequenze.nome_stud, Insegnamenti.nome_prof,
Frequenze.anno
FROM Frequenze, Insegnamenti
WHERE Frequenze.id_corso = Insegnamenti.id_corso
  AND Frequenze.anno = Insegnamenti.anno
  AND Frequenze.anno ≥ 2007
```

Le seguenti viste mostrano come lievi modifiche alla vista *View1* possono cambiare l'usabilità della vista stessa nel rispondere alla query:

```
CREATE VIEW View2 AS
SELECT Frequenze.nome_stud, Frequenze.anno
FROM Frequenze, Insegnamenti
```

```
WHERE Frequenze.id_corso = Insegnamenti.id_corso
  AND Frequenze.anno = Insegnamenti.anno
  AND Frequenze.anno ≥ 2008

CREATE VIEW View3 AS
SELECT Frequenze.nome_stud, Insegnamenti.nome_prof,
Frequenze.anno
FROM Frequenze, Insegnamenti
WHERE Frequenze.id_corso = Insegnamenti.id_corso
  AND Frequenze.anno ≥ 2007

CREATE VIEW View4 AS
SELECT Frequenze.nome_stud, Frequenze.anno,
Insegnamenti.nome_prof
FROM Frequenze, Insegnamenti, Tesisti, Professori
WHERE Frequenze.id_corso = Insegnamenti.id_corso
  AND Frequenze.anno = Insegnamenti.anno
  AND Insegnamenti.nome_prof = Tesisti.nome_prof
  AND Professori.area_ricerca IS NOT NULL
  AND Frequenze.anno ≥ 2008

CREATE VIEW View5 AS
SELECT Frequenze.nome_stud, Insegnamenti.nome_prof,
Frequenze.anno
FROM Frequenze, Insegnamenti
WHERE Frequenze.id_corso = Insegnamenti.id_corso
  AND Frequenze.anno = Insegnamenti.anno
  AND Frequenze.anno ≥ 2009
```

La vista *View2* è simile a *View1*, salvo per il fatto che essa non seleziona l'attributo *Insegnamenti.nome_prof*, necessario per effettuare la giunzione con la relazione *Tesisti* e per essere proiettato nella clausola SELECT della query. Quindi, per usare *View2* nella riscrittura, avremmo bisogno di fare una ulteriore giunzione della vista *View2* con la relazione *Insegnamenti* (in aggiunta al giunzione fatta con la relazione *Tesisti*). In questo caso possiamo avere un piano di esecuzione della query relativamente veloce se la giunzione delle relazioni *Frequenze* e *Insegnamenti* è molto selettiva.

La vista *View3* non effettua l'equi-join giacché non confronta l'uguaglianza degli attributi *Frequenze.anno* e *Insegnamenti.anno*. Poiché gli attributi *Frequenze.anno* e *Insegnamenti.anno* non sono entrambi selezionati dalla vista *View3*, il predicato di join della query da riscrivere non può essere applicato.

View4, invece, considera solo i professori che hanno almeno un'area di ricerca. Quindi, la vista applica una condizione aggiuntiva che non è chiesta nella query da riscrivere, dunque, la vista non può essere utilizzata in un riscrittura equivalente a meno che non permettiamo le operazioni di unione e la negazione nel linguaggio di interrogazione. Tuttavia, in presenza di un vincolo di integrità nello schema concettuale del database il quale afferma che ogni professore ha almeno un area di ricerca, allora un buon ottimizzatore dovrebbe essere in grado di rendersi conto che la vista *View4* è utilizzabile per una riscrittura equivalente.
Infine, la vista *View5* applica un predicato più forte di quello applicato nella query da riscrivere (*Frequenze.anno≥2009*), ed è quindi utilizzabile per una riscrittura contenuta (o massimamente-contenuta), ma non per una riscrittura equivalente della query.

5.3.1 Condizioni per query e viste di tipo SPJ

Elenchiamo adesso le condizioni che devono valere per una vista *V* di tipo *selezione-proiezione-giunzione* (*SPJ*) per essere utilizzabile in una riscrittura equivalente di una query *Q*. Le condizioni intuitive riportate di seguito possono essere rese formali una volta fissato lo specifico linguaggio di interrogazione:

1. Ci deve essere una corrispondenza φ tra le occorrenze delle tabelle menzionate nella clausola *FROM* di *V* a quelle menzionate nella clausola *FROM* di *Q*, questa corrispondenza mappa ogni nome di tabella a se stesso. Nel caso di semantica "bag di tuple", φ deve essere una corrispondenza 1-a-1, mentre nel caso di semantica "set di tuple", φ può essere una corrispondenza molti-a-1.
2. *V* deve applicare gli stessi predicati di join e di selezione che sono applicati in *Q* sugli attributi delle tabelle nel dominio di φ, oppure deve applicare dei predicati logicamente più deboli e selezionare gli attributi su cui questi predicati sono applicati in modo da poter riapplicare un predicato logicamente più forte.
3. *V* deve obbligatoriamente proiettare gli attributi delle tabelle nel dominio di φ che sono necessari per la selezione di *Q*, a meno che questi attributi possano essere recuperati in altro modo (ad esempio da un'altra vista o dalla relazione del database).

Notiamo che l'introduzione della semantica “bag di tuple” introduce complicazioni aggiuntive poiché dobbiamo assicurarci che la molteplicità delle tuple nella risposta alla query non si perda usando le viste. Questo è il caso del linguaggio SQL, dove si può avere la semantica insiemistica solo nel caso in cui sia specificata la clausola DISTINCT.

5.3.2 Query con operazioni di raggruppamento e aggregazione

Supponiamo adesso interrogazioni con operazioni di raggruppamento e aggregazione. Nel caso del linguaggio SQL questo tipo di interrogazioni sono costituite da un singolo blocco di tipo:

```
SELECT <attributi_raggruppamento>,[<operazioni_di_aggregazione>]
FROM …
[WHERE …]
GROUP BY <attributi_raggruppamento>
[HAVING …]
```

Queste query sono dette di tipo *selezione-proiezione-giunzione-raggruppamento*, o anche query di tipo *select-project-join-groupby* (*SPJG*).
Anche in questo caso, a causa della semantica “bag di tuple”, una vista è utilizzabile per rispondere a una query solo se esiste una corrispondenza 1-a-1 tra le relazioni menzionate nella clausola FROM della vista e un sottoinsieme delle relazioni menzionate dalla clausola FROM della query.

Nelle applicazioni di supporto alle decisioni, quando le query contengono il raggruppamento e l’aggregazione, vi è più di una possibilità di ottenere significativi incrementi nella velocità riutilizzando i risultati delle viste materializzate. Tuttavia, la presenza delle operazioni di raggruppamento e aggregazione nelle query o nelle viste introduce numerose nuove difficoltà al problema di rispondere alle query utilizzando le viste.
La prima difficoltà che sorge ha a che fare con le colonne aggregate. Ricordiamo che una vista è utilizzabile da una query se proietta gli attributi necessari alla query (che non

sono altrimenti recuperabili). Quando una vista esegue un'aggregazione su un attributo, perdiamo alcune informazioni riguardanti l'attributo stesso, e in un certo senso possiamo dire che lo proietta solo in modo *parziale*. In generale possiamo dire che una vista V è utilizzabile per rispondere a una query Q se la query richiede lo stesso raggruppamento o un raggruppamento più "grossolano" di quello specificato nella vista, e la colonna di aggregazione è disponibile o può essere ricostruita.
La seconda difficoltà che sorge è legata alla perdita della molteplicità dei valori sugli attributi su quali è eseguito il raggruppamento. Quando raggruppiamo su di un attributo *A*, si perde la molteplicità dell'attributo nei dati, perdendo così la capacità di effettuare successive operazioni quali somma, conteggio, o media. Vedremo alcuni casi in cui potrebbe essere possibile recuperare la molteplicità utilizzando informazioni aggiuntive.

Illustriamo adesso queste difficoltà che sorgono in presenza delle operazioni di raggruppamento e aggregazione attraverso un esempio. Supponiamo di avere la seguente vista *corsi_anno* a disposizione, la quale considera tutti i corsi per dottorandi o studenti delle lauree magistrali, e per ogni coppia <*id_corso, anno*> calcola il numero di studenti massimo avuto per quel corso durante l'anno, e il numero di volte che il corso è stato offerto (si suppone in questo esempio che lo stesso corso può essere tenuto più volte durante l'anno).

```
CREATE VIEW corsi_anno AS
SELECT id_corso, anno, Max(num_corsisti) AS max_corsisti,
  COUNT(*) AS offerte
FROM Insegnamenti
WHERE id_corso ≥ 800
GROUP BY id_corso, anno
```

La query seguente considera solo i corsi per dottorandi, e chiede il numero di corsisti massimo per qualsiasi corso in un determinato anno, e il numero di offerte dei corsi:

```
SELECT anno, MAX(num_corsisti), COUNT(*)
FROM Insegnamenti
WHERE id_corso ≥ 500
GROUP BY anno
```

Per rispondere alla query possiamo sfruttare la vista *corsi_anno* come nella seguente riscrittura:

```
SELECT anno, MAX(max_corsisti), SUM(offerte)
FROM corsi_anno
WHERE id_corso ≥ 500
GROUP BY anno
```

Ci sono dei punti da notare nella riscrittura. In primo luogo, anche se la vista effettuata un'operazione di aggregazione sull'attributo *num_corsisti*, si può ancora utilizzare la vista nel riscrivere la query, perché il raggruppamento nella query (su *anno*) è più grossolano di quello della vista (su *id_corso,anno*). Quindi, la risposta alla query può essere ottenuta fondendo gruppi della vista. In secondo luogo, dal momento la vista raggruppa le risposte per *id_corso*, e perde così la molteplicità di ogni corso, normalmente non dovremmo essere in grado di utilizzare la vista per calcolare il numero di corsi offerti durante l'anno. Tuttavia, poiché la vista include anche l'attributo *offerte*, ci fornisce informazioni sufficienti al fine di recuperare il numero totale di corsi offerti durante l'anno, sommando le offerte per tutti i corsi.

Tra i lavori esaminati, [29], [30], e [31],considerano il problema di rispondere a query usando le viste anche in presenza di operazioni di raggruppamento e aggregazione. Una strada possibile, discussa in [30], è quella di adottare un approccio puramente sintattico il quale coinvolge una serie di trasformazioni nella fase di riscrittura. In questo caso l'algoritmo esegue una serie di trasformazioni sintattiche sulla query fino a quando non è possibile individuare una sottoespressione della query che è identica alla vista, in modo da poter sostituire la sottoespressione stessa con la vista. Questo modo di operare è ovviamente limitativo poiché spesso una vista utilizzabile per rispondere a una query non corrisponde sintatticamente a una sua sottoespressione.
Si può dunque adottare un approccio più semantico come quello proposto in [32]. In questo caso si individua un insieme di condizioni necessarie per una vista al fine di essere utilizzabile per rispondere a una query in presenza di operazioni di raggruppamento e aggregazione, e successivamente l'algoritmo di riscrittura utilizza

queste condizioni. In questo modo si riesce a individuare l'utilizzabilità di una vista per rispondere a un'interrogazione anche quando non si ha una precisa corrispondenza sintattica con una sottoespressione della query.

È interessante notare che quando la vista contiene operazioni di raggruppamento e aggregazione e la query non le contiene, allora la vista non è utilizzabile dalla query salvo che essa non rimuova i duplicati nella clausola *SELECT*.

5.4 Classificazione degli approcci al problema

Come illustrato nei precedenti esempi, ci sono vari contesti nei quali possiamo classificare i diversi modi di trattare il problema di rispondere alle query utilizzando le viste. In questo paragrafo si illustrano le differenze tra di loro proponendo una classificazione dei diversi lavori che evidenzia le principali differenze.

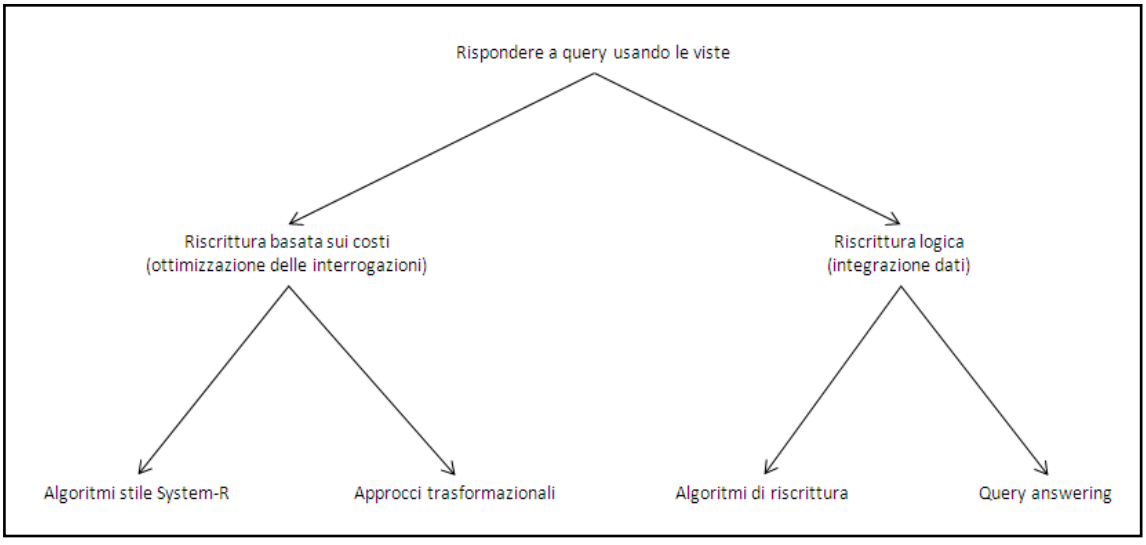

Figura 18: Tassonomia dei lavori sul problema di rispondere a query usando le viste

Come illustrato dallo schema in Figura 18 la distinzione più significativa tra i diversi lavori è l'obiettivo che si vuole raggiungere: può essere l'integrazione dei dati (riscrittura della query di tipo logico) oppure l'ottimizzazione delle interrogazioni

(riscrittura della query basata sui costi). La differenza fondamentale tra queste due classi di lavori è l'output dell'algoritmo. Nel primo caso, data una query *Q*, e un insieme di viste *V*, l'obiettivo dell'algoritmo è di produrre una riscrittura della query *Q'* che fa riferimento alle viste ed è equivalente o contenuta in *Q*. Nel secondo caso, l'algoritmo deve andare oltre e produrre un (auspicabilmente ottimale) piano di esecuzione della query per rispondere a *Q* utilizzando le viste (e forse anche le relazioni del database). In questo caso la riscrittura deve essere equivalente a *Q* al fine di garantire la correttezza del piano di esecuzione.
In entrambi i casi ci si chiede se la riscrittura della query sia equivalente o contenuta nella query stessa. Per il contesto dell'integrazione dati la correttezza logica della riscrittura è sufficiente, mentre nel contesto dell'ottimizzazione delle interrogazioni non basta. Infatti nel secondo caso c'è bisogno di trovare il piano di accesso *più conveniente* che può utilizzare le viste, e c'è un'ulteriore complicazione derivante dal fatto che gli algoritmi di ottimizzazione dovrebbero considerare anche le viste che non contribuiscono alla correttezza *logica* della riscrittura, ma solo a ridurre il costo del piano di esecuzione della query. Per questo motivo nel caso di ottimizzazione delle interrogazioni si necessita di una riscrittura basata sui costi.

Un'altra differenza riguarda il numero di viste disponibili nella base di dati. Nel contesto dell'integrazione dati è importante avere un algoritmo capace di trattare un gran numero di viste, che corrispondono alle diverse fonti dati. Nel contesto dell'ottimizzazione delle query si ritiene invece che generalmente che il numero di viste sia approssimativamente paragonabile alle dimensioni dello schema; questo può non essere vero, in particolar modo in contesti dove sono presenti strumenti automatici per la creazione di viste materializzate.

Una prima famiglia di algoritmi per l'integrazione dati tenta di sviluppare algoritmi di riscrittura scalabili fino a un gran numero di viste (come quelli descritti in [19], [15], e [18]), poiché vi è una possibile presenza di una moltitudine di fonti dati. Una seconda famiglia di algoritmi, invece, considera le diverse proprietà delle sorgenti dati (esempi

di tali lavori sono [16] e [17]). Ad esempio, si può distinguere tra fonti dati complete e incomplete: è opportuno osservare come, nel caso di fonti dati complete (cioè fonti dati contenenti tutte le tuple che soddisfano la loro definizione), il problema di rispondere alle query utilizzando le viste diventa computazionalmente più difficile. Il motivo intuitivo della difficoltà aggiunta è che, quando le fonti dati sono complete, possiamo dedurre informazioni negative effettuando particolari tipi di interrogazioni. In questo contesto il problema di rispondere alle interrogazioni usando le viste è anche detto *Query Answering Problem*.

Nel contesto dell'ottimizzazione delle query possiamo distinguere tra ottimizzatori stile System-R e ottimizzatori di tipo trasformazionale (o a regole): di conseguenza possiamo classificare i lavori in altre due famiglie di algoritmi per la riscrittura di query con viste materializzate. Entrambe le famiglie di algoritmi sono discusse nel prossimo paragrafo.
Si noti, inoltre, che nel contesto dell'ottimizzazione delle query, si assume sempre che le viste siano complete, e si chiedono esclusivamente riscritture *equivalenti* della query.

Un'altra dimensione di confronto di questi lavori è il linguaggio specifico utilizzato per esprimere le query e le viste. Molti algoritmi proposti si limitano al caso di query del tipo *selezione-proiezione-giunzione*, altri lavori invece si specializzano in particolari estensioni del linguaggio come: espressioni di tipo SPJG, viste con operatori insiemistici (ad esempio *UNION*), interrogazioni su dati semistrutturati, semantica "bag di tuple", OQL, espressioni SPJ multi-blocco, e altri.

Alcuni problemi strettamente connessi al problema di rispondere alle interrogazioni usando le viste, non sono trattati in questo lavoro. In particolare vi è il problema della manutenzione incrementale delle viste materializzate e il problema della selezione automatica di indici per esse. Inoltre non si tratterà ulteriormente il contesto dell'integrazione dati, per il quale sono stati proposti svariati algoritmi in letteratura come il *bucket-algorithm*, l'*inverse-rules*, e l'algoritmo *Minicon*.

Dunque dal prossimo paragrafo la discussione è incentrata sul problema di utilizzare le viste materializzate per l'ottimizzazione delle interrogazioni.

5.5 Usare le viste materializzate per l'ottimizzazione delle query

Si esamina in questo paragrafo i diversi algoritmi proposti per incorporare l'uso di viste materializzate nell'ottimizzazione delle interrogazioni. L'obiettivo di questi algoritmi è di decidere quando utilizzare le viste per rispondere a una query. Essi restituiscono in output un piano di esecuzione per l'interrogazione.

Possiamo distinguere due tipi di approcci che differiscono a seconda della fase in cui si modifica l'algoritmo di ottimizzazione delle query per considerare le viste materializzate: il primo prevede una radicale modifica dell'ottimizzatore stile System-R, mentre il secondo isola la parte relativa alla riscrittura con viste in un modulo dell'ottimizzatore. Quest'ultimo approccio è spesso adottato quando si tratta con un sottoinsieme più esteso del linguaggio SQL, come ad esempio query e viste del tipo SPJG.

Nella sezione 4.5.1 sono descritti gli algoritmi per ottimizzatori stile System-R, dove le viste materializzate sono considerate durante l'intera fase di *join enumeration*. Nella Sezione 4.5.2 sono invece descritti lavori basati su di ottimizzatori di tipo trasformazionale (anche detti ottimizzatori a regole). In quest'ultimo caso l'idea chiave è adottare regole di riscrittura che puntano a sostituire sottoespressione della query con sottoespressioni equivalenti che coinvolgono le viste.

5.5.1 Ottimizzazione stile System-R

Per ottimizzare l'esecuzione di un'interrogazione, un ottimizzatore stile Sytem-R è dotato di un modulo di *join enumeration* che ha il compito di esplorare possibili piani di esecuzione scegliendo quello per il quale si stima il costo inferiore. In questa sezione si

discutono le modifiche da apportare a un algoritmo di *join enumeration* al fine di includere l'ottimizzazione che utilizza le viste materializzate, supponendo di trattare esclusivamente con query del tipo *selezione-proiezione-giunzione* (*SPJ*).

Questo tipo di ottimizzatore affronta dunque un problema appartenete alla classe di complessità NP, e per risolverlo in modo rapido adotta tecniche di tipo euristico. Un'euristica spesso utilizzata è quella di scegliere esclusivamente piani di accesso profondi a sinistra (*left-deep*). Infatti, in presenza di una giunzione di *n* relazioni, possiamo combinarle in $\frac{(2(n-1))!}{(n-1)!}$ modi possibili. Considerando solo le combinazioni che possono essere descritta da un albero *left-deep*, si riduce drasticamente questo numero, e, effettuando le giunzioni sempre con una relazione della base di dati, aumenta lo spazio degli algoritmi alternativi utilizzabili, poiché si possono utilizzare gli indici per le giunzioni.

Prima di illustrare le possibili modifiche a un ottimizzatore stile System-R per considerare l'uso di viste materializzate, ricordiamo brevemente i principi alla base di questo tipo di ottimizzatore. Un sistema stile System-R adotta un approccio di tipo *bottom-up* per la costruzione dei piani di esecuzione delle interrogazioni. Nella prima fase, costruisce piani di dimensione 1, cioè sceglie i percorsi di accesso migliori per ogni singola tabella menzionata nella clausola *FROM* della query. Nella fase *n*, l'algoritmo considera piani di dimensione *n*, combinando coppie di piani ottenuti nelle fasi precedenti. L'algoritmo termina dopo la costruzione di piani che coprono tutte le relazioni utilizzate nella query. Nel caso si adotti l'euristica di considerare solo i piani di esecuzione descrivibili da alberi profondi a sinistra, allora durante l'iterazione *n-esima* si combinano i piani di dimensione *n-1* con piani di dimensione *1*. In caso contrario si considerano le combinazioni di tutti i piani di dimensione *k* con i piani di dimensione *n-k*.
Possiamo descrivere le varie iterazioni di un ottimizzatore stile Sytem-R in questo modo:

```
Iterazione 1:
   • Trovare tutti i possibili piani di esecuzione,
   • Confrontare i costi e selezionare il piano di accesso meno
     costoso,
   • Se l'interrogazione ha una sola relazione, allora STOP.

Iterazione 2:
   • Per ogni tabella coinvolta in un'operazione di giunzione
     non ancora considerata:
        o Considera la giunzione della tabella con i piani di
          esecuzione costruiti nell'iterazione precedente,
        o Confrontare i costi e selezionare il piano di
          accesso meno costoso.
   • Se l'interrogazione ha altre relazioni da considerare,
     allora STOP.

Iterazione 3:
   • …
```

L'efficienza degli ottimizzatori stile System-R deriva dalla partizione dei piani di esecuzione delle query in *classi di equivalenza*, e per ognuna di esse si considera un solo piano di esecuzione. Due piani sono nella stessa classe di equivalenza se valgono le seguenti condizioni:

1. coprono lo stesso insieme di relazioni della query (e quindi sono anche della stessa dimensione);
2. producono le risposte nello stesso ordine di interesse.

Nel processo di costruzione dei piani, due piani sono uniti allo scopo di formarne un altro, soltanto se essi coprono due insiemi disgiunti di relazioni.

Nel contesto dell'ottimizzazione delle interrogazioni con le viste materializzate, l'ottimizzatore costruisce i piani di esecuzione di query mediante l'accesso a un insieme di viste, piuttosto che un insieme di relazioni del database. Pertanto l'ottimizzatore ha come input anche le espressioni che definiscono le viste materializzate, e i metadati relativi a esse (ad esempio le statistiche e gli indici). L'algoritmo di seguito descritto calcola il piano di esecuzione per una riscrittura completa dell'interrogazione. Siccome una relazione del database può essere modellata come una vista, l'algoritmo resta valido

anche nel caso si permettano riscritture che fanno riferimento a relazioni della base di dati.

Un possibile algoritmo per un ottimizzatore stile System-R che considera l'uso di viste materializzate, descritto in [33], effettua queste operazioni:

Iterazione 1:
- Trovare tutte le viste rilevanti per rispondere all'interrogazione,
- Distinguere tra soluzioni parziali e complete per avere la risposta,
- Confrontare tutte le coppie di viste, e scartare quelle che non portano grossi contributi o hanno un costo alto,
- Se non ci sono altre soluzioni parziali, allora STOP.

Iterazione 2:
- Considerare tutti i possibili modi di effettuare le giunzioni tra le soluzioni parziali individuate nell'iterazione precedente,
- Distinguere tra soluzioni parziali e complete per avere la risposta,
- Scartare le soluzioni individuate che non sono rilevanti per l'interrogazione o per le quali è già stata individuata una soluzione migliore,
- Se non ci sono altre soluzioni parziali, allora STOP.

Iterazione 3:
- …

Per effettuare le operazioni descritte, l'ottimizzatore necessita di ulteriori informazioni non fornite a un ottimizzatore stile System-R convenzionale.
Per prima cosa, durante la prima iterazione l'algoritmo decide quali viste sono *rilevanti* per la query, ovvero se è utilizzabile per rispondere alla query (le condizioni sono illustrate nel paragrafo 4.3). Il passo corrispondente in un ottimizzatore tradizionale è banale: una relazione è rilevante per la query se è indicata nella clausola *FROM*.
Inoltre, giacché i piani di esecuzione delle query coinvolgono giunzioni su più viste, i piani di esecuzione non possono più essere nettamente divisi in classi di equivalenza poiché viste diverse possono far riferimento alle stesse relazioni della base di dati. Questa osservazione comporta diverse modifiche all'algoritmo tradizionale:

1. **Test di terminazione:** l'algoritmo ha bisogno di distinguere i *piani di esecuzione parziali* della query dai *piani di esecuzione completi* (producono una risposta equivalente alla risposta prodotta eseguendo l'interrogazione originale). L'enumerazione dei possibili join termina quando non ci sono più piani di esecuzione parziali inesplorati. Al contrario, nell'algoritmo tradizionale la fase di enumerazione termina dopo aver esaminato le classi di equivalenza che comprendono tutte le relazioni menzionate nella query.
2. **Potatura dei piani:** un ottimizzatore tradizionale confronta coppie di piani di esecuzione all'interno della stessa classe di equivalenza e salva solo quello più economica. Nell'algoritmo qui esaminato l'ottimizzatore deve confrontare tutte le possibili coppie di piani generati fino a quel momento. Un *p* piano è eliminato se c'è un altro piano *p'* il quale è più conveniente di *p*, e apporta un contributo alla query uguale o superiore a quello di *p*. Informalmente, un piano *p'* contribuisce di più alla query rispetto al piano *p*, se si copre un numero maggiore di relazioni e seleziona un numero maggiore di attributi necessari.
3. **La combinazione di piani parziali:** nell'algoritmo tradizionale, quando due piani parziali sono combinati, i predicati di join sono espliciti nella query, e l'algoritmo di enumerazione deve solo considerare il modo più efficace per applicare questi predicati. Nel nostro caso, non è evidente a priori quale giunzione ci porterà ad avere una riscrittura corretta della query, perché stiamo facendo giunzione tra viste, piuttosto che tra le relazioni della base di dati. Quindi, l'algoritmo di enumerazione ha bisogno di considerare diverse giunzioni alternative a quelle espresse esplicitamente nella query. Nella pratica il numero di giunzioni alternative che devono essere considerate possono essere potate in modo significativo utilizzando i meta-dati relativi allo schema. Per esempio, non ha senso cercare di fare join di un attributo stringa con un attributo numerico. Inoltre, in alcuni casi possiamo usare la conoscenza dei vincoli di integrità e la struttura della query per ridurre il numero di predicati di join presi in considerazione. Infine, dopo aver esaminato tutti i possibili predicati di join, l'ottimizzatore deve anche verificare che il piano di accesso risultante sia ancora una soluzione parziale alla query.

Molti dei lavori di questo tipo, come [33] e [20], considerano interrogazioni di tipo SPJ con semantica insiemistica. In questo caso si può verificare se una soluzione è completa adoperando procedure che fanno uso della nozione di inclusione e di dipendenze funzionali. Altri lavori considerano invece interrogazioni di tipo SPJ con semantica "bag di tuple". Con la semantica "bag di tuple", i modi in cui si possono usare le viste per rispondere a una query sono più limitati: ciò è dovuto al fatto che due interrogazioni sono in questo caso equivalenti se e solo se vi è una corrispondenza bi-direzionale 1-a-1

tra esse, che mappa il predicato di giunzione di una query a quello dell'altra. Quindi una vista è utilizzabile solo se è isomorfa a un sottoinsieme dell'interrogazione.

5.5.2 Approccio trasformazionale

Un ottimizzatore di tipo *trasformazionale*, anche detto ottimizzatore *basato su regole*, utilizza un insieme di regole di riscrittura allo scopo di trasformare l'albero sintattico relativo all'interrogazione in un piano di accesso. Le regole devono obbligatoriamente generare delle riscritture equivalenti, al fine di ottenere un piano di esecuzione corretto. Inoltre questo tipo di ottimizzatore non analizza le informazioni statistiche sui dati, dunque fissata un'interrogazione il piano di accesso generato è indipendente dallo stato delle relazioni.

In questa sezione descriviamo alcuni lavori che permettono l'uso di viste materializzate in ottimizzatori di tipo trasformazionali. Il tema comune a questi lavori è che la sostituzione di una parte di una query con una vista è considerata una trasformazione utilizzabile dall'ottimizzatore. Quindi alle regole di riscrittura esistenti, si aggiunge un'ulteriore regola di *View Matching* che punta a effettuare riscritture utilizzando le viste materializzate. Questo approccio è necessario quando l'ottimizzatore intero è di tipo trasformazionale, e permette di trattare facilmente con interrogazioni di tipo *selezione-proiezione-giunzione-groupby* (SPJG).

Nel primo lavoro considerato (si veda [31]) si descrive come un ottimizzatore del sistema BD2 di IBM utilizza la riscrittura della query con viste. L'algoritmo proposto, operando su una struttura dati a grafo chiamata *Query Graph Model* (QGM) che rappresenta un'interrogazione, riesce a decomporla in più sotto-query ognuna rappresentata da un grafo di tipo QGM (queste strutture sono dette *QGM-boxes*), ognuna corrispondente a un blocco SQL di tipo SPJ.

Prima di effettuare la riscrittura, l'algoritmo cerca di far corrispondere le *QGM-boxes* delle viste con quelle dell'interrogazione da riscrivere. Questo controllo è fatto in modo *bottom-up*, partendo dalle QGM-boxes foglia. Una corrispondenza tra una *QGM-box* dell'interrogazione e una *QGM-box* della vista può essere:

- esatta, nel senso che le due box rappresentano query equivalenti, o
- può richiedere un *compenso*, nel senso che richiede un insieme di operazioni aggiuntive che devono essere eseguite sulla *box* della vista al fine di ottenere un risultato equivalente alla *box* della query.

Una coppia di *QGM-boxes* è confrontata solo dopo aver considerato a ogni possibile coppia delle *QGM-boxes* figlie: in questo modo la corrispondenza (e l'eventuale compenso) può essere determinata senza guardare nei sottografi in esse contenuti. L'algoritmo termina quando trova una corrispondenza tra un *QGM-box* radice della vista e qualche *QGM-box* dell'interrogazione.
Oltre all'algoritmo proposto, si mostra come considerare le riscritture a livello *QGM-boxes*, permette di gestire facilmente interrogazioni contenenti più blocchi di tipo SPJ e interrogazioni di tipo SPJG.

Nel lavoro riportato in [34], si utilizza un approccio trasformazionale per integrare in modo uniforme l'uso di viste materializzate, indici specializzati, e vincoli di integrità semantica definiti sullo *storage schema*. Tutto questo è rappresentato all'interno del sistema da vincoli. L'algoritmo proposto prevede due fasi, ciascuna delle quali riguarda un diverso insieme di trasformazioni.
Nella prima fase, detta *chase*, la query è espansa per includere qualsiasi altra struttura (come le viste materializzate o le strutture di accesso) rilevante per la query, in modo da avere un piano di accesso *universale* per la query.
Nella seconda fase, detta *back-chase*, l'ottimizzatore analizza il piano di accesso *universale* ottenuto nella fase precedente al fine di rimuovere le strutture (e quindi le operazioni di giunzione) giudicate poco convenienti. Questa fase può essere accelerata con l'utilizzo di tecniche euristiche, in modo da ottenere velocemente un economico piano di esecuzione risultante.

La procedura proposta si adatta particolarmente a forme di query congiuntive e object-oriented.

In [35] si descrive come il DBMS Oracle 8i utilizza, in modo limitato, delle regole di trasformazione per includere l'uso delle viste durante l'ottimizzazione delle interrogazioni. Anche in questo caso l'algoritmo funziona in due fasi.
Nella prima fase l'algoritmo applica un insieme di regole di riscrittura per sostituire parti dell'interrogazione con riferimenti a viste materializzate esistenti. Le regole di riscrittura, oltre a considerare le condizioni descritte nel paragrafo 4.4, considerano anche vincoli di integrità comunemente incontrati nella pratica, come ad esempio i vincoli di chiave esterna e le dipendenze funzionali. Il risultato della fase di riscrittura è una query che fa riferimento alle viste.
Nella seconda fase l'algoritmo confronta il costo stimato di due piani: il costo del piano di accesso relativo alla riscrittura risultato della prima fase, e il costo del piano di accesso relativo all'interrogazione data in input al sistema (che non considera l'uso di viste materializzate). L'ottimizzatore sceglie di eseguire il piano di accesso più economico tra questi due.
Il vantaggio principale di questo approccio è la sua facilità di implementazione, poiché il modulo di riscrittura capace di usare le viste, è stato aggiunto all'ottimizzatore senza modificare il modulo di join enumeration. D'altra parte, l'algoritmo considera il costo di una sola possibile riscrittura dell'interrogazione che utilizza le viste materializzate, e quindi si rischia di non considerare riscritture migliori.

Un approccio molto interessate, descritto in [32], è quello utilizzato dall'ottimizzatore del sistema *Microsoft SQL Server*. Tra le regole dell'ottimizzatore trasformazionale è stata aggiunta una regola per effettuare il *view matching*, capace di effettuare riscritture su espressioni di tipo SPJ ed espressioni di tipo SPJG.
Per accedere velocemente alle definizioni delle viste materializzare, si introduce il *filter-tree*, una struttura indice intelligente che permette di filtrare in modo efficace l'insieme di viste che sono rilevanti per una particolare interrogazione di tipo SPJG. L'indice è

composto da vari sotto-indici, ognuno dei quali è costruito su una particolare proprietà delle viste (per esempio, l'insieme di tabelle utilizzate dalla vista, l'insieme di attributi proiettati dalla vista, l'insieme degli attributi di raggruppamento). I sotto-indici sono combinati in maniera gerarchica nella struttura del *filter-tree*, dove a ogni livello dell'albero si partizionano le viste in base ad un preciso criterio. Grazie a questa struttura si riescono a individuare riscritture in modo relativamente veloce anche in presenza di un grosso numero di viste.

La parte relativa alla riscrittura adotta un sofisticato algoritmo di matching semantico, il quale individua un buon compromesso tra velocità ed efficacia. Per meglio illustrarlo supponiamo di avere la seguente interrogazione:

```
SELECT *
FROM T1,…,Tk
WHERE Wq
```

Supponiamo inoltre di avere la seguente vista materializzata che seleziona nella clausola *FROM* le stesse relazioni selezionate dall'interrogazione:

```
SELECT *
FROM T1,…,Tk
WHERE Wv
```

A questo punto l'algoritmo di matching deve verificare se l'albero le condizioni W_q espresse nell'interrogazione implicano logicamente le condizioni W_v espresse nella definizione della vista, cioè se $W_q \rightarrow W_v$. La prima operazione svolta è portare entrambi i gruppi di predicati in forma normale congiuntiva, ottenendo due espressioni del tipo:

$$W_q = P_{q1} \wedge \ldots \wedge P_{qn}$$

$$W_v = P_{v1} \wedge \ldots \wedge P_{vm}$$

Successivamente i predicati sono riorganizzati in insiemi ottenendo la forma $W = PE \wedge PR \wedge PU$, dove:

- *PE* è l'insieme dei predicati di equivalenza tra colonne (predicati di equi-join);
- *PR* è l'insieme dei predicati di "*range*", di tipo *attributo-operatore-costante*;
- *PU* è l'insieme dei rimanenti predicati.

Dunque per i predicati relativi all'interrogazione si otterrà l'espressione $W_q = PE_q \wedge PR_q \wedge PU_q$ e per i predicati relativi alla vista si otterrà l'espressione $W_v = PE_v \wedge PR_v \wedge PU_v$. Dopo aver riorganizzato i predicati, si effettuano tre test su di essi:

1. Equi-join subsumption test: controlla se $PE_q \longrightarrow PE_v$.

   In questo test si creano delle classi di equivalenza per la vista e per la query in modo da avere nella stessa classe di equivalente gli attributi vincolati a essere uguali. Al momento della riscrittura se più classi di equivalenza della vista sono corrispondono alla stessa classe di equivalenza dell'interrogazione, allora c'è bisogno di introdurre nuovi predicati per *compensare* la riscrittura.

2. Range subsumption test: controlla se $PR_q \longrightarrow PR_v$.

   Per ogni attributo della query e della vista sono calcolati gli intervalli di valori validi per esso. Prima della riscrittura ogni intervallo relativo alla vista, deve essere associato a un intervallo relativo alla query. Se un intervallo relativo all'interrogazione è contenuto strettamente nell'intervallo della vista corrispondente, allora bisogna applicare un *compenso* nella riscrittura. Se la query ha intervalli aggiuntivi, si possono compensare riapplicandoli nella riscrittura.

3. Residual subsumtion test: controlla se $PU_q \longrightarrow PU_v$.

   Questo test controlla che gli ulteriori predicati della vista hanno un corrispettivo predicato nella query. Se la query ha un predicato aggiuntivo allora esso è aggiunto durante il *compenso* della riscrittura. Per questi test è adottato un approccio puramente sintattico, effettuando un confronto tra stringhe.

5.5.3 Altri approcci

Vi sono infine approcci che utilizzano, per l'ottimizzazione delle interrogazioni con viste materializzate, tecniche usate nel contesto dell'integrazione dati. In quest'ambito si considerano query *datalog* ricorsive, dette anche *query congiuntive*. *Datalog*, sottoinsieme del linguaggio *Prolog*, è un linguaggio per database deduttivi di

interrogazione e di scrittura delle regole. Le espressioni *datalog*, sono in grado di esprimere interrogazioni di tipo SPJ. Una query congiuntiva ha la forma:

$$q(X) :- r_1(X_1), \ldots, r_n(X_n)$$

dove q e $r_1, \ldots, r_n$ sono nomi di predicati. I nomi dei predicati $r_1, \ldots, r_n$ fanno riferimento a relazioni del database. L'atomo *q(X)* è chiamato la *testa* della query, e fa riferisce alla relazione della risposta. Gli atomi $r_1(X_1), \ldots, r_n(X_n)$ sono i *sotto-goal* nel corpo della query. Le tuple $X, X_1, \ldots, X_n$ contengono variabili o costanti. I *sotto-goal* possono essere teste di altre regole o anche predicati aritmetici di confronto $<, \leq, =, \neq$.

Un lavoro appartenente a questa categoria è descritto in [36]. In esso si considera l'utilizzo di viste per l'ottimizzazione delle query da una diversa angolazione. L'algoritmo proposto è simile all'algoritmo *Minicon* per l'integrazione dati, e adotta tre specifici modelli di costo che puntano a: ridurre al minimo il numero di viste nella riscrittura (e quindi ridurre il numero di giunzioni), ridurre la dimensione delle relazioni intermedie calcolate durante la riscrittura, e ridurre la dimensione delle relazioni intermedie, eliminando gli attributi irrilevanti che si hanno nei calcoli.

Capitolo 6

Implementazione nel sistema SADAS

Nel presente lavoro è stato progettato e implementato un modulo dell'ottimizzatore di SADAS, per la riscrittura di interrogazioni con viste materializzate. Questo componente è stato integrato nella struttura dell'ottimizzatore, come una nuova fase "di riscrittura", posta tra la fase di *semplificazione* e quella di *trasformazione logica*.

Per la riscrittura si utilizza un innovativo approccio basato su regole. A differenza degli approcci visti nel capitolo 4, non si punta a un algoritmo di riscrittura generico, ma ad avere una "regola" per ogni tipologia di query che si vuole riscrivere.

Oltre ad utilizzare algoritmi di riscrittura specifici per ogni regola definita nel sistema, il componente è affiancato da un modulo software per la selezione automatica delle viste. Quest'ultimo, per generare script SQL concernenti la creazione di viste materializzate, si avvale delle stesse regole utilizzate dal modulo di riscrittura.

Avendo in comune la parte di codice riguardante le regole, s'instaura uno stretto legame tra i due moduli, che permette, come illustrato in questo capitolo, di individuare le interrogazioni costose da eseguire in termini di tempo, e di creare per esse delle viste materializzate utilizzabili dal sistema di riscrittura.

6.1 Modulo per la selezione automatica delle viste da materializzare

Il sistema inizialmente era dotato di un software di tipo batch, scritto in linguaggio C++, per la selezione automatica delle viste. Questo modulo, chiamato "SdsViews", ha lo scopo di creare, a partire dal file di log di SADAS (SQL.log), un insieme di script per la creazione di viste che, se materializzate, possono essere utilizzate per velocizzare l'esecuzione di alcune tipologie di interrogazioni.

Queste tipologie di query sono quelle che, superata una fase di selezione iniziale, soddisfano una o entrambe le due regole definite dal software stesso. La regola 1, chiamata "Group By", e la regola 2, chiamata "Partizioni", sono discusse nei paragrafi 6.1.2 e 6.1.3.

La selezione iniziale delle query è eseguita basandosi sui seguenti parametri, dati in ingresso da linea di comando:

- *"DB-Name"*: nome del database sul quale eseguire l'analisi;
- *"Min-Elapsed"*: tempo di esecuzione minimo che deve avere una query per essere considerata;
- *"Min-Occurrences"*: numero minimo di occorrenze che deve avere una query nel file di log per essere considerata;
- *"Max-Percentage"*: percentuale massima ammessa del rapporto tra il numero di tuple nel risultato della query e il numero delle tuple della tabella più grande coinvolta nel calcolo.

SdsViews riesegue ogni query relativa al database "*DB-Name"* incontrata nel file di log, e per ciascuna di essa raccoglie le seguenti informazioni statistiche: tempo di esecuzione, numero di occorrenze della query nel file di log, rapporto tra il numero di tuple restituite e il numero di tuple della tabella più grande coinvolta nel calcolo. Queste informazioni sono utili a portare a termine una prima selezione superata dalle sole interrogazioni con le seguenti caratteristiche:

1. La query deve essere stata eseguita un numero considerevole di volte: è forse la caratteristica più banale e facilmente comprensibile, in quanto è inutile creare una struttura su memoria di massa per migliorare i tempi di risposta di interrogazioni eseguite occasionalmente o sporadicamente. Questo filtro è effettuato utilizzando il parametro "*Min-Occurrences"*.
2. La query deve avere un tempo di esecuzione lungo: è inutile velocizzare il calcolo di una query già veloce. Questo filtro è effettuato utilizzando il parametro "*Min-Elapsed"*.
3. La cardinalità della risposta alla query deve essere significativamente inferiore alla dimensione della tabella più grande coinvolta nel calcolo della risposta stessa: creare una vista, il cui numero di tuple soddisfacenti la definizione è nell'ordine di quello della tabella di partenza, non permette di migliorare le prestazioni. Questo filtro è effettuato utilizzando il parametro "*Max-Percentage"*.

Superata la selezione iniziale, le interrogazioni scelte per la costruzione delle viste sono quelle che soddisfano almeno una delle regole definite nel sistema.

Nella fase iniziale del presente lavoro, questo sistema per la selezione automatica delle viste è stato sottoposto a una fase di re-engineering. Particolare attenzione si è posta a riprogettare il software in modo modulare, al fine di agevolare l'aggiunta di nuove regole.
È stata inoltre ristrutturata la documentazione interna utilizzando il formato DoxyGen ([37]), che, stile Javadoc, permette la generazione automatica della documentazione a partire dal codice sorgente scritto nel linguaggio C++.

6.1.1 Re-engineering di SdsViews

Inizialmente *SdsViews* è stato ristrutturato e riorganizzato per facilitare la manutenzione e futuri cambiamenti. In questa fase iniziale di re-engineering, il software è stato diviso

in due moduli: il primo contiene la parte di codice riguardante la selezione iniziale delle query, e il secondo contiene la parte di codice concernente il controllo delle regole e alla generazione degli script SQL delle viste.

Il primo modulo mantiene uno stile di programmazione sostanzialmente procedurale, e il suo codice sorgente è interamente contenuto nel file "final.cpp". Gli interventi di manutenzione effettuati riguardano: l'eliminazione dell'uso di variabili globali, la corretta cancellazione dei file temporanei, e il corretto rilascio di memoria allocata per evitare dei *memory leak*. Si è inoltre eliminato l'uso dei parametri da linea di comando a favore di un file di configurazione (*SdsViews.ini*), il quale permette di specificare:

- I valori dei parametri *DB-Name*, *Min-Elapsed*, *Min-Occurrences*, *Max-Percentage*;
- Il nome del database sul quale lavorare;
- I percorsi dei file necessari al funzionamento del software;
- Le regole da attivare durante l'elaborazione.

```
01  DBNAM = TPCH
02  ELAPS = 500
03  OCCUR = 0
04  PERCE = 90
05  LOGFL = C:\SADAS\FILE\SQL.log
06  PAR01 = C:\SADAS\CHKVIEWS\final_tmpfile1.log
07  PAR02 = C:\SADAS\CHKVIEWS\final_tmpfile2.log
08  PAR03 = C:\SADAS\CHKVIEWS\final_tmpfile3.log
09  PAR04 = C:\SADAS\CHKVIEWS\final_tmpfile4.log
10  PAR05 = C:\SADAS\CHKVIEWS\final_tmpfile5.log
11  PAR06 = C:\SADAS\CHKVIEWS\final_tmpfile6.log
12  PAERR = C:\SADAS\CHKVIEWS\final_tmpfile_err.log
13  PALOG = C:\SADAS\CHKVIEWS\SdsViews.log
14  RULE1 = C:\SADAS\CHKVIEWS\Script_Rule1.sql
15  RULE2 = C:\SADAS\CHKVIEWS\Script_Rule2.sql
16  CHKR1 = 1
17  CHKR2 = 1
```

Tabella 1: Esempio di configurazione del file SdsViews.ini

Un esempio di file di configurazione è riportato in Tabella 1.

L'output di questo modulo è un file di *log temporaneo*, utilizzato poi dal secondo modulo, contenente l'elenco delle interrogazioni selezionate con le relative informazioni raccolte.

Il secondo modulo, il cui diagramma delle classi è riportato in Figura 19, è stato riprogettato seguendo un approccio object-oriented. La computazione è fatta partire dal primo modulo, che, dopo aver creato il file di log temporaneo prima menzionato, esegue il codice sorgente mostrato in Tabella 2. Facendo riferimento a questo frammento di codice, la prima operazione eseguita è la creazione dell'oggetto *cv* di tipo *TSDSCheckViews*. Successivamente si istanziano gli oggetti il cui compito è di controllare se le regole definite nel sistema sono soddisfatte. Nel caso specifico sono *TSDSRule1Checker* e *TSDSRule2Checker*: essi sono aggiunti all'oggetto *cv* con il metodo *addRule()*. Infine, alla riga 22, è chiamato il metodo *checkViews()*.

```
01  TSDSCheckViews * cv = new TSDSCheckViews(nomedatabase, filepathInput);
02
03  if (checkRule1 == 1) {
04    TSDSRule1Checker *rule1Checker = new TSDSRule1Checker;
05    rule1Checker->setScriptFilepath(scriptrule1);
06    rule1Checker->setDatabaseName(nomedatabase);
07    rule1Checker->setMinPercentage(Perc);
08    rule1Checker->setMinElapsed(ELMin);
09    cv->addRule((TSDSIRuleChecker *) rule1Checker);
10  }
11
12  if (checkRule2 == 1) {
13    TSDSRule2Checker *rule2Checker = new TSDSRule2Checker;
14    rule2Checker->setScriptFilepath(scriptrule2);
15    rule2Checker->setDatabaseName(nomedatabase);
16    rule2Checker->setMinPercentage(Perc);
17    rule2Checker->setMinElapsed(ELMin);
18    rule2Checker->setMinOccurences(OCMin);
19    cv->addRule((TSDSIRuleChecker *) rule2Checker);
20  }
21
22  cv->checkViews();
23
24  delete cv;
```

Tabella 2: Creazione di un oggetto di tipo TSDSCheckViews

La classe *TSDSCheckViews* ha il compito di leggere il file di log temporaneo in input, ed eseguire il controllo delle regole per le interrogazioni in esso contenute. Per recuperare facilmente informazioni dal file di log temporaneo, utilizza un oggetto di tipo *QueryExtractor*.
Per eseguire il controllo delle regole, si serve di specifiche classi: ogni regola definita nel sistema è implementata da un controllore (chiamato *checker*), il cui codice è incluso in una classe che implementa l'interfaccia *TSDSIRuleChecker*.
Nel sistema sono attualmente definite due regole: la regola "Group By", il cui codice è contenuto nella classe *TSDSRule1Checker*, e la regola "Partizioni", il cui codice è contenuto nella classe *TSDSRule2Checker*.

I metodi dell'interfaccia *TSDSIRuleChecker* utilizzati dall'istanza di *TSDSCheckViews* sono:

- *startReadingLog()*: chiamato quando si inizia a leggere il file di log temporaneo;
- *checkRule()*: utilizzato per ogni query da elaborare;
- *endOfLog()*: chiamato quando sono state elaborate tutte le interrogazioni contenute nel file di log temporaneo.

Questi tre metodi non restituiscono un valore di ritorno, e, la creazione degli script concernenti le viste, è gestita da *TSDSRule1Checker* e *TSDSRule2Checker*. Per questo motivo, ogni *checker* deve essere configurato, al momento dell'istanziazione, con opportuni parametri specifici della regola, come ad esempio il percorso del file contenente gli script creati.

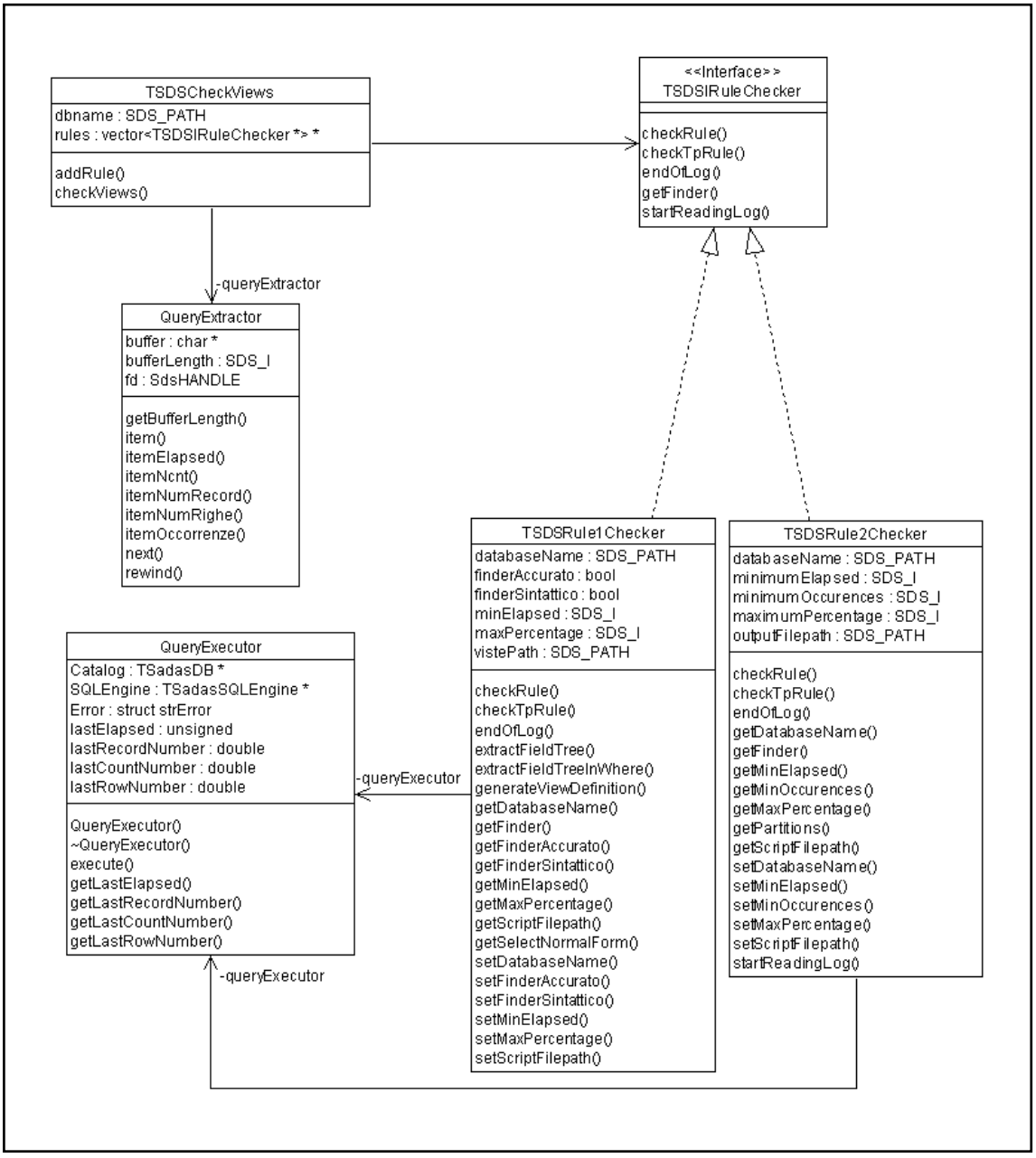

Figura 19: Diagramma delle classi del modulo per la selezione automatica delle viste

Ogni classe riguardante una regola deve, inoltre, implementare i metodi *checkTpRule()* e *getFinder()*, utilizzati nella fase di ottimizzazione delle interrogazioni discussa nel paragrafo 5.2.

Il sistema per la selezione automatica delle viste è stato progettato per essere ospitato sulla macchina server dove SADAS è installato. Idealmente *SdsViews* entra in funzione quando il sistema non sta lavorando e non sono previsti carichi di lavoro per un determinato numero di ore proporzionale alla quantità di tabelle e dati gestiti dal DBMS.
Il punto focale per un sistema di Data Warehousing non è l'immediatezza dei dati, in quanto, in un database di archivi storici, non si bada ad un aggiornamento giornaliero. Proprio per questo, SADAS crea un sistema chiamato *Shadow*, che contiene la copia esatta del database, e su questa copia sono effettuate tutte le modifiche, mentre il sistema originale è sempre in esecuzione e disponibile. Quando la macchina è dichiarata inattiva, allora il sistema *Shadow* aggiorna il sistema principale.
È, inoltre, possibile l'utilizzo di due sistemi separati: uno principale sempre attivo, e uno *Shadow* sul quale sono applicate le modifiche. Utilizzando questa configurazione, quando il server non è utilizzato, i sistemi si possono invertire: il sistema *Shadow* diventa quello principale, mentre l'altro diventa *Shadow*. In un sistema del genere *SdsViews* ha a disposizione anche più di ventiquattro ore per eseguire i calcoli riguardanti la selezione automatica di viste materializzate.

Al termine dell'esecuzione del software si ha in output un file di log contenente statistiche sull'elaborazione effettuata, e, per ogni regola, un file di script SQL contenente le definizioni relative alle viste generate. La materializzazione di tali viste non è, al momento, automatica; questo compito è lasciato all'amministratore di sistema, il quale può scegliere le viste da materializzare, oppure definirne di nuove.

6.1.2 Regola 1 – "Group By"

La regola 1, chiamata "Group By", è soddisfatta da tutte le interrogazioni del tipo:

```
SELECT <attributi di raggruppamento>,
      [<espressioni sui gruppi>]
FROM T1, …, Tn
[WHERE (condizioni sugli attributi di raggruppamento)]
```

```
GROUP BY <attributi di raggruppamento>
[HAVING <condizioni>]
[ORDER BY <attributi di ordinamento>]
```

In altre parole, la regola è soddisfatta da una generica query con clausola *GROUP BY*, dove nella clausola *WHERE* ci sono esclusivamente condizioni solo sugli attributi di raggruppamento. Inoltre, nel caso siano specificate più tabelle nella clausola *FROM*, l'unica forma di giunzione permessa è quella con i predicati espliciti di equi-join nella clausola *WHERE*, in questo caso la regola è soddisfatta anche da interrogazioni che non proiettano gli attributi che si riferiscono ai predicati di giunzione.

La visione multidimensionale dei dati, caratteristica principale dei sistemi di tipo OLAP, ha alla base l'operazione di raggruppamento. Le dimensioni del "cubo" sono definite dagli attributi di raggruppamento specificati nella clausola *GROUP BY*, e le misure sono definite dalle espressioni calcolate sui gruppi formati.
Essendo quest'operazione di utilizzo comune nel Data Warehousing, la presente regola punta a individuare quali sono i cubi utilizzati dall'utente, e creare per essi una vista materializzata.
L'operazione di raggruppamento è, inoltre, onerosa da eseguire, e il numero di tuple risultate da questo tipo di operazione può essere significativamente inferiore al numero delle tuple analizzato per eseguire il raggruppamento stesso.

Ogni qualvolta una query soddisfa questa regola, si genera uno script SQL di questo tipo:

```
CREATE OR REPLACE VIEW <nome della vista> AS
SELECT <attributi di raggruppamento>,
      COUNT(*) AS VCOUNT,
      [<espressioni sui gruppi, ordinate alfabeticamente>]
FROM T1, …, Tn
GROUP BY <attributi di raggruppamento>
```

Materializzando questo tipo di vista, oltre a poterla utilizzare per calcolare la risposta alla query generatrice, si possono ottenere vantaggi nell'esecuzione di interrogazioni con gli stessi attributi di raggruppamento, cioè interrogazioni fatte sullo stesso cubo. Le

misure specificate nell'interrogazione devono essere presenti nel cubo materializzato; al fine di utilizzare la vista in un numero maggiore di riscritture, si inserisce sempre nello script SQL l'espressione "COUNT(*) AS VCOUT".
Si noti come le restanti espressioni calcolate sui gruppi sono specificate in ordine alfabetico. Nel prossimo paragrafo si mostra come questa caratteristica è utilizzata dal modulo di riscrittura.

Per gli attributi di raggruppamento, si mantiene l'ordine specificato nella query generatrice. All'interno di SADAS l'ordine di questi attributi determina l'ordinamento fisico che i gruppi formati hanno sulla memoria permanente; un amministratore di sistema può scegliere di materializzare lo stesso cubo, ma con un differente ordinamento fisico, in modo da poter ottimizzare l'operazione di ordinamento quando possibile.

In teoria, questo tipo di viste, permette di velocizzare interrogazioni il cui insieme di attributi di raggruppamento è strettamente incluso nell'insieme specificato nella vista. Ad esempio una vista con attributi di raggruppamento "A,B,C", potrebbe essere utilizzata nel riscrivere interrogazioni con i raggruppamenti "A,B", "A,C", "B,C", "A", "B", e "C". Questo tipo di ottimizzazione non è, attualmente, previsto dal modulo di riscrittura discusso nel prossimo paragrafo.

6.1.3 Regola 2 – "Partizioni"

La regola 2, chiamata "Partizioni", è soddisfatta da tutte le query del tipo:

```
SELECT …
FROM <tabella dei fatti>, T1, …, Tn
WHERE <predicato di partizione> [AND …]
```

Dove il <*predicato di partizione*> individua una partizione della tabella dei fatti, ed è del tipo "<*tabella dei fatti*>.*attributo* <*operatore*> *costante*" dove l'operatore è del tipo '=' (uguale).

Tale regola vale anche nel caso in cui nella clausola *FROM* vi siano specificate più tabelle, a patto che il predicato di partizione riguardi un attributo della tabella dei fatti.

Lo scopo di questa regola è di individuare le situazioni in cui si hanno, in un'unica tabella, dei dati partizionati: ad esempio, in un archivio delle vendite di prodotti relativo a più anni, si vogliono svolgere operazioni solo sulle vendite di un determinato anno. Materializzando la partizione in una tabella separata (mediante una vista), la si può utilizzare per rieseguire la stessa interrogazione su una tabella dei fatti di cardinalità inferiore.

Nel caso l'interrogazione ha più predicati di partizione, la regola è comunque soddisfatta. Il metodo *checkRule()* della classe *TSDSRule2Checker*, ha il compito di esaminare l'interrogazione data in input, e, nel caso essa soddisfi la regola, creare per ogni predicato di partizione individuato una query del tipo:

```
SELECT *
FROM <tabella dei fatti>
WHERE <predicato di partizione>
```

Tutte le query create sono scritte in un file temporaneo, esaminato in seguito, dal metodo *TSDSRule2Checker::endOfLog()*. Quest'ultimo esegue ogni query contenuta nel file allo scopo di raccogliere le informazioni:

- tempo di esecuzione della query;
- numero di occorrenze della query nel file;
- rapporto tra il numero di tuple nel risultato della query e il numero delle tuple della tabella dei fatti.

Queste informazioni sono utilizzate per filtrare le query, in modo da selezionare solo quelle che hanno una risposta conveniente da materializzare. Per prendere questa decisione, la regola utilizza i valori degli attributi privati della classe TSDSRule2Checker:

- *minimumElapsed*: minimo tempo di esecuzione della query;
- *minimumOccurences*: minimo numero di occorrenze della query nel file;
- *maximumPercentage*: rapporto massimo tra il numero di tuple nel risultato della query e il numero delle tuple della tabella dei fatti.

Il valore degli attributi *minimumElapsed*, *minimumOccurences*, e *maximumPercentage*, è configurabile attraverso appositi metodi pubblici.

Per le interrogazioni che superano la selezione, si genera uno script SQL di questo tipo:

```
CREATE OR REPLACE VIEW <nome della vista> AS
SELECT *
FROM <tabella dei fatti>
WHERE <predicato di partizione>
```

Come per la prima regola, tutti questi script SQL sono scritti in un file di output, e l'effettiva materializzazione di tali viste è lasciata all'amministratore di sistema.

6.2 Modulo per l'utilizzo delle viste in fase di ottimizzazione

Durante il lavoro di tesi è stato progettato e implementato, in linguaggio C++, il modulo di riscrittura, chiamato *ChkViews*, per la riscrittura di interrogazioni con viste materializzate. La struttura complessiva del modulo, mostrata nel diagramma in Figura 20, segue un approccio object-oriented.

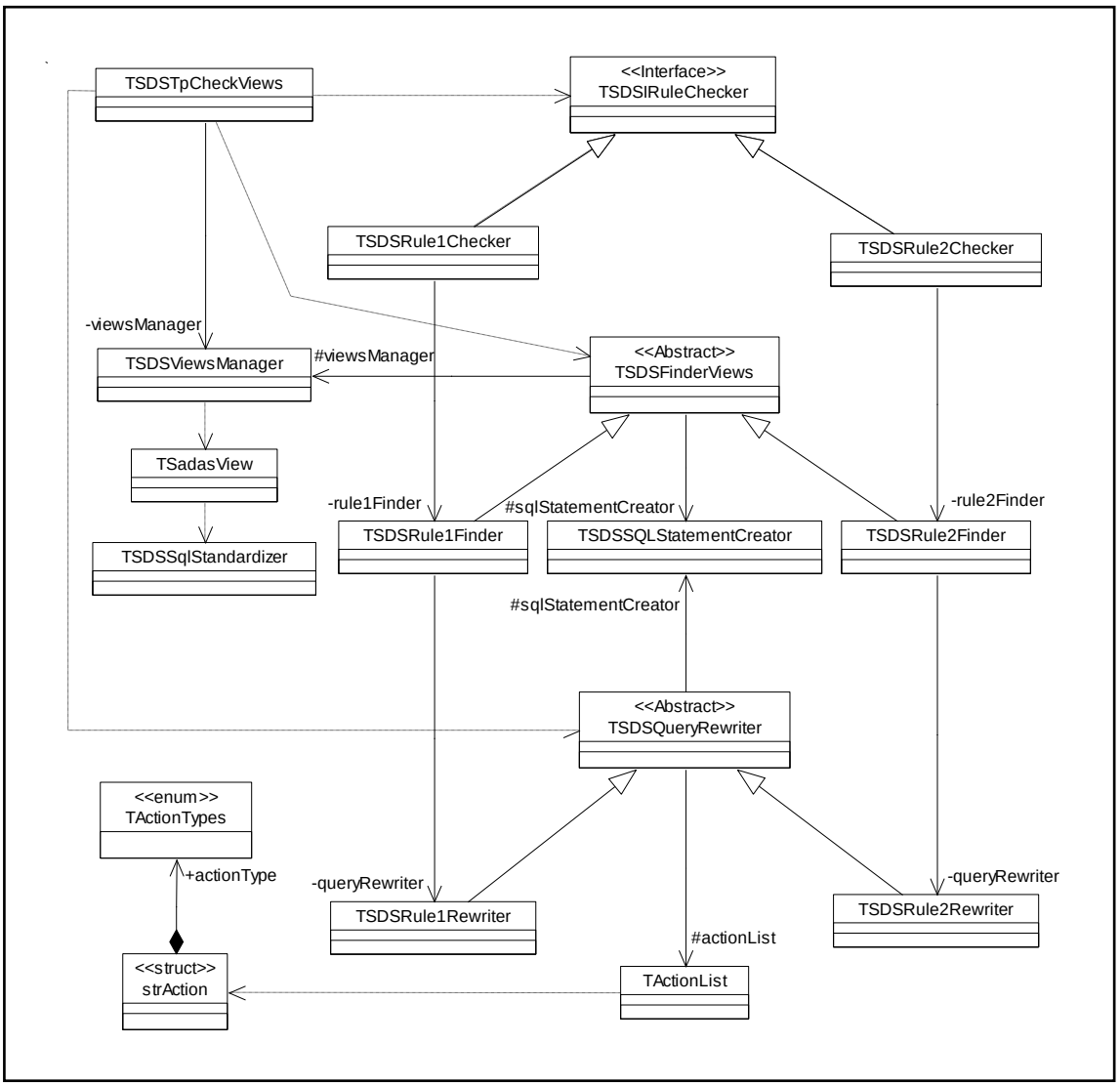

Figura 20: Diagramma delle classi del modulo *ChkViews*

Il modulo in questione è stato integrato nell'ottimizzatore di SADAS come una nuova fase "di riscrittura", posta tra la fase di *semplificazione* e quella di *trasformazione logica*. Per attivare questa nuova fase, l'ottimizzatore crea un oggetto *chkView* di tipo *TSDSTpCheckViews*, e, come mostrato in Tabella 3, di questo invoca il metodo *checkQryViews()*.

```
01  TSDSTpCheckViews *chkView = NULL;
02  chkView = new TSDSTpCheckViews(c, irisp);
03
04  char * newSqlt = chkView->checkQryViews();
05
06  delete chkView;
```

Tabella 3: Creazione di un oggetto di tipo TSDSTpCheckViews

Al costruttore della classe *TSDSTpCheckView* sono passati due parametri: *c* è un vettore di oggetti di tipo *TConnessione* contenente la lista delle connessioni al DBMS, e *irisp* è l'indice della connessione riguardante l'interrogazione che si sta eseguendo. Con questi dati l'oggetto *chkView* può risalire all'oggetto di tipo TSDSSQLStatement rappresentante la query in input, e al catalogo del database sul quale si sta operando.

Al termine dell'esecuzione, il metodo *TSDSTpCheckViews::checkQryViews()*, restituisce una stringa; il suo valore sarà nullo se non vi è stata la riscrittura, altrimenti conterrà il testo SQL dell'interrogazione riscritta.
In quest'ultimo caso, l'oggetto concernente l'ottimizzatore sarà deallocato, e ne sarà creato uno nuovo per l'interrogazione riscritta. Dopo aver rieseguito la fase di semplificazione per il nuovo script SQL, si evita la fase di riscrittura, e si procede con le fasi di trasformazione logica e ottimizzazione fisica.

6.2.1 Catalogo delle viste

Il modulo di riscrittura fa uso di un catalogo delle viste materializzate implementato dalla classe *TSDSViewsManager*. La struttura di questa classe rispecchia in parte quella del design pattern Singleton; per ogni database gestito dal DBMS, la classe gestisce un'unica istanza contenete il catalogo delle viste materializzate relative al particolare database. Per ottenerla, si utilizza il metodo statico:

```
static TSDSViewsManager * getInstance(TSadasDB *);
```

Ogni catalogo delle viste è implementato da un vettore di oggetti di tipo *TSadasView*, ognuno dei quali contiene il riferimento alla tabella nella quale la vista è materializzata, e una stringa contenente la definizione SQL della vista.

6.2.2 Uso delle regole nella riscrittura

L'oggetto *chkView* utilizza le regole "Group By" e "Partizioni", prima definite per il modulo di selezione automatica delle viste, discusso nel paragrafo 5.1. L'idea alla base è di eseguire la riscrittura di un'interrogazione solo se essa soddisfa almeno una delle regole definite nel sistema; in questo modo si possono utilizzare algoritmi di riscrittura specifici della regola, che, conoscendo il modo in cui SdsViews crea le definizioni delle viste, riescono ad avere ottime prestazioni.

Come per *SdsViews*, il codice di ogni regola definita nel sistema è racchiuso in una classe, detta *checker*, che implementa l'interfaccia *TSDSIRuleChecker*. Stavolta ogni regola è accompagnata da altre due classi, dette *finder* e *rewriter*, le quali estendono rispettivamente le classi *TSDSFinderViews* e *TSDSQueryRewriter*. L'utilizzo di *checker*, *finder*, e *rewriter*, nel caso di avvenuta riscrittura dell'interrogazione, è mostrato nel diagramma in Figura 21. Esaminando in dettaglio l'interazione mostrata, data in ingresso un'interrogazione, *chkView* consulta il *checker* della regola invocando il metodo *checkTpRule()*. Se l'interrogazione soddisfa la regola, questo metodo restituisce il valore booleano "true"; nel caso contrario "false", e la parte dell'elaborazione concernente la regola è interrotta.
Se, invece, l'interrogazione soddisfa la regola, si chiede al *checker* un oggetto di tipo *TSDSFinderViews*. Il *finder* ha il compito di trovare, per l'interrogazione soddisfacente la regola, una vista materializzata utilizzabile per la riscrittura. In particolare, quest'operazione è fatta dal metodo *getView()*, il quale, per perseguire il suo scopo, consulta il catalogo delle viste materializzate gestito da un oggetto istanza della classe TSDSViewsManager. Il metodo in questione restituisce un riferimento alla tabella

connessa alla vista materializzata (di tipo *TSadasTabella* *), nel caso si riesce a individuare una vista utile alla riscrittura.

Una volta avuto il riferimento alla tabella in cui la vista è materializzata, *chkView* chiede al *finder* un oggetto di tipo *rewriter*; quest'ultimo ha il compito di riscrivere la query utilizzando la vista individuata dal *finder*. Il metodo utilizzato è *rewrite()*, il quale restituisce all'oggetto chiamante una stringa contenente lo script SQL della query riscritta.

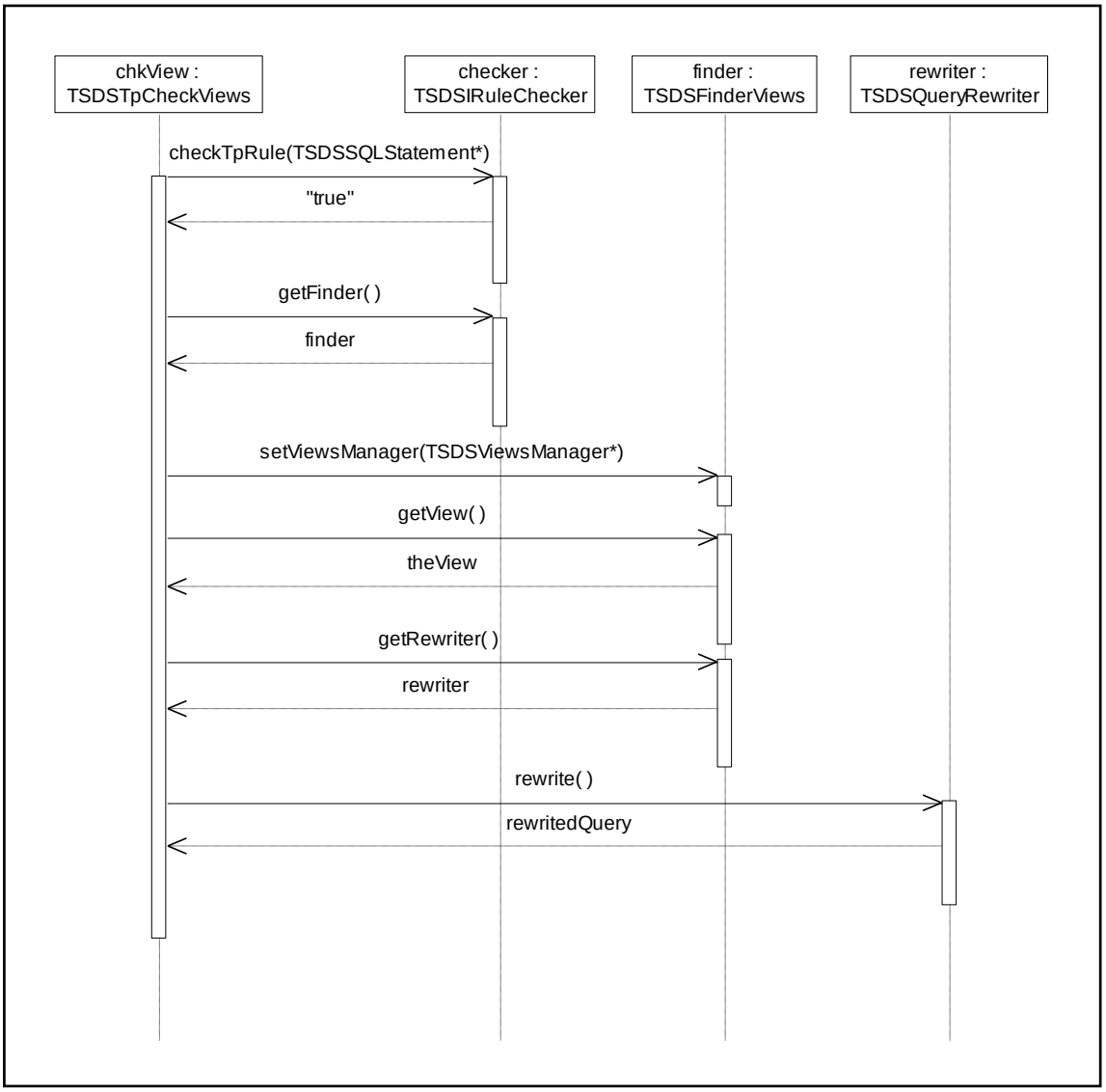

Figura 21: Messaggi scambiati in caso di riscrittura riuscita

Quanto appena descritto riguarda l'interazione degli oggetti che si riferiscono a una singola regola: l'oggetto *chkView* contiene una lista di regole, e, data una query da riscrivere, controlla la possibilità di riscrittura per ogni regola della lista. Questa è la parte centrale del modulo di riscrittura: non c'è un limite al numero di regole, e ognuna di essa è utilizzata mediante i metodi delle superclassi, in modo da disaccoppiare il codice dalle specifiche implementazioni delle regole.
L'ordine d'inserimento delle regole nella lista, ne determina la priorità, poiché l'elaborazione del modulo di riscrittura termina quando non vi sono più regole da controllare, oppure quando è stata eseguita una riscrittura.

6.2.3 La classe TActionList

Gli algoritmi utilizzati per implementare le classi *rewriter*, eseguono la riscrittura dell'interrogazione effettuando opportune modifiche all'oggetto di tipo *TSDSSQLStatement* relativo alla query. Questa scelta implementativa, oltre a riutilizzare le strutture dati esistenti, consente di avere algoritmi semplici, poiché la riscrittura sarà il risultato di tante piccole modifiche a queste strutture.

Dopo aver applicato tutti i cambiamenti all'oggetto *TSDSSQLStatement*, si utilizza il suo metodo *PrintSQL()* per generare il codice SQL dell'interrogazione riscritta: il puntatore a questa stringa di testo rappresenta il valore di ritorno del metodo *rewrite()*.
Per lasciare inalterato, all'uscita del metodo, lo stato delle strutture dati di *TSDSSQLStatement*, tutte le modifiche eseguite si annullano: per perseguire tale scopo si utilizza la classe *TActionList* (Figura 22).

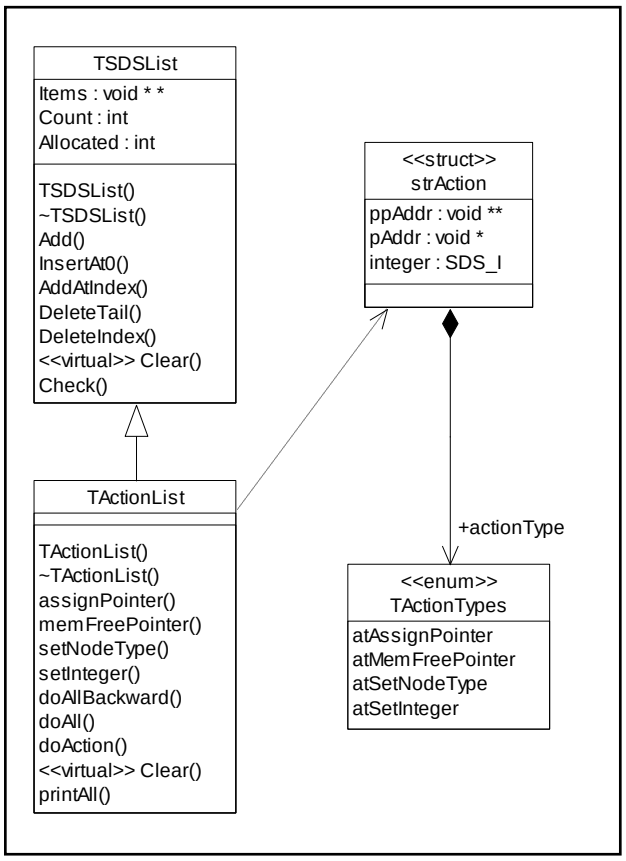

Figura 22: La classe TActionList

La classe *TActionList* implementa una lista, in cui ogni elemento è di tipo *struct strAction*, e rappresenta un'azione. Si possono avere quattro tipi di azione:

- "atAssignPointer": assegnare un puntatore;
- "atMemFreePointer": liberare la memoria referenziata da un puntatore;
- "atSetNodeType": impostare la proprietà *nodeType* contenuta nella classe TSDSNNode;
- "atSetInteger": cambia il valore di una variabile di tipo *int*.

Per ogni tipo di azione, si può inserire in coda alla lista un nuovo elemento, utilizzando i metodi: *assignPointer()*, *memFreePointer()*, *setNodeType()*, e *setInteger()*.

Le azioni inserite nella lista possono essere eseguite, dalla prima all'ultima, chiamando il metodo *doAll()*. Se invece si vogliono eseguire nell'ordine inverso, si utilizza il metodo *doAllBackward()*: è proprio quest'ultimo a essere utilizzato dal *rewriter*, che, per ogni modifica fatta alla struttura *TSDSSQLStatement*, inserisce nella lista un'azione per ripristinare la modifica applicata.

6.2.4 Riscrittura con regola "Group By"

Le classi *checker*, *finder*, e *rewriter*, per la prima regola "Group By", sono rispettivamente *TSDSRule1Checker, TSDSRule1Finder,* e *TSDSRule1Rewriter* (Figura 19).

Le interrogazioni che possono essere riscritte utilizzando questa regola sono, dunque, tutte quelle della forma discussa nella sezione 6.1.2:

```
SELECT <attributi di raggruppamento>,
      [<espressioni sui gruppi>]
FROM T1, …, Tn
[WHERE (condizioni sugli attributi di raggruppamento)]
GROUP BY <attributi di raggruppamento>
[HAVING <condizioni>]
[ORDER BY <attributi di ordinamento>]
```

Nel caso una query soddisfi la regola in questione, l'oggetto *checker* crea uno script SQL contenente la definizione di una possibile vista utilizzabile per la riscrittura. Quest'ultima è creata dal metodo *TSDSRule1Checker::generateViewDefinition()*; lo stesso utilizzato da SdsViews, per creare le definizioni delle viste.

Se la query da riscrivere è una di quelle "generatrici" per *SdsViews*, allora si può ottenere la definizione della possibile vista precedentemente materializzata, senza accedere al catalogo delle viste. Per questo motivo, il primo controllo eseguito dal *finder* è esclusivamente sintattico: si interroga il *view manager*, utilizzando il metodo *TSDSViewsManager::getViewByStdSql()*, per sapere se esiste nel database una vista materializzata avente la particolare definizione.

Nel capitolo 6 dedicato ai test, si mostrano i buoni risultati ottenuti dal controllo sintattico. Ci sono tuttavia dei casi in cui, pur essendo presente la vista materializzata, questo tipo di controllo non è efficace. Ad esempio consideriamo la vista:

```
CREATE OR REPLACE VIEW RONE_000 AS
SELECT CUSTOMER."C_NATIONKEY",
      COUNT(*) AS VCOUNT,
      SUM(LINEITEM."L_EXTENDEDPRICE")
FROM LINEITEM, ORDERS, CUSTOMER
WHERE ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY"
      AND LINEITEM."L_ORDERKEY" = ORDERS."O_ORDERKEY"
GROUP BY CUSTOMER."C_NATIONKEY";
```

Il controllo sintattico fallisce nel caso si ponga al sistema la seguente query, in cui compaiono, in ordine diverso, i predicati di equi-join o le tabelle specificate dalla clausola *FROM*:

```
SELECT CUSTOMER."C_NATIONKEY",
      COUNT(*) AS VCOUNT,
      SUM(LINEITEM."L_EXTENDEDPRICE")
FROM LINEITEM, CUSTOMER, ORDERS
WHERE LINEITEM."L_ORDERKEY" = ORDERS."O_ORDERKEY"
      AND ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY"
GROUP BY CUSTOMER."C_NATIONKEY";
```

Inoltre, il controllo sintattico fallisce quando la clausola SELECT dell'interrogazione non contiene tutti gli attributi specificati nella query generatrice della vista. Nell'esempio proposto, la seguente query non supera il controllo sintattico perché non proietta l'attributo *SUM(LINEITEM."L_EXTENDEDPRICE")*:

```
SELECT CUSTOMER."C_NATIONKEY",
      COUNT(*) AS VCOUNT
FROM LINEITEM, ORDERS, CUSTOMER
WHERE ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY"
      AND LINEITEM."L_ORDERKEY" = ORDERS."O_ORDERKEY"
GROUP BY CUSTOMER."C_NATIONKEY";
```

Al fine di aumentare i casi di successo del *finder*, se il controllo sintattico non individua una vista, si compie un ulteriore controllo approfondito. Quest'ultimo controllo, il cui codice è mostrato in Tabella 4, prevede di ispezionare la definizione delle viste materializzate contenute nel catalogo delle viste.

```
01  vector<TSadasView *> *views = viewsManager->getAllViews();
02  vector<TSadasView *>::iterator iter;
03  bool viewFound = false;
04
05  for (iter=views->begin() ; iter<views->end() ; iter++) {
06
07    TSadasView *aView = *iter;
08    viewFound = false;
09
10    TSDSSQLStatement *aViewStm =
11      sqlStatementCreator->getStatementFromText(aView);
12    bool viewFound = canRewriteRule1(theStatement, aViewStm);
13    delete aViewStm;
14
15    if (viewFound) {
16      theView = aView->getTable(); // Ho individuato la vista
17      break; // Non cerco altre viste
18    }
19  }
```

Tabella 4: Controllo approfondito eseguito da TSDSRule1Finder

Nella riga 1 di Tabella 4, si chiede al *view manager* l'elenco di tutte le viste materializzate contenute nel database. In seguito questo elenco è ispezionato nel corpo del ciclo *for*: per ogni vista si crea un oggetto di tipo *TSDSSQLStatement* (righe 10-11), e lo si passa in input al metodo *TSDSRule1Finder::canRewriteRule1()*, il quale ha il compito di decidere se la vista in questione è utilizzabile per la riscrittura.
Quest'ultimo metodo esegue, in ordine, i seguenti controlli:

1. La definizione della vista deve specificare la clausola *GROUP BY*, e non deve:
    a. Avere le clausole *HAVING* e *ORDER BY*;
    b. Specificare giunzioni esplicite nella clausola *FROM*.
2. La query e la vista, relativamente alla clausola *FROM*, devono far riferimento alle stesse tabelle (lo stesso numero di volte);
3. La clausola *WHERE* della definizione della vista, se esiste, può contenere esclusivamente predicati di equi-join. Questi predicati, inoltre, devono essere presenti nella query;
4. Gli attributi di raggruppamento per la query e per la vista devono essere gli stessi, e specificati nello stesso ordine;
5. Gli attributi di proiezione della query devono essere inclusi negli attributi proiettati dalla vista.

Seguendo la teoria illustrata nel capitolo 4, un'operazione del genere dovrebbe anche verificare che tutti gli attributi specificati nelle clausole *WHERE, HAVING*, e *ORDER BY* della query siano proiettati dalla vista oppure accessibili in altro modo.
Nel caso specifico sappiamo che l'interrogazione soddisfa la prima regola, la quale non permette la specifica di condizioni su attributi non proiettati. Inoltre, il dialetto SQL di SADAS ci assicura che gli attributi specificati dalle clausole *HAVING* e *ORDER BY* siano proiettati.
Queste considerazioni, e i controlli sopra elencati, ci assicurano l'utilizzabilità della vista nel riscrivere la query.

Si noti come il *finder* di questa regola non considera il costo stimato d'esecuzione che avrà l'interrogazione riscritta. Il motivo è che la prima regola punta ad evitare, quando possibile, l'operazione di raggruppamento, in genere molto costosa nei Data Warehouse. Inoltre, per ridurre al minimo i tempi di riscrittura, ci si ferma alla prima vista individuata, e si esclude la possibilità di trovare altre viste utili alla riscrittura.

Nel caso il *finder* individui una vista materializzata utile alla riscrittura, il compito di riscrivere l'interrogazione è lasciato al *rewriter*. Quest'ultimo, dopo aver raccolto informazioni sulla vista da utilizzare, riscrive la query compiendo, in ordine, le seguenti modifiche sul relativo oggetto di tipo *TSDSSQLStatement*:

1. Sposta i predicati espressi dalla proprietà *Having*, nell'albero indicato dalla proprietà *Conditions*. Quest'operazione è fatta creando un nuovo albero per la proprietà *Conditions,* il cui nodo radice è di tipo AND (classe *TSDSANDORNode*); i due figli di questo nodo saranno, a sinistra, l'albero espresso nella clausola *WHERE* della query, e, a destra, l'albero espresso nella *HAVING* della query;
2. Elimina, dall'albero dei predicati della query (proprietà *Conditions*), tutti i predicati già applicati nella definizione della vista;
3. Imposta il valore *NULL* per la proprietà *GropuBy*;

4. Sostituisce, dove opportuno, gli attributi specificati nelle liste *Columns*, *OrderBy*, Conditions, e *Having*, con i corrispondenti attributi proiettati dalla vista;
5. Cancella, dalla lista *Tables*, tutti i riferimenti alle tabelle specificate nella definizione della vista. Dopo inserisce in questa lista il nome della vista, e aggiorna nell'intera query tutti i riferimenti alle tabelle cancellate con il nome della vista.

Dopo aver applicato tutti i cambiamenti all'oggetto *TSDSSQLStatement*, si utilizza il suo metodo *PrintSQL()* per generare il codice SQL dell'interrogazione riscritta: il puntatore a questa stringa di testo rappresenta il valore di ritorno del metodo *rewrite()*.
L'algoritmo qui discusso, conserva traccia delle modifiche fatte alle strutture dati utilizzando una lista di tipo *TActionList*; al termine dell'elaborazione si annullano tali modifiche utilizzando il metodo *TActionList::doAllBackward()*.

6.2.5 Riscrittura con regola "Partizioni"

Le classi *checker*, *finder*, e *rewriter*, per la seconda regola "Partizioni", sono rispettivamente *TSDSRule2Checker*, *TSDSRule2Finder*, e *TSDSRule2Rewriter* (Figura 19).
Le interrogazioni che possono essere riscritte utilizzando questa regola sono, dunque, tutte quelle della forma discussa nella sezione 6.1.3:

```
SELECT …
FROM <tabella dei fatti>, T1, …, Tn
WHERE <predicato di partizione> [AND …]
```

Come già visto per il software di selezione automatica delle viste, questo tipo di interrogazioni può avere più di un predicato di partizione. Il *checker*, nel caso una query soddisfi la regola, conserva ognuno di questi predicati in un vettore di puntatori al tipo *TSDSExprNode*.

In seguito il *finder* genera, per ogni predicato di partizione, uno script SQL della forma:

```
SELECT *
FROM <tabella dei fatti>
WHERE <predicato di partizione>
```

Questo script ha la stessa forma generata da *SdsViews* nel caso il predicato di partizione sia selezionato per la creazione di una vista. Anche stavolta si è in grado di ricostruire la possibile definizione della vista in precedenza materializzata, senza accedere al catalogo delle viste.
Per ogni script SQL generato, si controlla l'effettiva esistenza della vista materializzata, utilizzando il metodo TSDSViewsManager::getViewByStdSql(). Nel caso esista, si conservano le informazioni concernenti la vista e al numero di tuple che soddisfano la sua definizione. La vista restituita dal *finder* sarà quella con il minor numero di tuple, in modo da minimizzare il numero di dati da elaborare per calcolare la risposta alla query riscritta.

Nel caso il *finder* individui una vista materializzata utile alla riscrittura, il compito di riscrivere l'interrogazione è lasciato al *rewriter*. Quest'ultimo, dopo aver raccolto informazioni sulla vista da utilizzare, riscrive la query compiendo, in ordine, le seguenti modifiche sul relativo oggetto di tipo TSDSSQLStatement:

1. Cancella dalla lista *Tables* il nome della tabella dei fatti, e al suo posto inserisce il nome della vista;
2. Esamina le strutture *Columns*, *Conditions*, *GroupBy*, *Having*, e *OrderBy*, per sostituire ogni riferimento alla tabella dei fatti con il nome della vista.

Dopo aver applicato i cambiamenti all'oggetto *TSDSSQLStatement*, si utilizza il suo metodo *PrintSQL()* per generare il codice SQL dell'interrogazione riscritta: il puntatore a questa stringa di testo rappresenta il valore di ritorno del metodo *rewrite()*.
L'algoritmo qui discusso, conserva traccia delle modifiche fatte alle strutture dati utilizzando una lista di tipo *TActionList*; al termine dell'elaborazione si annullano tali modifiche utilizzando il metodo *TActionList::doAllBackward()*.

Capitolo 7

Test

In questo capitolo sono riportati i risultati dei test eseguiti sul sistema, riguardanti sia il modulo di selezione automatica delle viste *SdsViews*, sia il modulo di riscrittura delle interrogazioni *ChkViews*.
In particolare si paragonano le prestazioni del server SADAS avute con e senza il modulo *ChkViews*.

7.1 Ambiente di test

I test sono stati eseguiti su una macchina con processore AMD Athlon™ 3000+ a 1,80GHz, e 1,18 GB di memoria RAM. Il sistema operativo è Microsoft Windows XP Professional, aggiornato alla versione 5.1.2600 con Service Pack 2.

Sulla macchina in questione si è installato il server SADAS, e si è creato un database di test utilizzando lo schema TPC-H (Figura 23), introdotto nel benchmark H definito dall'ente TCP[8].

[8] Il TPC (Transaction Processing Performance Council) è un'organizzazione, non a scopo di lucro, fondata con l'intenzione di fornire degli strumenti validi per misurare le performance dei DBMS. Composta da rappresentanti provenienti dalle maggiori società di software ed hardware al mondo, il TPC mette a disposizione diverse tipologie di benchmark. Il benchmark™ H proposto dal TPC (TPC- H) è definito come un benchmark per sistemi di supporto alle decisioni. I dati che popolano lo schema sono stati scelti in modo tale da assicurare un'ampia pertinenza in vari contesti "business- oriented".

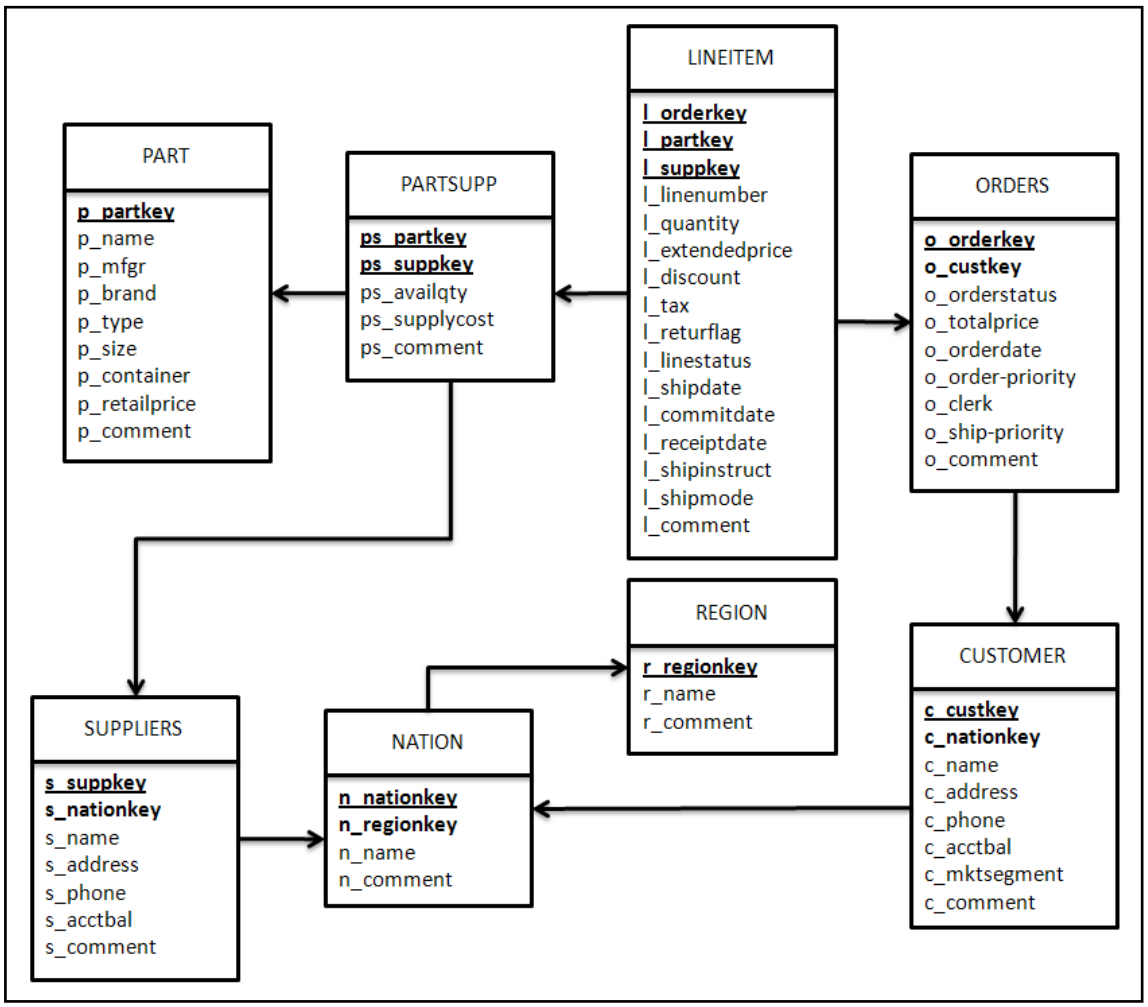

Figura 23: Schema TPC-H

Il database creato è stato alimentato con 5GB di dati, generati seguendo le indicazioni fornite dal benchmark stesso.

7.2 Esecuzione del software di selezione automatica delle viste

È stato eseguito il software di selezione automatica delle viste, dando in input un file di log di SADAS contenente 228 differenti interrogazioni poste sul database di test. Il file di configurazione "SdsViews.ini" è stato configurato con i seguenti parametri:

- Tempo di esecuzione minimo della query (*Min-Elapsed*): 500 ms

- Percentuale massima ammessa del rapporto tra il numero di tuple nel risultato della query ed il numero delle tuple della tabella più grande coinvolta nel calcolo (*Max-Percentage*): 90
- Numero di occorrenze minimo della query nel file di log (*Min-Occurrences*): 0

L'output prodotto da *SdsViews* è riportato in Tabella 4.

```
Batch program C:\SDS2006\exe\SdsViews.exe. Start time: 13:28:44.796 05/01/2011
Reading parameter from configuration file SdsViews.ini...
        Database name: TPCH
        Minimum elapsed time: 500
        Minimum occurrences: 0
        Max percentage (#rows/#recs): 90
        OUTPUT rule 1 script: C:\SADAS\CHKVIEWS\SCRIPT_RULE1.SQL
        OUTPUT rule 2 script: C:\SADAS\CHKVIEWS\SCRIPT_RULE2.SQL
        SADAS Log File C:\SADAS\FILE\SQL.LOG
[  0%] filtering TPCH queries...
[  9%] done.
[ 10%] preparing the file for sorting...
[ 19%] done.
[ 20%] sorting the file (13:28:44.875 05/01/2011)...
[ 29%] done (13:28:44.906 05/01/2011).
[ 30%] running and filtering queries (may take a long time)...
[ 49%] done.
[ 50%] sorting the file again (13:45:28.343 05/01/2011)...
[ 59%] done (13:45:28.343 05/01/2011).
[ 60%] creating the log file C:\SADAS\CHKVIEWS\SDSVIEWS.LOG
[ 69%] done.
[ 70%] identifying the possible definitions of materialized view...
[ 95%] done.
[ 98%] deleting temporary files...
[100%] done.
Total batch processing time: 1243s
```

Tabella 5: Output prodotto da SdsViews

Al termine dell'esecuzione, la regola "Group By" ha prodotto le seguenti 15 definizioni di viste:

```
CREATE OR REPLACE VIEW RONE_000 AS
SELECT CUSTOMER."C_NATIONKEY", COUNT(*) AS VCOUNT,
      SUM(LINEITEM."L_EXTENDEDPRICE")
FROM LINEITEM, ORDERS, CUSTOMER
WHERE ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY"
      AND LINEITEM."L_ORDERKEY" = ORDERS."O_ORDERKEY"
GROUP BY CUSTOMER."C_NATIONKEY";

CREATE OR REPLACE VIEW RONE_001 AS
SELECT LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
```

```
      MAX(ORDERS."O_ORDERDATE")
FROM LINEITEM, ORDERS
WHERE LINEITEM."L_ORDERKEY" = ORDERS."O_ORDERKEY"
GROUP BY LINEITEM."L_QUANTITY";

CREATE OR REPLACE VIEW RONE_002 AS
SELECT ABS(LINEITEM."L_QUANTITY"), COUNT(*) AS VCOUNT
FROM LINEITEM
GROUP BY ABS(LINEITEM."L_QUANTITY");

CREATE OR REPLACE VIEW RONE_003 AS
SELECT LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
      ROUND(SUM(LINEITEM."L_EXTENDEDPRICE"), 0),
      SUM(LINEITEM."L_EXTENDEDPRICE"),
      SUM(ROUND(LINEITEM."L_EXTENDEDPRICE", 0))
FROM LINEITEM
GROUP BY LINEITEM."L_QUANTITY";

CREATE OR REPLACE VIEW RONE_004 AS
SELECT ORDERS."O_ORDERDATE", COUNT(*) AS VCOUNT,
      MAX(LINEITEM."L_SHIPDATE")
FROM LINEITEM, ORDERS
WHERE LINEITEM."L_ORDERKEY" = ORDERS."O_ORDERKEY"
GROUP BY ORDERS."O_ORDERDATE";

CREATE OR REPLACE VIEW RONE_005 AS
SELECT LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
      ROUND(SUM(LINEITEM."L_EXTENDEDPRICE"), 0),
      SUM(LINEITEM."L_EXTENDEDPRICE"),
      SUM(ROUND(LINEITEM."L_EXTENDEDPRICE", 0))
FROM LINEITEM
GROUP BY LINEITEM."L_QUANTITY";

CREATE OR REPLACE VIEW RONE_006 AS
SELECT LINEITEM."L_RETURNFLAG", COUNT(*) AS VCOUNT,
      (SUM(LINEITEM."L_DISCOUNT")) + (SUM(LINEITEM."L_QUANTITY"))
      - (SUM(LINEITEM."L_TAX")) -
      (SUM(LINEITEM."L_EXTENDEDPRICE")),
      (SUM(LINEITEM."L_DISCOUNT")) - (SUM(LINEITEM."L_TAX")),
      (SUM(LINEITEM."L_EXTENDEDPRICE")) -
      (SUM(LINEITEM."L_QUANTITY"))
FROM LINEITEM
GROUP BY LINEITEM."L_RETURNFLAG";

CREATE OR REPLACE VIEW RONE_007 AS
SELECT DATEPART('YY', LINEITEM."L_SHIPDATE"),
      LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
      SUM(LINEITEM."L_EXTENDEDPRICE")
FROM LINEITEM
GROUP BY DATEPART('YY', LINEITEM."L_SHIPDATE"),
      LINEITEM."L_QUANTITY";

CREATE OR REPLACE VIEW RONE_008 AS
SELECT LINEITEM."L_RETURNFLAG", COUNT(*) AS VCOUNT,
```

```
      SUM((LINEITEM."L_QUANTITY") * (5.5))
FROM LINEITEM
GROUP BY LINEITEM."L_RETURNFLAG";

CREATE OR REPLACE VIEW RONE_009 AS
SELECT LINEITEM."L_RETURNFLAG", COUNT(*) AS VCOUNT,
      (SUM(LINEITEM."L_QUANTITY")) -
(SUM(LINEITEM."L_DISCOUNT")),
      (SUM(LINEITEM."L_QUANTITY")) - (SUM(LINEITEM."L_TAX")),
      (SUM(LINEITEM."L_TAX")) - (SUM(LINEITEM."L_DISCOUNT"))
FROM LINEITEM
GROUP BY LINEITEM."L_RETURNFLAG";

CREATE OR REPLACE VIEW RONE_010 AS
SELECT (LINEITEM."L_QUANTITY") * (2),
      LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
      SUM(LINEITEM."L_EXTENDEDPRICE")
FROM LINEITEM
GROUP BY (LINEITEM."L_QUANTITY") * (2), LINEITEM."L_QUANTITY";

CREATE OR REPLACE VIEW RONE_011 AS
SELECT '01', LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
      SUM(LINEITEM."L_EXTENDEDPRICE")
FROM LINEITEM
GROUP BY '01', LINEITEM."L_QUANTITY";

CREATE OR REPLACE VIEW RONE_012 AS
SELECT 999, LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
      SUM(LINEITEM."L_EXTENDEDPRICE")
FROM LINEITEM
GROUP BY 999, LINEITEM."L_QUANTITY";

CREATE OR REPLACE VIEW RONE_013 AS
SELECT LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
      (ROUND(SUM(LINEITEM."L_EXTENDEDPRICE"), 0)) +
      (ROUND(SUM(LINEITEM."L_EXTENDEDPRICE"), 0))
FROM LINEITEM
GROUP BY LINEITEM."L_QUANTITY";

CREATE OR REPLACE VIEW RONE_014 AS
SELECT LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
      (SUM(LINEITEM."L_EXTENDEDPRICE")) +
      (SUM(LINEITEM."L_EXTENDEDPRICE"))
FROM LINEITEM
GROUP BY LINEITEM."L_QUANTITY";
```

La regola “Partizioni” ha, invece, prodotto le seguenti 4 definizioni di viste:

```
CREATE OR REPLACE VIEW RTWO_000 AS
SELECT *
FROM LINEITEM
WHERE L_RETURNFLAG = 'R';
```

```
CREATE OR REPLACE VIEW RTWO_001 AS
SELECT * FROM LINEITEM
WHERE L_RETURNFLAG = 'A';

CREATE OR REPLACE VIEW RTWO_002 AS
SELECT *
FROM LINEITEM
WHERE L_RETURNFLAG = 'N';

CREATE OR REPLACE VIEW RTWO_003 AS
SELECT *
FROM LINEITEM
WHERE L_SHIPDATE = '10/02/1995';
```

Tutte le viste suggerite da SdsViews sono state materializzate nella base di dati.
Allo scopo di testare il funzionamento del modulo di riscrittura, e le prestazioni ottenute con il suo utilizzo, si sono creati due test-set: il primo, riportato nell'appendice A, contenente 30 interrogazioni relative alla riscrittura con regola "Group By", ed il secondo, riportato nell'appendice B, contenente 29 interrogazioni relative alla riscrittura con regola "Partizioni".

7.3 Test di riscrittura con regola "Group By"

Si sono eseguite le interrogazioni contenute nel test-set per la regola "Group By", su una versione di SADAS in cui era stata disabilitata la *fase di riscrittura*. In seguito si è abilitato l'uso del modulo *ChkViews*, e si sono rieseguite le query.
In Figura 24 sono riportati, in millisecondi, i tempi di esecuzione ottenuti. Tutte le query del test-set sono state riscritte con successo; il corrispondente codice SQL è riportato nell'appendice C.

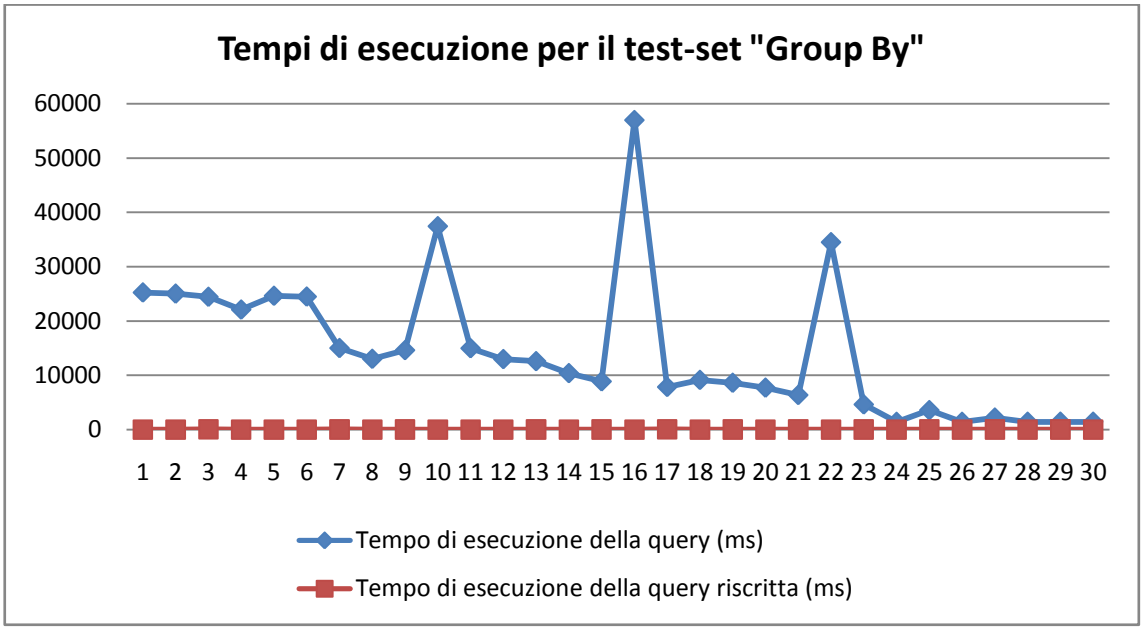

Figura 24: Tempi di esecuzione per il test-set "Group By"

Analizzando il grafico di Figura 24, si nota come il tempo di esecuzione della query riscritta sia notevolmente inferiore a quello della query originale: quest'evidente aumento di prestazioni è dovuto all'eliminazione dell'operazione di raggruppamento.

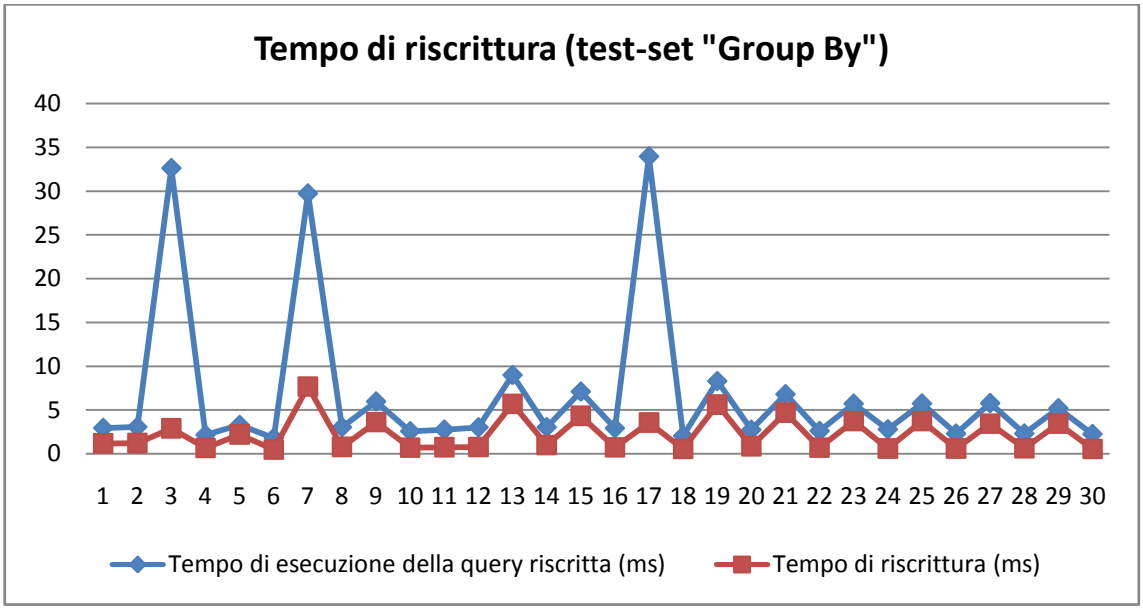

Figura 25: Tempi di riscrittura per il test-set "Group By"

I tempi di esecuzione delle query riscritte sono meglio rappresentati in Figura 25, dove è confrontato con il tempo impiegato dal modulo *ChkViews* per la riscrittura. Si noti come i tempi di riscrittura, mediamente pari a 2,23 ms, sono inferiori al tempi di esecuzione dell'interrogazione riscritta.

Finora sono stati esaminati i test eseguiti con un *finder* che si avvale sia del controllo sintattico, sia del controllo approfondito. Al fine di esaminare più in dettaglio le prestazioni avute con questi due controlli, si è provato a rieseguire il test-set, con un *finder* dotato solo di controllo sintattico, e con un finder dotato solo di controllo approfondito.
I tempi di riscrittura ottenuti sono riportati in Figura 26. Si noti come la riscrittura che utilizza esclusivamente il controllo sintattico sia, in tutti i casi, più veloce di quella con il controllo approfondito.
Si noti, inoltre, come i tempi di riscrittura tendono ad aumentare; questo è dovuto al fatto che le viste sono gestite dal catalogo mediante una lista. Infatti, le viste sono state materializza nell'ordine suggerito da SdsViews, e, sempre nello stesso ordine, sono stati costruiti i casi di test. Di conseguenza i tempi di riscrittura sono pesantemente influenzati dal tempo di accesso lineare al catalogo delle viste.

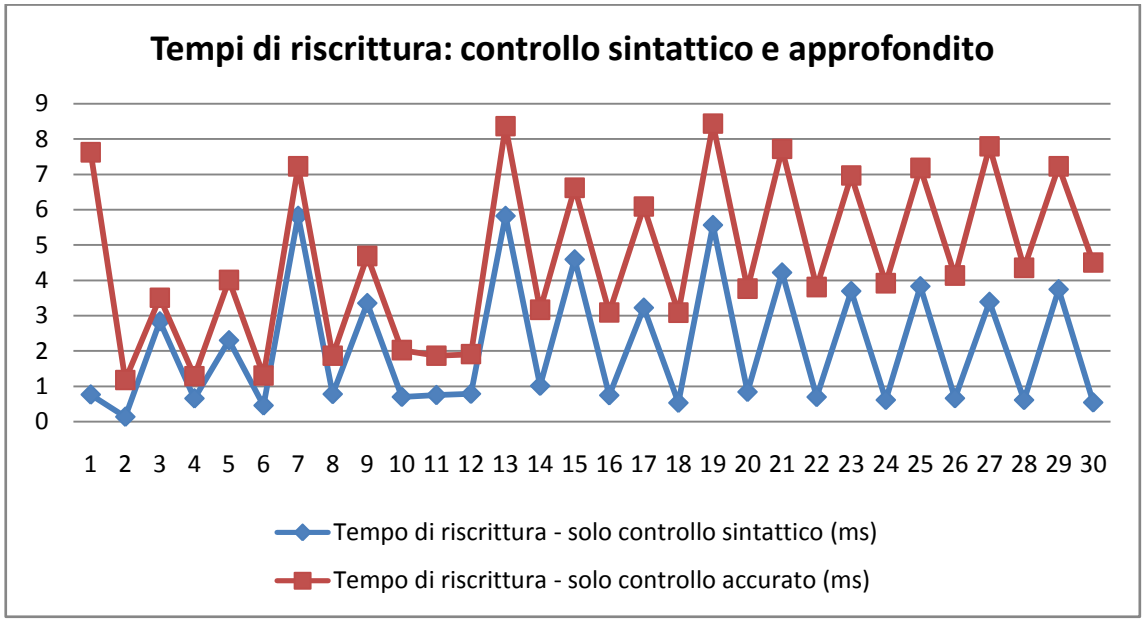

Figura 26: Confronto tra controllo sintattico e controllo approfondito

Con un software di selezione automatica come *SdsViews*, il numero delle viste materializzate può aumentare notevolmente rispetto al numero delle tabelle contenute nel database, e con esso aumenta il tempo medio di riscrittura avuto dal modulo *ChkViews*.

Per evitare questo degrado di prestazioni, la classe *TSDSViewsManager* andrebbe riprogettata, in modo da fornire un metodo di accesso alle viste materializzate più efficiente. Ispirandosi al metodo di accesso descritto in [32], si potrebbe pensare di dividere le viste in base alle proprietà della relativa definizione; un primo criterio potrebbe essere "avere la clausola GROUP BY". In questo modo il finder in questione potrebbe svolgere la sua ricerca prendendo in considerazione esclusivamente le viste contenenti l'operazione di raggruppamento.

Un'altra soluzione a questo problema, potrebbe essere quella di calcolare il valore hash[9] della definizione della vista, e fornire un accesso al catalogo delle viste mediante questo

[9] "Una funzione hash è una funzione non iniettiva che mappa una stringa di lunghezza arbitraria in una stringa di lunghezza predefinita. Esistono numerosi algoritmi che realizzano funzioni hash con particolari proprietà che dipendono dall'applicazione" (si veda [38]).

valore. Quest'ultimo tipo di accesso è però utile solo per il controllo sintattico, e non migliora le prestazioni del controllo approfondito.

7.4 Test di riscrittura con regola "Partizioni"

Analogamente a quanto fatto per la prima regola, si sono eseguite le interrogazioni contenute nel test-set per la regola "Partizioni", con una versione di SADAS in cui era stata disabilitata la *fase di riscrittura*, e con un'altra in cui si è abilitato il modulo *ChkViews*.
Tutte le query del test-set sono state riscritte con successo; il corrispondente codice SQL è riportato nell'appendice D.

In Figura 27 sono riportati, in millisecondi, i tempi di esecuzione ottenuti.
Diversamente a quanto accade per la regola "Group By", questo tipo di riscrittura non permette di evitare operazioni costose, come quella di raggruppamento, ma permette di esaminare un numero minore di dati nel rispondere alla query. I tempi ottenuti sono leggermente migliori dei tempi di esecuzione della corrispondente query originale.

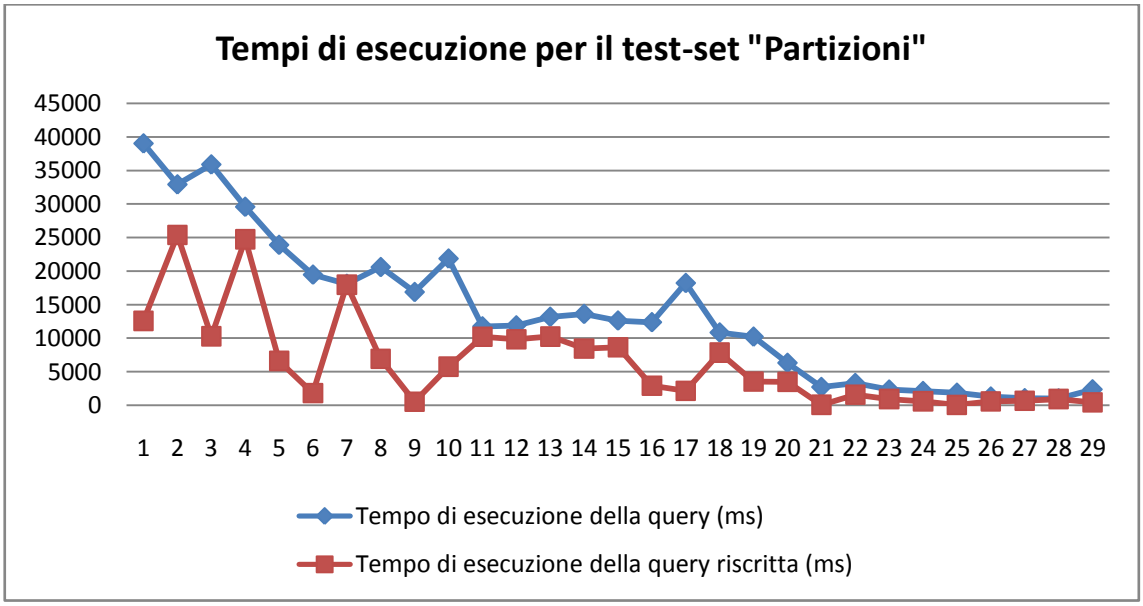

Figura 27: Tempi di esecuzione per il test-set "Partizioni"

In ambienti di produzione reali, si opera con database in cui le tabelle hanno dimensione maggiore di quelle utilizzate in questi test. In questi database si possono individuare viste convenienti da materializzare, eseguendo *SdsViews* con un valore maggiore per il parametro *Min-Elapsed*. In questi casi si può avere un incremento di prestazione migliore di quello avuto in questi test.

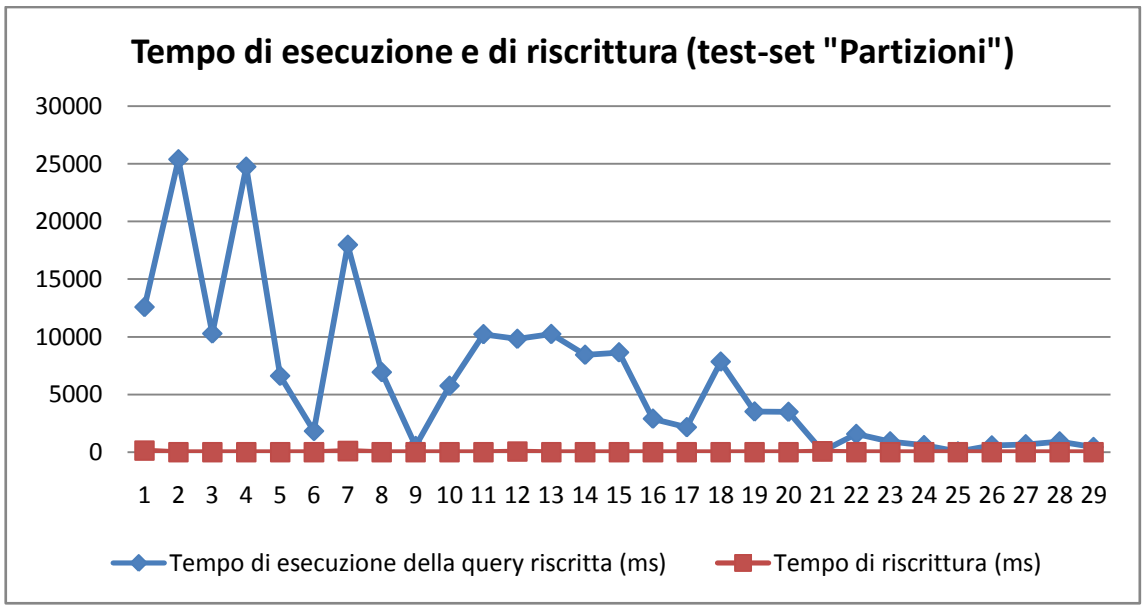

Figura 28: Tempi di riscrittura per il test-set "Partizioni"

Riportato in Figura 28 vi sono è i tempi di esecuzione delle query riscritte, confrontati con i tempi di riscrittura riportati dal modulo *ChkViews*. Anche in questo caso i tempi di riscrittura, mediamente pari a 12,44 ms, sono inferiori al tempi di esecuzione dell'interrogazione riscritta.

Capitolo 8

Conclusioni

SADAS è un DBMS specializzato nella gestione di Data Warehouse, sviluppato dall'azienda *Advanced Systems*. Basato sul modello dei dati relazionale, SADAS utilizza un metodo di archiviazione dei dati colonnare (*column-based*) e altre strutture di indicizzazione specializzate. Queste caratteristiche permettono di avere, in ambienti di Data Warehousing, performance migliori rispetto ai DBMS tradizionali.

Una tecnica comunemente utilizzata dagli analisti dei dati è di creare viste materializzate allo scopo di velocizzare il calcolo della risposta a interrogazioni (*query*) frequentemente poste alla base di dati.
Tale tecnica è senza dubbio più efficace se affiancata da un sistema di ottimizzazione delle interrogazioni in grado di utilizzare automaticamente le viste materializzate. Un approccio utilizzato per risolvere tale problema riguarda la *riscrittura dell'interrogazione con viste materializzate* (*view-based query rewriting*).

Nel lavoro svolto è stato introdotto un modulo dell'ottimizzatore di SADAS, per la riscrittura di interrogazioni con viste materializzate. Questo componente è stato integrato nella struttura dell'ottimizzatore, come una nuova fase "di riscrittura", posta tra la fase di *semplificazione* e quella di *trasformazione logica*.
L'intera parte implementativa è stata fatta utilizzando linguaggio C++, in modo da agevolarne l'integrazione, e sfruttare il codice preesistente.

Il sistema sviluppato affronta il problema di rispondere a query utilizzando le viste, mediante la riscrittura dell'interrogazione. È stato introdotto un innovativo approccio basato su regole, nel quale non si punta a un algoritmo di riscrittura generico, ma ad avere una "regola" per ogni tipologia di query che si vuole riscrivere.

Il componente per la riscrittura è stato affiancato da un modulo software per la selezione automatica delle viste. Quest'ultimo, per generare script SQL concernenti la creazione di viste materializzate, si serve dello stesso insieme di regole definite per il modulo di riscrittura.
Avendo in comune la parte di codice riguardante le regole, si è instaurato uno stretto legame tra i due moduli, che permette di individuare le interrogazioni costose da eseguire in termini di tempo, e di creare per esse delle viste materializzate utilizzabili dal sistema di riscrittura.

Il codice prodotto è stato testato con strumenti di *memory profiler*, nel caso specifico *CodeGuard* integrato nell'ambiente di sviluppo *Borland Developer Studio 2006*, al fine di ottimizzare l'utilizzo della memoria. Inoltre, la documentazione interna è stata redatta utilizzando il formato DoxyGen (si veda[37]), il quale permette la generazione automatica della documentazione, dal codice sorgente scritto nel linguaggio C++.

I risultati ottenuti dai test prestazionali (*performance test*), riportati nel capitolo 7, mostrano un ottimo incremento delle prestazioni avuto dall'utilizzo congiunto del modulo di riscrittura e del modulo di selezione automatica delle viste.

Successivi sviluppi potrebbero riguardare l'introduzione di nuove regole al sistema di riscrittura, in modo da riuscire a riscrivere più tipologie di query.
Altri progressi potrebbero, invece, rivedere l'accesso al catalogo delle viste materializzate. Infatti, con un software di selezione automatica, il numero delle viste può aumentare notevolmente rispetto al numero delle tabelle contenute nel database.

Per evitare un degrado di prestazioni dovuto all'accesso al catalogo implementato dalla classe *TSDSViewsManager*, la stessa andrebbe riprogettata, in modo da fornire un metodo di accesso alle viste materializzate più efficiente. Si potrebbe pensare di dividere le viste in base alle proprietà della relativa definizione; un primo criterio potrebbe essere "avere la clausola GROUP BY". In questo modo il finder in questione potrebbe svolgere la sua ricerca prendendo in considerazione esclusivamente le viste contenenti l'operazione di raggruppamento.

Un'altra soluzione a questo problema, potrebbe essere quella di calcolare il valore hash della definizione della vista, e fornire un accesso al catalogo delle viste mediante questo valore. Quest'ultimo tipo di accesso è però utile solo per il controllo di tipo sintattico, e non migliora le prestazioni del controllo approfondito definito nella regola "Group By".

Appendice A. Test-set per la regola “Group By”

Di seguito sono elencate le interrogazioni utilizzate per il test della regola “Group By”. Ad ognuna di essa è associato un numero progressivo, utilizzato, nel capitolo 6, come identificativo della query.

Query N°1

```
SELECT C_NATIONKEY, SUM(L_EXTENDEDPRICE) FROM LINEITEM, ORDERS,
CUSTOMER WHERE L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY GROUP
BY C_NATIONKEY;
```

Query N°2

```
SELECT CUSTOMER."C_NATIONKEY", COUNT(*) AS VCOUNT,
SUM(LINEITEM."L_EXTENDEDPRICE") FROM LINEITEM, ORDERS, CUSTOMER WHERE
ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" AND LINEITEM."L_ORDERKEY" =
ORDERS."O_ORDERKEY" GROUP BY CUSTOMER."C_NATIONKEY";
```

Query N°3

```
SELECT L_QUANTITY,COUNT(*), MAX ( O_ORDERDATE) FROM LINEITEM, ORDERS
WHERE L_ORDERKEY = O_ORDERKEY AND L_QUANTITY > 1 GROUP BY L_QUANTITY;
```

Query N°4

```
SELECT LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
MAX(ORDERS."O_ORDERDATE") FROM LINEITEM, ORDERS WHERE
LINEITEM."L_ORDERKEY" = ORDERS."O_ORDERKEY" GROUP BY
LINEITEM."L_QUANTITY";
```

Query N°5

```
SELECT ABS(L_QUANTITY) FROM LINEITEM GROUP BY ABS(L_QUANTITY);
```

Query N°6

```
SELECT ABS(LINEITEM."L_QUANTITY"), COUNT(*) AS VCOUNT FROM LINEITEM
GROUP BY ABS(LINEITEM."L_QUANTITY");
```

Query N°7

```
SELECT SUM (L_EXTENDEDPRICE), SUM (ROUND(L_EXTENDEDPRICE,0)) ,
ROUND(SUM(L_EXTENDEDPRICE),0),L_QUANTITY FROM LINEITEM GROUP BY
L_QUANTITY;
```

Query N°8

```
SELECT LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
ROUND(SUM(LINEITEM."L_EXTENDEDPRICE"), 0),
SUM(LINEITEM."L_EXTENDEDPRICE"), SUM(ROUND(LINEITEM."L_EXTENDEDPRICE",
0)) FROM LINEITEM GROUP BY LINEITEM."L_QUANTITY";
```

Query N°9

```
SELECT O_ORDERDATE, MAX (L_SHIPDATE) FROM LINEITEM, ORDERS WHERE
L_ORDERKEY = O_ORDERKEY AND O_ORDERDATE > '31/12/1996' GROUP BY
O_ORDERDATE;
```

Query N°10

```
SELECT ORDERS."O_ORDERDATE", COUNT(*) AS VCOUNT,
MAX(LINEITEM."L_SHIPDATE") FROM LINEITEM, ORDERS WHERE
LINEITEM."L_ORDERKEY" = ORDERS."O_ORDERKEY" GROUP BY
ORDERS."O_ORDERDATE";
```

Query N°11

```
SELECT L_QUANTITY,SUM (L_EXTENDEDPRICE), SUM
(ROUND(L_EXTENDEDPRICE,0)) , ROUND(SUM(L_EXTENDEDPRICE),0) FROM
LINEITEM GROUP BY L_QUANTITY;
```

Query N°12

```
SELECT LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
ROUND(SUM(LINEITEM."L_EXTENDEDPRICE"), 0),
SUM(LINEITEM."L_EXTENDEDPRICE"), SUM(ROUND(LINEITEM."L_EXTENDEDPRICE",
0)) FROM LINEITEM GROUP BY LINEITEM."L_QUANTITY";
```

Query N°13

```
SELECT L_RETURNFLAG, COUNT(*) , SUM(L_EXTENDEDPRICE) - SUM(L_QUANTITY)
AS VL1, SUM(L_DISCOUNT) - SUM(L_TAX) AS VL2 , VL2 - VL1 AS DIFF FROM
LINEITEM GROUP BY L_RETURNFLAG HAVING DIFF < 0;
```

Query N°14

```
SELECT LINEITEM."L_RETURNFLAG", COUNT(*) AS VCOUNT,
(SUM(LINEITEM."L_DISCOUNT")) + (SUM(LINEITEM."L_QUANTITY")) -
(SUM(LINEITEM."L_TAX")) - (SUM(LINEITEM."L_EXTENDEDPRICE")),
(SUM(LINEITEM."L_DISCOUNT")) - (SUM(LINEITEM."L_TAX")),
```

```
(SUM(LINEITEM."L_EXTENDEDPRICE")) - (SUM(LINEITEM."L_QUANTITY")) FROM
LINEITEM GROUP BY LINEITEM."L_RETURNFLAG";
```

Query N°15

```
SELECT DATEPART('YY',L_SHIPDATE) AS PIPPO, L_QUANTITY,
SUM(L_EXTENDEDPRICE) FROM LINEITEM WHERE L_QUANTITY > 45 GROUP BY
PIPPO, L_QUANTITY;
```

Query N°16

```
SELECT DATEPART('YY', LINEITEM."L_SHIPDATE"), LINEITEM."L_QUANTITY",
COUNT(*) AS VCOUNT, SUM(LINEITEM."L_EXTENDEDPRICE") FROM LINEITEM
GROUP BY DATEPART('YY', LINEITEM."L_SHIPDATE"), LINEITEM."L_QUANTITY";
```

Query N°17

```
SELECT L_RETURNFLAG , SUM(L_QUANTITY * 55/10) AS SOMMA FROM LINEITEM
GROUP BY L_RETURNFLAG HAVING SOMMA > 0 ORDER BY SOMMA;
```

Query N°18

```
SELECT LINEITEM."L_RETURNFLAG", COUNT(*) AS VCOUNT,
SUM((LINEITEM."L_QUANTITY") * (5.5)) FROM LINEITEM GROUP BY
LINEITEM."L_RETURNFLAG";
```

Query N°19

```
SELECT L_RETURNFLAG , SUM(L_QUANTITY - L_TAX) AS SOMMA, SUM(L_QUANTITY
- L_DISCOUNT) AS SOMMA2, SOMMA2 - SOMMA AS SOMMAX FROM LINEITEM GROUP
BY L_RETURNFLAG HAVING SOMMAX < 0 ORDER BY SOMMA;
```

Query N°20

```
SELECT LINEITEM."L_RETURNFLAG", COUNT(*) AS VCOUNT,
(SUM(LINEITEM."L_QUANTITY")) - (SUM(LINEITEM."L_DISCOUNT")),
(SUM(LINEITEM."L_QUANTITY")) - (SUM(LINEITEM."L_TAX")),
(SUM(LINEITEM."L_TAX")) - (SUM(LINEITEM."L_DISCOUNT")) FROM LINEITEM
GROUP BY LINEITEM."L_RETURNFLAG";
```

Query N°21

```
SELECT L_QUANTITY *2 AS C1, L_QUANTITY AS C2, SUM(L_EXTENDEDPRICE)
FROM LINEITEM WHERE L_QUANTITY>45 GROUP BY C1,C2;
```

Query N°22

```
SELECT (LINEITEM."L_QUANTITY") * (2), LINEITEM."L_QUANTITY", COUNT(*)
AS VCOUNT, SUM(LINEITEM."L_EXTENDEDPRICE") FROM LINEITEM GROUP BY
(LINEITEM."L_QUANTITY") * (2), LINEITEM."L_QUANTITY";
```

Query N°23

```
SELECT '01' AS C1, L_QUANTITY AS C2, SUM(L_EXTENDEDPRICE) FROM
LINEITEM WHERE L_QUANTITY>35 GROUP BY C1,C2;
```

Query N°24

```
SELECT '01', LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
SUM(LINEITEM."L_EXTENDEDPRICE") FROM LINEITEM GROUP BY '01',
LINEITEM."L_QUANTITY";
```

Query N°25

```
SELECT 999 AS C1, L_QUANTITY AS C2, SUM(L_EXTENDEDPRICE) FROM LINEITEM
WHERE L_QUANTITY>35 GROUP BY C1,C2;
```

Query N°26

```
SELECT 999, LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
SUM(LINEITEM."L_EXTENDEDPRICE") FROM LINEITEM GROUP BY 999,
LINEITEM."L_QUANTITY";
```

Query N°27

```
SELECT L_QUANTITY,COUNT(*), ROUND(SUM(L_EXTENDEDPRICE),0) +
ROUND(SUM(L_EXTENDEDPRICE),0) AS SOMMA FROM LINEITEM GROUP BY
L_QUANTITY HAVING SOMMA > 100000;
```

Query N°28

```
SELECT LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
(ROUND(SUM(LINEITEM."L_EXTENDEDPRICE"), 0)) +
(ROUND(SUM(LINEITEM."L_EXTENDEDPRICE"), 0)) FROM LINEITEM GROUP BY
LINEITEM."L_QUANTITY";
```

Query N°29

```
SELECT L_QUANTITY,COUNT(*), SUM(L_EXTENDEDPRICE) +
SUM(L_EXTENDEDPRICE) AS SOMMA FROM LINEITEM GROUP BY L_QUANTITY HAVING
SOMMA < 1000;
```

Query N°30

```
SELECT LINEITEM."L_QUANTITY", COUNT(*) AS VCOUNT,
(SUM(LINEITEM."L_EXTENDEDPRICE")) + (SUM(LINEITEM."L_EXTENDEDPRICE"))
FROM LINEITEM GROUP BY LINEITEM."L_QUANTITY";
```

Appendice B. Test-set per la regola “Partizioni”

Di seguito sono elencate le interrogazioni utilizzate per il test della regola “Partizioni”. Ad ognuna di essa è associato un numero progressivo, utilizzato, nel capitolo 6, come identificativo della query.

Query N°1

```
SELECT L_ORDERKEY,L_PARTSUPPKEY, COUNT(*),SUM(L_EXTENDEDPRICE),
MAX(O_ORDERDATE) FROM LINEITEM, ORDERS, PARTSUPP WHERE L_ORDERKEY =
O_ORDERKEY AND L_PARTSUPPKEY = PS_PARTSUPPKEY AND L_RETURNFLAG='R' AND
O_ORDERDATE = '01/12/1994' GROUP BY L_ORDERKEY,L_PARTSUPPKEY;
```

Query N°2

```
SELECT LINEITEM.L_ORDERKEY, ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY,
CUSTOMER.C_CUSTKEY, CUSTOMER.C_NATIONKEY, LINEITEM.L_EXTENDEDPRICE
FROM LINEITEM, ORDERS, CUSTOMER WHERE
LINEITEM.L_ORDERKEY=ORDERS.O_ORDERKEY AND
ORDERS.O_CUSTKEY=CUSTOMER.C_CUSTKEY AND LINEITEM.L_RETURNFLAG='R' AND
ORDERS.O_CUSTKEY>149990 AND CUSTOMER.C_NATIONKEY>5 ORDER BY
ORDERS.O_CUSTKEY DESC;
```

Query N°3

```
SELECT L_ORDERKEY,L_PARTSUPPKEY,COUNT(*), SUM(L_EXTENDEDPRICE) FROM
LINEITEM, ORDERS, PARTSUPP WHERE L_ORDERKEY = O_ORDERKEY AND
L_PARTSUPPKEY = PS_PARTSUPPKEY AND L_RETURNFLAG='R' AND O_ORDERDATE =
'01/12/1994' GROUP BY L_ORDERKEY,L_PARTSUPPKEY;
```

Query N°4

```
SELECT LINEITEM.L_ORDERKEY, ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY,
CUSTOMER.C_CUSTKEY, CUSTOMER.C_NATIONKEY, LINEITEM.L_EXTENDEDPRICE
FROM LINEITEM, ORDERS, CUSTOMER WHERE
LINEITEM.L_ORDERKEY=ORDERS.O_ORDERKEY AND
ORDERS.O_CUSTKEY=CUSTOMER.C_CUSTKEY AND LINEITEM.L_RETURNFLAG='R' AND
ORDERS.O_CUSTKEY>149990 AND CUSTOMER.C_NATIONKEY>5 ORDER BY
CUSTOMER.C_CUSTKEY DESC;
```

Query N°5

```
SELECT L_ORDERKEY, COUNT(*),SUM(L_EXTENDEDPRICE), MAX(O_ORDERDATE)
FROM LINEITEM, ORDERS, PARTSUPP WHERE L_ORDERKEY = O_ORDERKEY AND
L_PARTSUPPKEY = PS_PARTSUPPKEY AND L_RETURNFLAG='R' AND O_ORDERDATE =
'01/12/1994' GROUP BY L_ORDERKEY;
```

Query N°6

```
SELECT L_ORDERKEY,COUNT(*), SUM(L_EXTENDEDPRICE) FROM LINEITEM,
ORDERS, PARTSUPP WHERE L_ORDERKEY = O_ORDERKEY AND L_PARTSUPPKEY =
PS_PARTSUPPKEY AND L_RETURNFLAG='R' AND O_ORDERDATE = '01/12/1994'
GROUP BY L_ORDERKEY;
```

Query N°7

```
SELECT C_NATIONKEY, SUM(L_EXTENDEDPRICE) FROM LINEITEM, ORDERS,
CUSTOMER WHERE L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY AND
L_RETURNFLAG='N' GROUP BY C_NATIONKEY;
```

Query N°8

```
SELECT C_NATIONKEY, O_ORDERDATE, SUM(L_EXTENDEDPRICE), MAX(O_CUSTKEY)
FROM LINEITEM, ORDERS, CUSTOMER WHERE L_ORDERKEY = O_ORDERKEY AND
O_CUSTKEY = C_CUSTKEY AND L_RETURNFLAG='R' AND O_ORDERDATE =
'01/12/1994' GROUP BY C_NATIONKEY, O_ORDERDATE;
```

Query N°9

```
SELECT C_NATIONKEY, SUM(L_EXTENDEDPRICE) FROM LINEITEM, ORDERS,
CUSTOMER WHERE L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY AND
L_RETURNFLAG='R' AND O_ORDERDATE = '01/12/1994' GROUP BY C_NATIONKEY;
```

Query N°10

```
SELECT L_SHIPDATE, O_ORDERDATE, SUM(L_EXTENDEDPRICE), MAX(O_CUSTKEY)
FROM LINEITEM, ORDERS, PARTSUPP WHERE L_ORDERKEY = O_ORDERKEY AND
L_PARTSUPPKEY = PS_PARTSUPPKEY AND L_RETURNFLAG='R' AND O_ORDERDATE =
'01/12/1994' GROUP BY L_SHIPDATE, O_ORDERDATE;
```

Query N°11

```
SELECT C_NATIONKEY, SUM(L_EXTENDEDPRICE) FROM LINEITEM, ORDERS,
CUSTOMER WHERE L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY AND
L_RETURNFLAG='R' GROUP BY C_NATIONKEY;
```

Query N°12

```
SELECT C_NATIONKEY, SUM(L_EXTENDEDPRICE) FROM LINEITEM, ORDERS,
CUSTOMER WHERE L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY AND
L_RETURNFLAG='A' GROUP BY C_NATIONKEY;
```

Query N°13

```
SELECT C_NATIONKEY, O_ORDERDATE, SUM(L_EXTENDEDPRICE), MAX(O_CUSTKEY)
FROM LINEITEM, ORDERS, CUSTOMER WHERE L_ORDERKEY = O_ORDERKEY AND
O_CUSTKEY = C_CUSTKEY AND L_RETURNFLAG='R' AND O_ORDERDATE >
'01/12/1994' GROUP BY C_NATIONKEY, O_ORDERDATE;
```

Query N°14

```
SELECT MAX(LINEITEM.L_ORDERKEY), MIN(ORDERS.O_ORDERKEY),
MAX(CUSTOMER.C_CUSTKEY), SUM(LINEITEM.L_EXTENDEDPRICE) FROM LINEITEM,
ORDERS, CUSTOMER WHERE LINEITEM.L_ORDERKEY=ORDERS.O_ORDERKEY AND
ORDERS.O_CUSTKEY=CUSTOMER.C_CUSTKEY AND LINEITEM.L_RETURNFLAG='R' AND
ORDERS.O_CUSTKEY>149990 AND CUSTOMER.C_NATIONKEY>5;
```

Query N°15

```
SELECT MAX(LINEITEM.L_ORDERKEY), MIN(ORDERS.O_ORDERKEY),
MAX(CUSTOMER.C_CUSTKEY), SUM(LINEITEM.L_EXTENDEDPRICE) FROM LINEITEM,
ORDERS, CUSTOMER WHERE LINEITEM.L_ORDERKEY=ORDERS.O_ORDERKEY AND
ORDERS.O_CUSTKEY=CUSTOMER.C_CUSTKEY AND LINEITEM.L_RETURNFLAG='R' AND
ORDERS.O_CUSTKEY>149990;
```

Query N°16

```
SELECT MAX(LINEITEM.L_ORDERKEY), MIN(ORDERS.O_ORDERKEY),
MAX(CUSTOMER.C_CUSTKEY), SUM(LINEITEM.L_EXTENDEDPRICE) FROM LINEITEM,
ORDERS, CUSTOMER WHERE LINEITEM.L_ORDERKEY=ORDERS.O_ORDERKEY AND
ORDERS.O_CUSTKEY=CUSTOMER.C_CUSTKEY AND LINEITEM.L_RETURNFLAG='R';
```

Query N°17

```
SELECT C_NATIONKEY, SUM(L_EXTENDEDPRICE), MAX(O_ORDERDATE) FROM
LINEITEM, ORDERS, CUSTOMER WHERE L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY
= C_CUSTKEY AND L_RETURNFLAG='R' AND O_ORDERDATE = '01/12/1994' GROUP
BY C_NATIONKEY;
```

Query N°18

```
SELECT MAX(LINEITEM.L_ORDERKEY), MIN(ORDERS.O_ORDERKEY),
MAX(CUSTOMER.C_CUSTKEY), SUM(LINEITEM.L_EXTENDEDPRICE) FROM LINEITEM,
ORDERS, CUSTOMER WHERE LINEITEM.L_ORDERKEY=ORDERS.O_ORDERKEY AND
ORDERS.O_CUSTKEY=CUSTOMER.C_CUSTKEY AND LINEITEM.L_RETURNFLAG='R' AND
CUSTOMER.C_NATIONKEY>5;
```

Query N°19

```
SELECT C_NATIONKEY, SUM(L_EXTENDEDPRICE), MAX(O_ORDERDATE) FROM
LINEITEM, ORDERS, CUSTOMER WHERE L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY
```

```
= C_CUSTKEY AND L_RETURNFLAG='R' AND O_ORDERDATE > '01/12/1994' GROUP
BY C_NATIONKEY;
```

Query N°20

```
SELECT L_QUANTITY, COUNT(*), SUM(L_EXTENDEDPRICE) FROM LINEITEM,
ORDERS, CUSTOMER WHERE L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY =
C_CUSTKEY AND L_RETURNFLAG='R' AND O_ORDERDATE > '01/12/1994' AND
C_NATIONKEY > 5 GROUP BY L_QUANTITY;
```

Query N°21

```
SELECT L_SUPPKEY, COUNT (*) FROM LINEITEM WHERE L_SHIPDATE =
'10/02/1995' GROUP BY L_SUPPKEY;
```

Query N°22

```
SELECT LINEITEM.L_ORDERKEY, ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY,
CUSTOMER.C_CUSTKEY, CUSTOMER.C_NATIONKEY, LINEITEM.L_EXTENDEDPRICE
FROM LINEITEM, ORDERS, CUSTOMER WHERE
LINEITEM.L_ORDERKEY=ORDERS.O_ORDERKEY AND
ORDERS.O_CUSTKEY=CUSTOMER.C_CUSTKEY AND LINEITEM.L_RETURNFLAG='R' AND
ORDERS.O_CUSTKEY>149990 AND CUSTOMER.C_NATIONKEY>5;
```

Query N°23

```
SELECT LINEITEM.L_ORDERKEY, ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY,
CUSTOMER.C_CUSTKEY, CUSTOMER.C_NATIONKEY, LINEITEM.L_EXTENDEDPRICE,
LINEITEM.L_RETURNFLAG FROM LINEITEM, ORDERS, CUSTOMER WHERE
LINEITEM.L_ORDERKEY=ORDERS.O_ORDERKEY AND
ORDERS.O_CUSTKEY=CUSTOMER.C_CUSTKEY AND LINEITEM.L_RETURNFLAG='R' AND
CUSTOMER.C_NATIONKEY>5;
```

Query N°24

```
SELECT LINEITEM.L_RETURNFLAG, LINEITEM.L_ORDERKEY, ORDERS.O_ORDERKEY,
ORDERS.O_CUSTKEY, CUSTOMER.C_CUSTKEY, CUSTOMER.C_NATIONKEY,
LINEITEM.L_EXTENDEDPRICE FROM LINEITEM, ORDERS, CUSTOMER WHERE
LINEITEM.L_ORDERKEY=ORDERS.O_ORDERKEY AND
ORDERS.O_CUSTKEY=CUSTOMER.C_CUSTKEY AND LINEITEM.L_RETURNFLAG='R' AND
ORDERS.O_CUSTKEY>149990;
```

Query N°25

```
SELECT LINEITEM.L_RETURNFLAG, LINEITEM.L_ORDERKEY, ORDERS.O_ORDERKEY,
ORDERS.O_CUSTKEY, CUSTOMER.C_CUSTKEY, CUSTOMER.C_NATIONKEY,
LINEITEM.L_EXTENDEDPRICE FROM LINEITEM, ORDERS, CUSTOMER WHERE
LINEITEM.L_ORDERKEY=ORDERS.O_ORDERKEY AND
ORDERS.O_CUSTKEY=CUSTOMER.C_CUSTKEY AND LINEITEM.L_RETURNFLAG='R';
```

Query N°26

```
SELECT * FROM LINEITEM WHERE L_RETURNFLAG = 'R';
```

Query N°27

```
SELECT * FROM LINEITEM WHERE L_RETURNFLAG = 'A';
```

Query N°28

```
SELECT * FROM LINEITEM WHERE L_RETURNFLAG = 'N';
```

Query N°29

```
SELECT * FROM LINEITEM WHERE L_SHIPDATE = '10/02/1995';
```

Appendice C. Query riscritte del test-set “Group By”

Di seguito è riportata, per ogni query appartenente al test-set “Group By”, la corrispondente query riscritta dal modulo ChkViews.

Query N°1 riscritta:

```
SELECT RONE_000."C_NATIONKEY",
RONE_000."Sum_Lineitem._L_Extendedprice__" /* AS
Sum("L_Extendedprice") */  FROM RONE_000 WHERE (TRUE) AND (TRUE)
```

Query N°2 riscritta:

```
SELECT RONE_000."C_NATIONKEY", RONE_000."VCOUNT" AS VCOUNT,
RONE_000."SUM_LINEITEM._L_EXTENDEDPRICE__" /* AS
Sum(Lineitem."L_Extendedprice") */  FROM RONE_000 WHERE (TRUE) AND
(TRUE)
```

Query N°3 riscritta:

```
SELECT RONE_001."L_QUANTITY", RONE_001."VCOUNT" /* AS Count(*) */ ,
RONE_001."Max_Orders._O_Orderdate__" /* AS Max("O_Orderdate") */  FROM
RONE_001 WHERE (TRUE) AND (RONE_001."L_QUANTITY" > 1)
```

Query N°4 riscritta:

```
SELECT RONE_001."L_QUANTITY", RONE_001."VCOUNT" AS VCOUNT,
RONE_001."MAX_ORDERS._O_ORDERDATE__" /* AS Max(Orders."O_Orderdate")
*/  FROM RONE_001 WHERE TRUE
```

Query N°5 riscritta:

```
SELECT RONE_002."Abs_Lineitem._L_Quantity__" /* AS Abs("L_Quantity")
*/  FROM RONE_002
```

Query N°6 riscritta:

```
SELECT RONE_002."ABS_LINEITEM._L_QUANTITY__" /* AS
Abs(Lineitem."L_Quantity") */ , RONE_002."VCOUNT" AS VCOUNT FROM
RONE_002
```

Query N°7 riscritta:

```
SELECT RONE_003."Sum_Lineitem._L_Extendedprice__" /* AS
Sum("L_Extendedprice") */ ,
RONE_003."Sum_Round_Lineitem._L_Extendedprice___0__" /* AS
Sum(Round("L_Extendedprice", 0)) */ ,
RONE_003."Round_Sum_Lineitem._L_Extendedprice____0_" /* AS
Round(Sum("L_Extendedprice"), 0) */ , RONE_003."L_QUANTITY" FROM
RONE_003
```

Query N°8 riscritta:

```
SELECT RONE_003."L_QUANTITY", RONE_003."VCOUNT" AS VCOUNT,
RONE_003."ROUND_SUM_LINEITEM._L_EXTENDEDPRICE____0_" /* AS
Round(Sum(Lineitem."L_Extendedprice"), 0) */ ,
RONE_003."SUM_LINEITEM._L_EXTENDEDPRICE__" /* AS
Sum(Lineitem."L_Extendedprice") */ ,
RONE_003."SUM_ROUND_LINEITEM._L_EXTENDEDPRICE___0__" /* AS
Sum(Round(Lineitem."L_Extendedprice", 0)) */  FROM RONE_003
```

Query N°9 riscritta:

```
SELECT RONE_004."O_ORDERDATE", RONE_004."Max_Lineitem._L_Shipdate__"
/* AS Max("L_Shipdate") */  FROM RONE_004 WHERE (TRUE) AND
(RONE_004."O_ORDERDATE" > '31/12/1996')
```

Query N°10 riscritta:

```
SELECT RONE_004."O_ORDERDATE", RONE_004."VCOUNT" AS VCOUNT,
RONE_004."MAX_LINEITEM._L_SHIPDATE__" /* AS Max(Lineitem."L_Shipdate")
*/  FROM RONE_004 WHERE TRUE
```

Query N°11 riscritta:

```
SELECT RONE_003."L_QUANTITY",
RONE_003."SUM_LINEITEM._L_EXTENDEDPRICE__" /* AS
Sum("L_Extendedprice") */ ,
RONE_003."SUM_ROUND_LINEITEM._L_EXTENDEDPRICE___0__" /* AS
Sum(Round("L_Extendedprice", 0)) */ ,
RONE_003."ROUND_SUM_LINEITEM._L_EXTENDEDPRICE____0_" /* AS
Round(Sum("L_Extendedprice"), 0) */  FROM RONE_003
```

Query N°12 riscritta:

```
SELECT RONE_003."L_QUANTITY", RONE_003."VCOUNT" AS VCOUNT,
RONE_003."ROUND_SUM_LINEITEM._L_EXTENDEDPRICE____0_" /* AS
Round(Sum(Lineitem."L_Extendedprice"), 0) */ ,
RONE_003."SUM_LINEITEM._L_EXTENDEDPRICE__" /* AS
Sum(Lineitem."L_Extendedprice") */ ,
```

```
RONE_003."SUM_ROUND_LINEITEM._L_EXTENDEDPRICE___0__" /* AS
Sum(Round(Lineitem."L_Extendedprice", 0)) */  FROM RONE_003
```

Query N°13 riscritta:

```
SELECT RONE_006."L_RETURNFLAG", RONE_006."VCOUNT" /* AS Count(*) */ ,
RONE_006."_SUM_LINEITEM._L_EXTENDEDPRICE_______SUM_LINEITEM._L_QUANTIT
Y___" AS VL1,
RONE_006."_SUM_LINEITEM._L_DISCOUNT_______SUM_LINEITEM._L_TAX___" AS
VL2,
RONE_006."_SUM_LINEITEM._L_DISCOUNT_______SUM_LINEITEM._L_QUANTITY____
___SUM_LINEITEM._L_TAX_______SUM_LINEITEM._L_EXTENDEDPRICE___" AS DIFF
FROM RONE_006 WHERE
(RONE_006."_SUM_LINEITEM._L_DISCOUNT_______SUM_LINEITEM._L_QUANTITY___
____SUM_LINEITEM._L_TAX_______SUM_LINEITEM._L_EXTENDEDPRICE___" <
0.00)
```

Query N°14 riscritta:

```
SELECT RONE_006."L_RETURNFLAG", RONE_006."VCOUNT" AS VCOUNT,
RONE_006."_SUM_LINEITEM._L_DISCOUNT_______SUM_LINEITEM._L_QUANTITY____
___SUM_LINEITEM._L_TAX_______SUM_LINEITEM._L_EXTENDEDPRICE___",
RONE_006."_SUM_LINEITEM._L_DISCOUNT_______SUM_LINEITEM._L_TAX___",
RONE_006."_SUM_LINEITEM._L_EXTENDEDPRICE_______SUM_LINEITEM._L_QUANTIT
Y___" FROM RONE_006
```

Query N°15 riscritta:

```
SELECT RONE_007."Datepart__YY___Lineitem._L_Shipdate__" AS PIPPO,
RONE_007."L_QUANTITY", RONE_007."Sum_Lineitem._L_Extendedprice__" /*
AS Sum("L_Extendedprice") */  FROM RONE_007 WHERE
RONE_007."L_QUANTITY" > 45
```

Query N°16 riscritta:

```
SELECT RONE_007."DATEPART__YY___LINEITEM._L_SHIPDATE__" /* AS
Datepart('YY', Lineitem."L_Shipdate") */ , RONE_007."L_QUANTITY",
RONE_007."VCOUNT" AS VCOUNT,
RONE_007."SUM_LINEITEM._L_EXTENDEDPRICE__" /* AS
Sum(Lineitem."L_Extendedprice") */  FROM RONE_007
```

Query N°17 riscritta:

```
SELECT RONE_008."L_RETURNFLAG",
RONE_008."Sum__Lineitem._L_Quantity______5.5__" AS SOMMA FROM RONE_008
WHERE (RONE_008."Sum__Lineitem._L_Quantity______5.5__" > 0) ORDER BY
RONE_008."Sum__Lineitem._L_Quantity______5.5__" ASCENDING
```

Query N°18 riscritta:

```
SELECT RONE_008."L_RETURNFLAG", RONE_008."VCOUNT" AS VCOUNT,
RONE_008."SUM__LINEITEM._L_QUANTITY______5.5__" /* AS
Sum((Lineitem."L_Quantity") * (5.5)) */  FROM RONE_008
```

Query N°19 riscritta:

```
SELECT RONE_009."L_RETURNFLAG",
RONE_009."_SUM_LINEITEM._L_QUANTITY_______SUM_LINEITEM._L_TAX___" AS
SOMMA,
RONE_009."_SUM_LINEITEM._L_QUANTITY_______SUM_LINEITEM._L_DISCOUNT___"
AS SOMMA2,
RONE_009."_SUM_LINEITEM._L_TAX_______SUM_LINEITEM._L_DISCOUNT___" AS
SOMMAX FROM RONE_009 WHERE
(RONE_009."_SUM_LINEITEM._L_TAX_______SUM_LINEITEM._L_DISCOUNT___" <
0.00) ORDER BY
RONE_009."_SUM_LINEITEM._L_QUANTITY_______SUM_LINEITEM._L_TAX___"
ASCENDING
```

Query N°20 riscritta:

```
SELECT RONE_009."L_RETURNFLAG", RONE_009."VCOUNT" AS VCOUNT,
RONE_009."_SUM_LINEITEM._L_QUANTITY_______SUM_LINEITEM._L_DISCOUNT___"
, RONE_009."_SUM_LINEITEM._L_QUANTITY_______SUM_LINEITEM._L_TAX___",
RONE_009."_SUM_LINEITEM._L_TAX_______SUM_LINEITEM._L_DISCOUNT___" FROM
RONE_009
```

Query N°21 riscritta:

```
SELECT RONE_010."_LINEITEM._L_QUANTITY______2_" AS C1,
RONE_010."L_QUANTITY" AS C2,
RONE_010."Sum_Lineitem._L_Extendedprice__" /* AS
Sum("L_Extendedprice") */  FROM RONE_010 WHERE RONE_010."L_QUANTITY" >
45
```

Query N°22 riscritta:

```
SELECT RONE_010."_LINEITEM._L_QUANTITY______2_",
RONE_010."L_QUANTITY", RONE_010."VCOUNT" AS VCOUNT,
RONE_010."SUM_LINEITEM._L_EXTENDEDPRICE__" /* AS
Sum(Lineitem."L_Extendedprice") */  FROM RONE_010
```

Query N°23 riscritta:

```
SELECT RONE_011."_01_" AS C1, RONE_011."L_QUANTITY" AS C2,
RONE_011."Sum_Lineitem._L_Extendedprice__" /* AS
Sum("L_Extendedprice") */  FROM RONE_011 WHERE RONE_011."L_QUANTITY" >
35
```

Query N°24 riscritta:

```
SELECT RONE_011."_01_", RONE_011."L_QUANTITY", RONE_011."VCOUNT" AS
VCOUNT, RONE_011."SUM_LINEITEM._L_EXTENDEDPRICE__" /* AS
Sum(Lineitem."L_Extendedprice") */  FROM RONE_011
```

Query N°25 riscritta:

```
SELECT RONE_012."999" AS C1, RONE_012."L_QUANTITY" AS C2,
RONE_012."Sum_Lineitem._L_Extendedprice__" /* AS
Sum("L_Extendedprice") */  FROM RONE_012 WHERE RONE_012."L_QUANTITY" >
35
```

Query N°26 riscritta:

```
SELECT RONE_012."999", RONE_012."L_QUANTITY", RONE_012."VCOUNT" AS
VCOUNT, RONE_012."SUM_LINEITEM._L_EXTENDEDPRICE__" /* AS
Sum(Lineitem."L_Extendedprice") */  FROM RONE_012
```

Query N°27 riscritta:

```
SELECT RONE_013."L_QUANTITY", RONE_013."VCOUNT" /* AS Count(*) */ ,
RONE_013."_ROUND_SUM_LINEITEM._L_EXTENDEDPRICE____0______ROUND_SUM_LIN
EITEM._L_EXTENDEDPRICE____0__" AS SOMMA FROM RONE_013 WHERE
(RONE_013."_ROUND_SUM_LINEITEM._L_EXTENDEDPRICE____0______ROUND_SUM_LI
NEITEM._L_EXTENDEDPRICE____0__" > 100000)
```

Query N°28 riscritta:

```
SELECT RONE_013."L_QUANTITY", RONE_013."VCOUNT" AS VCOUNT,
RONE_013."_ROUND_SUM_LINEITEM._L_EXTENDEDPRICE____0______ROUND_SUM_LIN
EITEM._L_EXTENDEDPRICE____0__" FROM RONE_013
```

Query N°29 riscritta:

```
SELECT RONE_014."L_QUANTITY", RONE_014."VCOUNT" /* AS Count(*) */ ,
RONE_014."_SUM_LINEITEM._L_EXTENDEDPRICE_______SUM_LINEITEM._L_EXTENDE
DPRICE___" AS SOMMA FROM RONE_014 WHERE
(RONE_014."_SUM_LINEITEM._L_EXTENDEDPRICE_______SUM_LINEITEM._L_EXTEND
EDPRICE___" < 1000.00)
```

Query N°30 riscritta:

```
SELECT RONE_014."L_QUANTITY", RONE_014."VCOUNT" AS VCOUNT,
RONE_014."_SUM_LINEITEM._L_EXTENDEDPRICE_______SUM_LINEITEM._L_EXTENDE
DPRICE___" FROM RONE_014
```

Appendice D. Query riscritte del test-set “Partizioni”

Di seguito è riportata, per ogni query appartenente al test-set “Partizioni”, la corrispondente query riscritta dal modulo ChkViews.

Query N°1 riscritta:

```
SELECT RTWO_000."L_ORDERKEY", RTWO_000."L_PARTSUPPKEY", COUNT(*) /* AS
Count(*) */ , SUM(RTWO_000."L_EXTENDEDPRICE") /* AS
Sum("L_Extendedprice") */ , MAX(ORDERS."O_ORDERDATE") /* AS
Max("O_Orderdate") */ FROM RTWO_000, ORDERS, PARTSUPP WHERE
(RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /* *Join */) AND
(RTWO_000."L_PARTSUPPKEY" = PARTSUPP."PS_PARTSUPPKEY" /* *Join */) AND
(TRUE) AND (ORDERS."O_ORDERDATE" = '01/12/1994') GROUP BY
RTWO_000."L_ORDERKEY", RTWO_000."L_PARTSUPPKEY"
```

Query N°2 riscritta:

```
SELECT RTWO_000."L_ORDERKEY", ORDERS."O_ORDERKEY", ORDERS."O_CUSTKEY",
CUSTOMER."C_CUSTKEY", CUSTOMER."C_NATIONKEY",
RTWO_000."L_EXTENDEDPRICE" FROM RTWO_000, ORDERS, CUSTOMER WHERE
(RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /* *Join */) AND
(ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE) AND
(ORDERS."O_CUSTKEY" > 149990) AND (CUSTOMER."C_NATIONKEY" > 5) ORDER
BY ORDERS."O_CUSTKEY" DESCENDING
```

Query N°3 riscritta:

```
SELECT RTWO_000."L_ORDERKEY", RTWO_000."L_PARTSUPPKEY", COUNT(*) /* AS
Count(*) */ , SUM(RTWO_000."L_EXTENDEDPRICE") /* AS
Sum("L_Extendedprice") */ FROM RTWO_000, ORDERS, PARTSUPP WHERE
(RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /* *Join */) AND
(RTWO_000."L_PARTSUPPKEY" = PARTSUPP."PS_PARTSUPPKEY" /* *Join */) AND
(TRUE) AND (ORDERS."O_ORDERDATE" = '01/12/1994') GROUP BY
RTWO_000."L_ORDERKEY", RTWO_000."L_PARTSUPPKEY"
```

Query N°4 riscritta:

```
SELECT RTWO_000."L_ORDERKEY", ORDERS."O_ORDERKEY", ORDERS."O_CUSTKEY",
CUSTOMER."C_CUSTKEY", CUSTOMER."C_NATIONKEY",
```

```
RTWO_000."L_EXTENDEDPRICE" FROM RTWO_000, ORDERS, CUSTOMER WHERE
(RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /* *Join */) AND
(ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE) AND
(ORDERS."O_CUSTKEY" > 149990) AND (CUSTOMER."C_NATIONKEY" > 5) ORDER
BY CUSTOMER."C_CUSTKEY" DESCENDING
```

Query N°5 riscritta:

```
SELECT RTWO_000."L_ORDERKEY", COUNT(*) /* AS Count(*) */ ,
SUM(RTWO_000."L_EXTENDEDPRICE") /* AS Sum("L_Extendedprice") */ ,
MAX(ORDERS."O_ORDERDATE") /* AS Max("O_Orderdate") */ FROM RTWO_000,
ORDERS, PARTSUPP WHERE (RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /*
*Join */) AND (RTWO_000."L_PARTSUPPKEY" = PARTSUPP."PS_PARTSUPPKEY" /*
*Join */) AND (TRUE) AND (ORDERS."O_ORDERDATE" = '01/12/1994') GROUP
BY RTWO_000."L_ORDERKEY"
```

Query N°6 riscritta:

```
SELECT RTWO_000."L_ORDERKEY", COUNT(*) /* AS Count(*) */ ,
SUM(RTWO_000."L_EXTENDEDPRICE") /* AS Sum("L_Extendedprice") */ FROM
RTWO_000, ORDERS, PARTSUPP WHERE (RTWO_000."L_ORDERKEY" =
ORDERS."O_ORDERKEY" /* *Join */) AND (RTWO_000."L_PARTSUPPKEY" =
PARTSUPP."PS_PARTSUPPKEY" /* *Join */) AND (TRUE) AND
(ORDERS."O_ORDERDATE" = '01/12/1994') GROUP BY RTWO_000."L_ORDERKEY"
```

Query N°7 riscritta:

```
SELECT CUSTOMER."C_NATIONKEY", SUM(RTWO_002."L_EXTENDEDPRICE") /* AS
Sum("L_Extendedprice") */  FROM RTWO_002, ORDERS, CUSTOMER WHERE
(RTWO_002."L_ORDERKEY" = ORDERS."O_ORDERKEY" /* *Join */) AND
(ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE)
GROUP BY CUSTOMER."C_NATIONKEY"
```

Query N°8 riscritta:

```
SELECT CUSTOMER."C_NATIONKEY", ORDERS."O_ORDERDATE",
SUM(RTWO_000."L_EXTENDEDPRICE") /* AS Sum("L_Extendedprice") */ ,
MAX(ORDERS."O_CUSTKEY") /* AS Max("O_Custkey") */  FROM RTWO_000,
ORDERS, CUSTOMER WHERE (RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /*
*Join */) AND (ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */)
AND (TRUE) AND (ORDERS."O_ORDERDATE" = '01/12/1994') GROUP BY
CUSTOMER."C_NATIONKEY", ORDERS."O_ORDERDATE"
```

Query N°9 riscritta:

```
SELECT CUSTOMER."C_NATIONKEY", SUM(RTWO_000."L_EXTENDEDPRICE") /* AS
Sum("L_Extendedprice") */  FROM RTWO_000, ORDERS, CUSTOMER WHERE
(RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /* *Join */) AND
```

```
(ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE) AND
(ORDERS."O_ORDERDATE" = '01/12/1994') GROUP BY CUSTOMER."C_NATIONKEY"
```

Query N°10 riscritta:

```
SELECT RTWO_000."L_SHIPDATE", ORDERS."O_ORDERDATE",
SUM(RTWO_000."L_EXTENDEDPRICE") /* AS Sum("L_Extendedprice") */ ,
MAX(ORDERS."O_CUSTKEY") /* AS Max("O_Custkey") */  FROM RTWO_000,
ORDERS, PARTSUPP WHERE (RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /*
*Join */) AND (RTWO_000."L_PARTSUPPKEY" = PARTSUPP."PS_PARTSUPPKEY" /*
*Join */) AND (TRUE) AND (ORDERS."O_ORDERDATE" = '01/12/1994') GROUP
BY RTWO_000."L_SHIPDATE", ORDERS."O_ORDERDATE"
```

Query N°11 riscritta:

```
SELECT CUSTOMER."C_NATIONKEY", SUM(RTWO_000."L_EXTENDEDPRICE") /* AS
Sum("L_Extendedprice") */  FROM RTWO_000, ORDERS, CUSTOMER WHERE
(RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /* *Join */) AND
(ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE)
GROUP BY CUSTOMER."C_NATIONKEY"
```

Query N°12 riscritta:

```
SELECT CUSTOMER."C_NATIONKEY", SUM(RTWO_001."L_EXTENDEDPRICE") /* AS
Sum("L_Extendedprice") */  FROM RTWO_001, ORDERS, CUSTOMER WHERE
(RTWO_001."L_ORDERKEY" = ORDERS."O_ORDERKEY" /* *Join */) AND
(ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE)
GROUP BY CUSTOMER."C_NATIONKEY"
```

Query N°13 riscritta:

```
SELECT CUSTOMER."C_NATIONKEY", ORDERS."O_ORDERDATE",
SUM(RTWO_000."L_EXTENDEDPRICE") /* AS Sum("L_Extendedprice") */ ,
MAX(ORDERS."O_CUSTKEY") /* AS Max("O_Custkey") */  FROM RTWO_000,
ORDERS, CUSTOMER WHERE (RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /*
*Join */) AND (ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */)
AND (TRUE) AND (ORDERS."O_ORDERDATE" > '01/12/1994') GROUP BY
CUSTOMER."C_NATIONKEY", ORDERS."O_ORDERDATE"
```

Query N°14 riscritta:

```
SELECT MAX(RTWO_000."L_ORDERKEY") /* AS Max(Lineitem."L_Orderkey") */
, MIN(ORDERS."O_ORDERKEY") /* AS Min(Orders."O_Orderkey") */ ,
MAX(CUSTOMER."C_CUSTKEY") /* AS Max(Customer."C_Custkey") */ ,
SUM(RTWO_000."L_EXTENDEDPRICE") /* AS Sum(Lineitem."L_Extendedprice")
*/  FROM RTWO_000, ORDERS, CUSTOMER WHERE (RTWO_000."L_ORDERKEY" =
ORDERS."O_ORDERKEY" /* *Join */) AND (ORDERS."O_CUSTKEY" =
CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE) AND (ORDERS."O_CUSTKEY" >
149990) AND (CUSTOMER."C_NATIONKEY" > 5)
```

Query N°15 riscritta:

```
SELECT MAX(RTWO_000."L_ORDERKEY") /* AS Max(Lineitem."L_Orderkey") */
, MIN(ORDERS."O_ORDERKEY") /* AS Min(Orders."O_Orderkey") */ ,
MAX(CUSTOMER."C_CUSTKEY") /* AS Max(Customer."C_Custkey") */ ,
SUM(RTWO_000."L_EXTENDEDPRICE") /* AS Sum(Lineitem."L_Extendedprice")
*/  FROM RTWO_000, ORDERS, CUSTOMER WHERE (RTWO_000."L_ORDERKEY" =
ORDERS."O_ORDERKEY" /* *Join */) AND (ORDERS."O_CUSTKEY" =
CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE) AND (ORDERS."O_CUSTKEY" >
149990)
```

Query N°16 riscritta:

```
SELECT MAX(RTWO_000."L_ORDERKEY") /* AS Max(Lineitem."L_Orderkey") */
, MIN(ORDERS."O_ORDERKEY") /* AS Min(Orders."O_Orderkey") */ ,
MAX(CUSTOMER."C_CUSTKEY") /* AS Max(Customer."C_Custkey") */ ,
SUM(RTWO_000."L_EXTENDEDPRICE") /* AS Sum(Lineitem."L_Extendedprice")
*/  FROM RTWO_000, ORDERS, CUSTOMER WHERE (RTWO_000."L_ORDERKEY" =
ORDERS."O_ORDERKEY" /* *Join */) AND (ORDERS."O_CUSTKEY" =
CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE)
```

Query N°17 riscritta:

```
SELECT CUSTOMER."C_NATIONKEY", SUM(RTWO_000."L_EXTENDEDPRICE") /* AS
Sum("L_Extendedprice") */ , MAX(ORDERS."O_ORDERDATE") /* AS
Max("O_Orderdate") */  FROM RTWO_000, ORDERS, CUSTOMER WHERE
(RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /* *Join */) AND
(ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE) AND
(ORDERS."O_ORDERDATE" = '01/12/1994') GROUP BY CUSTOMER."C_NATIONKEY"
```

Query N°18 riscritta:

```
SELECT MAX(RTWO_000."L_ORDERKEY") /* AS Max(Lineitem."L_Orderkey") */
, MIN(ORDERS."O_ORDERKEY") /* AS Min(Orders."O_Orderkey") */ ,
MAX(CUSTOMER."C_CUSTKEY") /* AS Max(Customer."C_Custkey") */ ,
SUM(RTWO_000."L_EXTENDEDPRICE") /* AS Sum(Lineitem."L_Extendedprice")
*/  FROM RTWO_000, ORDERS, CUSTOMER WHERE (RTWO_000."L_ORDERKEY" =
ORDERS."O_ORDERKEY" /* *Join */) AND (ORDERS."O_CUSTKEY" =
CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE) AND
(CUSTOMER."C_NATIONKEY" > 5)
```

Query N°19 riscritta:

```
SELECT CUSTOMER."C_NATIONKEY", SUM(RTWO_000."L_EXTENDEDPRICE") /* AS
Sum("L_Extendedprice") */ , MAX(ORDERS."O_ORDERDATE") /* AS
Max("O_Orderdate") */  FROM RTWO_000, ORDERS, CUSTOMER WHERE
(RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /* *Join */) AND
(ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE) AND
(ORDERS."O_ORDERDATE" > '01/12/1994') GROUP BY CUSTOMER."C_NATIONKEY"
```

Query N°20 riscritta:

```
SELECT RTWO_000."L_QUANTITY", COUNT(*) /* AS Count(*) */ ,
SUM(RTWO_000."L_EXTENDEDPRICE") /* AS Sum("L_Extendedprice") */  FROM
RTWO_000, ORDERS, CUSTOMER WHERE (RTWO_000."L_ORDERKEY" =
ORDERS."O_ORDERKEY" /* *Join */) AND (ORDERS."O_CUSTKEY" =
CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE) AND (ORDERS."O_ORDERDATE"
> '01/12/1994') AND (CUSTOMER."C_NATIONKEY" > 5) GROUP BY
RTWO_000."L_QUANTITY"
```

Query N°21 riscritta:

```
SELECT RTWO_003."L_SUPPKEY", COUNT(*) /* AS Count(*) */  FROM RTWO_003
WHERE TRUE GROUP BY RTWO_003."L_SUPPKEY"
```

Query N°22 riscritta:

```
SELECT RTWO_000."L_ORDERKEY", ORDERS."O_ORDERKEY", ORDERS."O_CUSTKEY",
CUSTOMER."C_CUSTKEY", CUSTOMER."C_NATIONKEY",
RTWO_000."L_EXTENDEDPRICE" FROM RTWO_000, ORDERS, CUSTOMER WHERE
(RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /* *Join */) AND
(ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */) AND (TRUE) AND
(ORDERS."O_CUSTKEY" > 149990) AND (CUSTOMER."C_NATIONKEY" > 5)
```

Query N°23 riscritta:

```
SELECT RTWO_000."L_ORDERKEY", ORDERS."O_ORDERKEY", ORDERS."O_CUSTKEY",
CUSTOMER."C_CUSTKEY", CUSTOMER."C_NATIONKEY",
RTWO_000."L_EXTENDEDPRICE", RTWO_000."L_RETURNFLAG" FROM RTWO_000,
ORDERS, CUSTOMER WHERE (RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /*
*Join */) AND (ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */)
AND (TRUE) AND (CUSTOMER."C_NATIONKEY" > 5)
```

Query N°24 riscritta:

```
SELECT RTWO_000."L_RETURNFLAG", RTWO_000."L_ORDERKEY",
ORDERS."O_ORDERKEY", ORDERS."O_CUSTKEY", CUSTOMER."C_CUSTKEY",
CUSTOMER."C_NATIONKEY", RTWO_000."L_EXTENDEDPRICE" FROM RTWO_000,
ORDERS, CUSTOMER WHERE (RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /*
*Join */) AND (ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */)
AND (TRUE) AND (ORDERS."O_CUSTKEY" > 149990)
```

Query N°25 riscritta:

```
SELECT RTWO_000."L_RETURNFLAG", RTWO_000."L_ORDERKEY",
ORDERS."O_ORDERKEY", ORDERS."O_CUSTKEY", CUSTOMER."C_CUSTKEY",
CUSTOMER."C_NATIONKEY", RTWO_000."L_EXTENDEDPRICE" FROM RTWO_000,
ORDERS, CUSTOMER WHERE (RTWO_000."L_ORDERKEY" = ORDERS."O_ORDERKEY" /*
```

```
*Join */) AND (ORDERS."O_CUSTKEY" = CUSTOMER."C_CUSTKEY" /* *Join */)
AND (TRUE)
```

Query N°26 riscritta:

```
SELECT * FROM RTWO_000 WHERE TRUE
```

Query N°27 riscritta:

```
SELECT * FROM RTWO_001 WHERE TRUE
```

Query N°28 riscritta:

```
SELECT * FROM RTWO_002 WHERE TRUE
```

Query N°29 riscritta:

```
SELECT * FROM RTWO_003 WHERE TRUE
```

Bibliografia

[1] Wikimedia Foundation. (2010, Dicembre) Wikipedia. [Online]. http://it.wikipedia.org/wiki/Business_intelligence

[2] M. Golfarelli and S. Rizzi, Data Warehouse, teoria e pratica della progettazione, 2006.

[3] Wikimedia Foundation. (2010, Dicembre) Wikipedia. [Online]. http://it.wikipedia.org/wiki/Data_warehouse

[4] E. F. Codd, "Adding value to relational and legacy DBMS: the OLAP mandate," 1994.

[5] N. Pendse and R. Creeth. (2005) The OLAP-report – What is OLAP? An analysis of what the often misused OLAP term is supposed to mean. [Online]. http://www.olapreport.com/fasmi.htm

[6] Jedox AG. (2010) PALO. [Online]. http://www.jedox.com/

[7] A. Albano, G. Ghelli, and R. Orsini, *Fondamenti di basi di dati*. Bologna: Zanichelli, 2005.

[8] R. Elmasri and S. Navathe, *Sistemi di basi di dati Fondamenti, 4°ed*. Milano: Addison-Wesley, 2004.

[9] A. Albano, *Costruire sistemi per basi di dati*. Milano: Addison-Wesley, 2001.

[10] ARB. (2010) The OpenMP API specification for parallel programming. [Online]. http://openmp.org/

[11] A. Albano, SADAS An Innovative Column-Oriented DBMS for Business Intelligence applications.

[12] A. Meduna, *Automata and Languages*.: Springer London Ltd, 1999.

[13] GNU. (2010) The Lex & Yacc Page. [Online]. http://dinosaur.compilertools.net/

[14] D. Calvanese, G. De Giacomo, and M. Lenzerini, "What is view-based query rewriting?".

[15] R. Pottinger and A. Levy, "A scalable algorithm for answering queriesusing views," *VLDB*, 2000.

[16] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi, "Answering regular path queries using views," *ICDE*, 2000.

[17] G. Grahne and A.O. Mendelzon, "Tableau techniques for querying information sources through global schemas," *ICDT*, 1999.

[18] A.Y. Levy, A. Rajaraman, and J.J. Ordille, "Querying heterogeneous information sources using source descriptions," *VLDB*, 1996.

[19] H.Z. Yang and P.A. Larson, "Query transformation for PSJ-queries," *VLDB*, 1987.

[20] O.G. Tsatalos, M.H. Solomon, and Y.E. Ioannidis, "The GMAP: a versatile tool for physical data indipendence," *VLDB*, 1996.

[21] D.D. Florescu, "Search spaces for object-oriented query optimization," *PhD thesis. University of Paris VI*, 1996.

[22] V. Harinarayan, A. Rajaraman, and J.D. Ullman, "Implementing data cubes efficiently," *SIGMOD*, 1996.

[23] D. Theodoratos and T. Sellis, "Data Warehouse configuration," *VLDB*, 1997.

[24] H. Gupta, V. Harinarayan, A. Rajaraman, and J.D. Ullman, "Index selection for OLAP," *ICDE*, 1997.

[25] J. Yang, K. Karlapalem, and Q. Li, "Algorithms for materialized view design in Data Warehousing environment," *VLDB*, 1997.

[26] S. Dar, M.J. Franklin, B. Jonsson, D. Srivasta, and M. Tan, "Semantic data caching and replacement," *VLDB*, 1996.

[27] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez, "Caching strategies for data-intensive web sities," *VLDB*, 2000.

[28] A.Y. Halevy, "Answering queries using views: a survey".

[29] D. Srivastava, S. Dar, H.V. Jagadish, and A.Y. Levy, "Answering SQL queries using materialized views," *VLDB*, 1996.

[30] A. Gupta, V. Harinarayan, and D. Quass, "Aggregate-query processing in Data Warehousing environments," 1995.

[31] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata, "Answering complex SQL queries using automaticsummary tables," 2000.

[32] J. Goldstein and P.A. Larson, "Optimizing queries using materialized views: a practical, scalable solution," 2001.

[33] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing queries with materialized views," 1955.

[34] A. Deutsch, L. Popa, and V. Tannen, "Physical data independence, constraints and optimization with universal plans," 1999.

[35] R. Bello et al., "Materialized views in Oracle," 1998.

[36] F. Afrati, C. Li, and J. Ullman, "Generating efficient plans for queries using views," 2001.

[37] Dimitri van Heesch. (1997-2010) Doxygen. [Online]. http://www.doxygen.org/

[38] Wikimedia Foundation. (2011, Gennaio) Wikipedia. [Online]. http://it.wikipedia.org/wiki/Hash