D2I

Integrazione, Warehousing e Mining di sorgenti eterogenee

Programma di ricerca (cofinanziato dal MURST, esercizio 2000)

Techniques for logical design and efficient querying of data warehouses

P. CIACCIA, M. GOLFARELLI, A. MAZZITELLI, S. RIZZI, F. SCARCELLO

D2.R4 **2 gennaio 2002**

Sommario

Logical design of data warehouses (DW) encompasses the sequence of steps which, given a core workload, determine the logical scheme for the DW. A key step in logical design is view materialization. In this paper we propose an original approach to materialization in which the workload is characterized by the presence of complex queries represented by Nested Generalized Projection/Selection/Join expressions, in which sequences of aggregate operators may be applied to measures and selection predicates may be defined, at different granularities, on both dimensions and measures. Then, we propose a novel approach to estimate the cardinality of views based on a-priori information derived from the application domain. We face the problem by first computing satisfactory bounds for the cardinality, then by determining a good probabilistic estimate for it. The results we present here concern the computation of upper bounds for the cardinality of a view considering a set of cardinality constraints expressed on some other views. Finally, we deal with the problem of populating and refreshing the data warehouse, which typically involves queries spanning several tables over the reconciled schema. We present a structural method, based on the notion of hypertree decomposition, for solving these queries efficiently. Then, we extend this method in order to take into account also quantitative information on the data values.

Tema	Tema 2: Progettazione e interrogazione di Data Warehouse
Codice	D2.R4
Data	2 gennaio 2002
Tipo di prodotto	Rapporto Tecnico
Numero di pagine	34
Unità responsabile	BO
Unità coinvolte	CS
Autore da contattare	Stefano Rizzi DEIS Università di Bologna Viale Risorgimento 2, 40136 Bologna, Italia srizzi@deis.unibo.it

Techniques for logical design and efficient querying of data warehouses

P. Ciaccia, M. Golfarelli, A. Mazzitelli, S. Rizzi, F. Scarcello

2 gennaio 2002

Abstract

Logical design of data warehouses (DW) encompasses the sequence of steps which, given a core workload, determine the logical scheme for the DW. A key step in logical design is view materialization. In this paper we propose an original approach to materialization in which the workload is characterized by the presence of complex queries represented by Nested Generalized Projection/Selection/Join expressions, in which sequences of aggregate operators may be applied to measures and selection predicates may be defined, at different granularities, on both dimensions and measures. Then, we propose a novel approach to estimate the cardinality of views based on a-priori information derived from the application domain. We face the problem by first computing satisfactory bounds for the cardinality, then by determining a good probabilistic estimate for it. The results we present here concern the computation of upper bounds for the cardinality of a view considering a set of cardinality constraints expressed on some other views. Finally, we deal with the problem of populating and refreshing the data warehouse, which typically involves queries spanning several tables over the reconciled schema. We present a structural method, based on the notion of hypertree decomposition, for solving these queries efficiently. Then, we extend this method in order to take into account also quantitative information on the data values.

1 Introduction

A *data warehouse* (DW) is a repository which provides an integrated environment for decision support. From a conceptual point of view, a DW is organized according to the so-called *multidimensional model*[2]; thus, it represents facts of interest for the decision process into *cubes* in which each cell contains numerical *measures* which quantify the fact from different points of view, while each axis represents an interesting *dimension* for analysis. For example, in a sales cube modeling a chain store, time, product and store are typical dimensions while quantity sold and retail price are measures. On the other hand, within a 4-dimensional cube modeling the phone calls supported by a telecommunication company, the dimensions might be the calling number, the number called, the date, and the time segment in which the call is placed; each cube cell could be associated to a measure of the total duration of the calls made from a given number to another number on a given time segment and date.

The basic mechanism to extract significant information from the huge quantity of fine-grained data stored in base cubes is aggregation according to hierarchies of *attributes* rooted in dimensions [32], which produces aggregate cubes called *views*. Within a view, the values of measures are summarized according to some aggregation pattern, i.e., to a set of attributes defining the coarseness of aggregation; in particular, they are obtained by applying one or more aggregation operators to the data in the base fact table or in another view with a finer pattern. In most application cases, cubes are significantly sparse (for instance, most couples of telephone numbers are never connected by a call in a given date), and so are the views.

Logical design of data warehouses (DW) encompasses the sequence of steps which, starting from the conceptual scheme and given a core workload and the target logical model, determine the logical scheme for the DW. While during conceptual design the designer decides which portion of the application

domain is to be included in each data mart, and how data marts will be perceived and accessed by final users, during logical design (s)he defines which structures will be used to store information and how their performance can be optimized. It is important to emphasize that logical design of DWs requires completely different techniques than those used for operational databases, since their characteristics and objectives are deeply different. The target logical model we consider here is the relational model.

A key step in optimizing the performance of relational DWs consists in materializing a set of views which the system can use to solve the workload more efficiently. Of course, the larger the number of queries a view can be used to answer, the more effective materializing it [11, 59]. Since pre-computing all the possible views is unfeasible, several techniques to select an appropriate subset of views to materialize have been proposed. While most approaches only focus on the aggregation patterns required by queries, very few works analyze how the presence of different operators affects aggregation. On the other hand, using only one operator (typically, the sum) to aggregate measures when materializing views, makes them useless for several queries.

In this paper we propose an original approach to materialization in which the workload is characterized by the presence of complex queries which cannot be effectively described only by their aggregation pattern. In particular, we consider queries represented by Nested Generalized Projection/Selection/Join (NGPSJ) expressions, in which sequences of aggregate operators may be applied to measures and selection predicates may be defined, at different granularities, on both dimensions and measures. As a result, the correspondence between measures in the views and in the base cube is not necessarily one-to-one: some measures in the base cube may not be present at all in a view, while others may correspond to several measures in the view; furthermore, additional (either support or derived) measures may be included in the view in order to correctly compute aggregations. Particular emphasis is given to the effects on materialization of the contemporary presence of multiple aggregation operators in a query. Under these assumptions, an efficient algorithm to determine the restricted set of views which could be usefully materialized (*candidate views*) is proposed. The algorithm builds a query view graph whose vertices represent candidate views and whose arcs denote the possibility of computing a view from another. The query view graph may then be fed into an optimization algorithm like the ones proposed in [28, 30] which select, from the set of the candidate views, the subset which minimizes the response to the workload under a given space constraint.

A crucial problem related to view materialization is that of accurately estimating the actual cardinality of each view [57, 53]. Since the number of possible views which can be derived by aggregating a cube is exponential in the number of attributes, most approaches assume that a constraint on the total disk space occupied by materialization is posed, and attempt to find the subset of views which contemporarily satisfies this constraint and minimizes the workload cost [18, 28, 33]. Another case where estimation of view cardinalities is relevant is index selection [29].

If the data warehouse has already been loaded, view cardinalities can be quite accurately estimated by using statistical techniques based, say, on histograms [42] or sampling [34]. However, such techniques cannot be applied at all if the data warehouse is still under development, and the estimation of view cardinalities is needed for design purposes. To obviate this, current approaches are based on estimation models that only exploit the cardinality of the base cube and that of the single attribute domains [45, 52], which however leads to significant overestimation.

In this paper we propose a novel approach to estimate the cardinality of views based on a-priori information derived from the application domain. Similarly to what is done when estimating the cardinality of projections in relational databases [10], we face the problem by first computing satisfactory bounds for the cardinality, then by capitalizing on these bounds to determine a good probabilistic estimate for it. The results we present here concern the computation of upper bounds for the cardinality of a view considering the functional dependencies between attributes of the multidimensional scheme and a set of cardinality constraints expressed on some other views. In particular, we propose a bounding strategy which achieves an effective trade-off between the tightness of the bounds produced and the computational complexity, and outline a branch-and-bound approach to compute it.

The cubes are populated by periodically launching batch queries on the reconciled operational database.

These queries acquire data of interest, rearrange them to comply with the multidimensional model, and possibly groups them to the required granularity. Differently from OLAP queries on the star schemes, feeding queries typically span on several tables in the reconciled scheme in order to update both dimension and fact tables; this is particularly true, since dimension tables are denormalized, if the attribute hierarchy they represent is large. As the warehouse is refreshed while off-line, querying efficiency becomes a relevant issue. Query optimizers based on *quantitative methods* examine a number of alternative plans for answering a given query and then choose the best one, according to some cost model and the available information on the data. A completely different approach to query answering is based on the identification of some *structural property* of queries (e.g., acyclicity), rather than on quantitative information about data values. Exploiting such properties is possible to answer large classes of queries in polynomial-time. We first show how to employ the structural notion of hypertree decomposition [22, 25, 50] for the efficient answering of queries having bounded hypertree width, and how to compute optimal hypertree decompositions. Then, we describe a technique that combines quantitative and structural methods, by exploiting the notion of hypertree decomposition for restricting the search to well-structured query plans that guarantee a polynomial-time upper bound, and data information for selecting those plans that minimize the estimated execution-time (w.r.t. to a given cost model).

The paper is organized as follows. Section 2 describes our approach to view materialization. Section 3 discusses the problem of estimating the cardinality of views. Section 4 deals with the optimization of queries for populating and refreshing data warehouses. Finally, Section 5 discusses the most interesting open issues.

2 View Materialization

In this section, our approach to materialization of NGPSJ views is described. After proposing a motivating example and briefly reviewing some related literature, we define NGPSJ expressions, outline the algorithm for determining candidate views and finally show some experimental results.

2.1 Background and Motivating Example

Definition 1 (Cube) *We call* cube $\mathcal{C}$ *a triple* (U, F, M) *where* U *is a set of* attributes, $F = \{A_i \rightarrow A_j | A_i, A_j \in U\}$ *is a set of functional dependencies which relate the attributes of* U *into a set of pairwise disjoint directed trees (*hierarchies*), and* M *is a set of* measures. *We call* dimensions *the attributes* $A_k \in U$ *in which the trees are rooted, i.e., such that* $\forall A_i \in U \ (A_i \rightarrow A_k) \notin F$*; let* $dim(\mathcal{C}) \subseteq U$ *denote the set of dimensions of* $\mathcal{C}$*.*

A cube implemented on a relational DBMS is usually organized according to the so-called star scheme [35], composed by one fact table containing the measures and one denormalized dimension table for each dimension of analysis.

Definition 2 (Pattern) *A* pattern *on cube* $\mathcal{C} = (U, F, M)$ *is a subset of attributes* $P \subseteq U$ *such that* $\forall A_i, A_j \in P \ (A_i \rightarrow A_j) \notin F^+$*, where* F^+ *denotes the set of all functional dependencies logically implied by* F*. We define on the set of all patterns on* $\mathcal{C}$ *a partial ordering called* roll-up*, denoted by* $\leq$*, such that* $P_i \leq P_j$ *iff* $P_j \rightarrow P_i$*.*

Definition 3 (View) *Given a cube* $\mathcal{C}$ *and a pattern* P *on it, a* view *on* $\mathcal{C}$ *at pattern* P *is a cube* $\mathcal{C}'$ *such that (1)* $dim(\mathcal{C}') = P$*; (2) the attributes and the functional dependencies are those included into the subtrees of the hierarchies of* $\mathcal{C}$ *rooted in the attributes of* P*; (3) the measures are defined by applying expressions to the measures in* $\mathcal{C}$*.*

Example 1 The Sales cube includes three hierarchies: Time, Store and Product.

$$U = \{\mathsf{TDay}, \mathsf{TMonth}, \mathsf{TYear}, \mathsf{PName}, \mathsf{PType}, \mathsf{PCategory}, \mathsf{SName}, \mathsf{SCity}\}$$
$$F = \{\mathsf{TDay} \rightarrow \mathsf{TMonth}, \mathsf{TMonth} \rightarrow \mathsf{TYear}, \mathsf{SName} \rightarrow \mathsf{SCity}, \mathsf{PName} \rightarrow \mathsf{PType}, \mathsf{PType} \rightarrow \mathsf{PCategory}\}$$
$$M = \{\mathsf{Qty}, \mathsf{Prc}\}$$

thus having $dim(\mathsf{Sales}) = \{\mathsf{TDay}, \mathsf{SName}, \mathsf{PName}\}$. Examples of patterns are $P_1 = \{\mathsf{TMonth}, \mathsf{SCity}, \mathsf{PType}\}$, $P_2 = \{\mathsf{TYear}, \mathsf{SName}\}$, and $P_3 = \{\mathsf{PCategory}\}$; the only roll-up relationship between these patterns if $P_3 \leq P_1$. The star scheme modeling the Sales cube is defined below:

PRODUCT(<u>ProductId</u>, PName, PType, PCategory)
STORE(<u>StoreId</u>, SName, SCity)
TIME(<u>TimeId</u>, TDay, TMonth, TYear)
SALE(<u>TimeId</u>, <u>StoreId</u>, <u>ProductId</u>, Qty, Prc)

Tuples in dimension tables are identified by surrogate keys; in the rest of the paper, for the sake of simplicity, surrogate keys will not be considered, assuming that tuples are identified by an attribute of the hierarchy. The simplest example of view on pattern $\{\mathsf{TYear}, \mathsf{SName}\}$ is that whose fact table is defined by the following SQL query:

```
SELECT TIME.TYear,STORE.SName,SUM(Qty),SUM(Prc)
FROM SALE,TIME,STORE
WHERE SALE.TimeId = TIME.TimeId
AND SALE.StoreId = STORE.StoreId
GROUP BY TIME.TYear,STORE.SName
```

□

Consider the following queries on the Sales cube:

- Q_1: "For each product, find the average selling price and the maximum total quantity sold for the stores which sold more then 1000 units of that product"
- Q_2: "For each month and for each type of food product, find the average selling price and the total revenue"
- Q_3: "For product of type beverage and for each year, find the total quantity sold and the average of the total monthly revenues"

Query Q_1 requires to first aggregate data on pattern $\{\mathsf{SName}, \mathsf{PName}\}$, calculating the average price and the sum of the quantities sold, then to select the stores for which this sum is greater than 1000, finally to further aggregate on $\{\mathsf{PName}\}$ calculating the maximum quantity and the average price. Query Q_2 aggregates foodstuff sales on $\{\mathsf{TMonth}, \mathsf{PType}\}$, calculating for each month the average price and the sum of the derived measure $\mathsf{R} = \mathsf{Qty} \times \mathsf{Prc}$. Query Q_3 requires to aggregate on $\{\mathsf{TMonth}, \mathsf{PName}\}$ calculating the monthly sums of the quantities and that of the revenues, then to aggregate on $\{\mathsf{TYear}, \mathsf{PName}\}$ calculating the sums of the monthly quantities and the averages of the monthly revenues.

These examples give rise to some considerations:

1. Different aggregate operators may be applied to the same measure (e.g., $\mathsf{SUM}(\mathsf{Qty})$, $\mathsf{MAX}(\mathsf{Qty})$).
2. Sequences of aggregate operators may be applied to a measure (e.g., the maximum of the sums).
3. Some aggregations may require support measures in order to be correctly calculated from partial aggregates (e.g., the average operator requires the cardinality of each partial aggregate).

4. Calculating a derived measure from its component measures may determine a wrong result if the query is not solved directly on the base cube (e.g., $\mathsf{SUM(Qty)} \times \mathsf{AVG(Prc)} \neq \mathsf{SUM(Qty} \times \mathsf{Prc)}$).

In all these cases, in order to take full advantage of materialization, it may be necessary to include additional measures in views. Besides, the results of aggregation are affected by the selections which operate on its source data.

2.2 Related Literature

Several works adopt graph-like structures to represent views, but a few focus on how to build it. In [54] an algorithm to build a *multiquery graph* is outlined; the approach deals with the problem of building base fact tables from operational databases. Besides, since only join, selection and classical projection operators are involved, and no aggregation semantics is introduced, the approach cannot be used to determine aggregate views.

The approach which is most closely related to ours is the one described in [3]. That work is improved by ours with reference to the class of queries considered (GPSJ queries with distributive operators vs. NGPSJ queries with distributive, algebraic and holistic operators) and to the computational cost of the graph-building algorithm. In [3] the complexity is always exponential since, for each query, all the possible evaluation strategies are built and organized into a graph; the query view graph is then obtained by applying a set of rules that prune non-candidate views. The complexity of our algorithm is exponential only in the worst case, to become polynomial in the best case.

Query nesting is considered in [46], where subqueries involve multiple dependent aggregates at multiple granularities (*multi-feature cubes*). Though interesting results concerning distributive and algebraic cubes are provided, the queries considered are quite different from NGPSJ queries, since no nesting of distinct aggregation operators at different granularities is possible.

2.3 Query Modeling

In principle, the workload for a DW is dynamic and unpredictable. A possible approach to cope with this fact consists in monitoring the actual workload while the DW is operating. Otherwise, the designer may try to determine a core workload a priori: in fact, on the one hand, the user typically knows in advance which kind of data analysis (s)he will carry out more often for decisional or statistical purposes; on the other, a substantial amount of queries are aimed at extracting summary data to fill standard reports.

Basically, the approaches to view materialization differ in the level of detail they adopt to model the queries in the workload. Some approaches only consider the aggregation patterns [4], while others analyze the query features in more detail [28].

Our approach falls in the second group; in particular, we consider queries represented, in the relational algebra, by *nested GPSJ expressions*. A GPSJ (Generalized Projection / Selection / Join) expression [27] is a selection σ_1 over a generalized projection π over a selection σ_2 over a set of joins χ: $\sigma_1\pi\sigma_2\chi$. The *generalized projection* operator, $\pi_{P,M}(R)$, is an extension of duplicate eliminating projection, where P denotes a pattern, i.e., the set of group-by attributes, and M denotes a set of aggregate operators applied to the attributes in R. Thus, GPSJ expressions extend select-joins expressions with aggregation grouping and group selection.

Nesting GPSJ expressions means using the result from an expression as the input for another; it adds expressive power to GPSJ expressions since it allows sequences of aggregate operators to be used on a measure. For instance, the queries in Section 2.1 can be formulated on the Sales star scheme as follows:

$$Q_1: \quad \pi_{\mathsf{PName,WAVG(P,C),MAX(Q)}}(\sigma_{\mathsf{Q<1000}}(\pi_{\mathsf{PName,SName,Q=SUM(Qty),P=AVG(Prc),C=COUNT(*)}}(\mathsf{JS})))$$

$$Q_2: \quad \pi_{\mathsf{PType,TMonth,AVG(Prc),SUM(Qty\cdot Prc)}}(\sigma_{\mathsf{PCategory='Foodstuffs'}}(\mathsf{JS}))$$

$$Q_3: \quad \pi_{\mathsf{TYear,PName,SUM(Q),AVG(R)}}(\pi_{\mathsf{TYear,TMonth,PName,Q=SUM(Qty),R=SUM(Qty\cdot Prc)}}(\sigma_{\mathsf{PType='Bev'}}(\mathsf{JS})))$$

where $\mathsf{JS} = \mathsf{SALE} \bowtie \mathsf{PRODUCT} \bowtie \mathsf{TIME} \bowtie \mathsf{STORE}$ and $WAVG(m, w)$ computes the weighted average of measure m based on the weights w. The SQL formulation of Q_1 is the following:

```
SELECT PN,SUM(P*C)/SUM(C),MAX(Q)
FROM (SELECT PRODUCT.PName AS PN,STORE.SName AS SN,
             SUM(SALE.Qty) AS Q,AVG(SALE.Prc) AS P,COUNT(*) AS C
      FROM SALE,PRODUCT,TIME,STORE
      WHERE (...join conditions...)
      GROUP BY PRODUCT.PName,STORE.SName )
WHERE Q > 1000
GROUP BY PN
```

The expressions above are in *normal form* [27], i.e.:

- the necessary joins are pushed below all projections and selections;
- projection and selections are coalesced as much as possible;
- selections on dimensional attributes are pushed below all projections and selections on measures.

Let cube $\mathcal{C}$ be modeled as a star scheme with base fact table FT and dimension tables $DT_1, \ldots DT_n$. An NGPSJ expression in normal form on $\mathcal{C}$ is structured as follows:

$$E: \quad \sigma_{PredM_h}(\pi_{P_h,M_h}(\ldots\sigma_{PredM_1}(\pi_{P_1,M_1}(\sigma_{PredA \wedge PredM_0}(J)))\ldots))$$

where:

- M_j $(j = 1, \ldots h)$ is a set of aggregate measures, each defined as $OP(m_1, m_{max_{OP}})$ where OP is an aggregate operator, max_{OP} is the number of arguments of OP and m_i is an algebraic expression involving measures in M_{j-1}; M_0 denotes the set of measures in $\mathcal{C}$.
- $PredM_j$ $(j = 0, \ldots h)$ is a conjunction/disjunction of Boolean predicates expressed on measures in M_j.
- P_j $(j = 1, \ldots h)$ is a pattern; it is $P_{j+1} \leq P_j$ for $j = 1, \ldots h-1$.
- $PredA$ is a conjunction/disjunction of Boolean predicates expressed on attributes of $\mathcal{C}$.
- $J = FT \bowtie DT_1 \ldots \bowtie DT_n$

From a logical point of view, an NGPSJ expression E on cube $\mathcal{C}$ defines a view on $\mathcal{C}$ whose set of dimensions and set of measures correspond, respectively, to the pattern P_h and to the measures returned by the outer (leftmost) projection of E. Expression E itself defines a semantics for the measures returned. From this point of view, the base cube $\mathcal{C}$ is defined by the trivial NGPSJ expression $FT \bowtie DT_1 \ldots \bowtie DT_n$.

We wish to emphasize that, in this paper, we are not interested in determining optimal execution plans. Expressions are written in normal form for the sake of clarity and in order to simplify their management in the algorithm for building the query view graph; the order in which operators appear within the expressions *does not* reflect the execution plans that will be used to calculate them. For instance, pushing all the joins below the other operators would obviously be inefficient in several cases, and pushing some projections/selections down might be highly preferable.

For the same reason, we will assume for simplicity that the fact table is always joined to *all* the dimension tables (though in some queries, depending on the pattern required, it may be possible to omit one or more of them; see Q_1, Q_2 and Q_3 for instance). As to projections, consistently with the definition of pattern we will write them in a concise form by dropping all the attributes functionally determined by other attributes. For instance, Q_3 will be rewritten in concise form as

$$Q_3: \quad \pi_{\mathsf{TYear,PName,SUM(Q),AVG(R)}}(\pi_{\mathsf{TMonth,PName,Q=SUM(Qty),R=SUM(Qty\cdot Prc)}}(\sigma_{\mathsf{PType='Bev'}}(\mathsf{JS})))$$

by dropping attribute TYear from the intermediate projection since $\mathsf{TMonth} \rightarrow \mathsf{TYear}$.

Definition 4 (Roll-up of NGPSJ expressions) *We define on the set of all NGPSJ expressions on cube $\mathcal{C}$ a partial ordering called* roll-up*, denoted by $\leq$, such that $E_i \leq E_j$ iff, by applying a sequence of generalized projections and selections to E_j, it is possible to obtain an NGPSJ expression equivalent to E_i. We will denote with $E_i \oplus E_j$ the least upper bound of E_i and E_j in the roll-up partial ordering* [1].

In practice, $E_i \leq E_j$ holds when every measure returned in E_i can be calculated from those returned in E_j. In evaluating roll-up relationships, three factors must be taken into account: aggregation patterns, selections on both dimension attributes and measures, and the aggregation operators applied to measures. For instance, expression

$$\pi_{\mathsf{PType,TYear,SUM(Q)}}(\sigma_{\mathsf{Q}>2000}(\pi_{\mathsf{PName,TMonth,Q=SUM(Qty)}}(\sigma_{\mathsf{PCategory='Foodstuffs'} \wedge \mathsf{TYear}>\text{'1997'}}(\mathsf{JS}))))$$

can be obtained from

$$\mathsf{R}: \quad \sigma_{\mathsf{Q}>1000}(\pi_{\mathsf{PName,TMonth,Q=SUM(Qty)}}(\mathsf{JS}))$$

by computing

$$\pi_{\mathsf{PType,TYear,SUM(Q)}}(\sigma_{\mathsf{PCategory='Foodstuffs'} \wedge \mathsf{TYear}>\text{'1997'} \wedge \mathsf{Q}>2000}(\mathsf{R})).$$

2.4 Aggregate Operators

An aggregate operator is a function mapping a multiset of values into a single value. An attempt to define rules for aggregation of measures is presented in [37] and [51]: measures are classified into *flow*, *stock* and *value-per-unit*, and a table defines when each class of measures can be aggregated depending on the operators used and on the type of hierarchy (temporal, non-temporal). A classification of aggregation functions which is relevant to our approach is presented in [26]:

- *Distributive*, that allows aggregates to be computed directly from partial aggregates.
- *Algebraic*, that require additional information (*support measures*) to calculate aggregates from partial aggregates.
- *Holistic*, that do not allow aggregates to be computed from partial aggregates through a finite number of support measures.

Other works demonstrating the importance of aggregation operators are [47], which studies the problem of determining if a conjunction of aggregation constraints is feasible, and [7], which presents a theoretical framework for studying aggregations from a declarative and operational point of view.

2.5 NGPSJ Views

View materialization consists in selecting, among all the possible views, those that optimize the response to the workload under both a disk space and a maintenance cost constraint; this can be formulated as a knapsack problem, and is obviously NP-hard. Most methods in the literature aim at determining the subset of views that could be useful with reference to the workload (*candidate views*), then adopt heuristic algorithms to find sub-optimal solutions [28, 30]. In [3], this is done with reference to distributive aggregate functions; neither derived/support measures nor nesting of GPSJ queries are considered. Within this paper we focus on determining candidate views defined by NGPSJ expressions; the exponential dimension of the space of the possible views makes this step more complex and crucial than that of selecting the candidate views to be materialized.

[1] We conjecture that two NGPSJ expressions are equivalent iff their normal forms are equal (except for equivalent rewritings of aggregation operators); this implies that the least upper bound of two expressions is unique

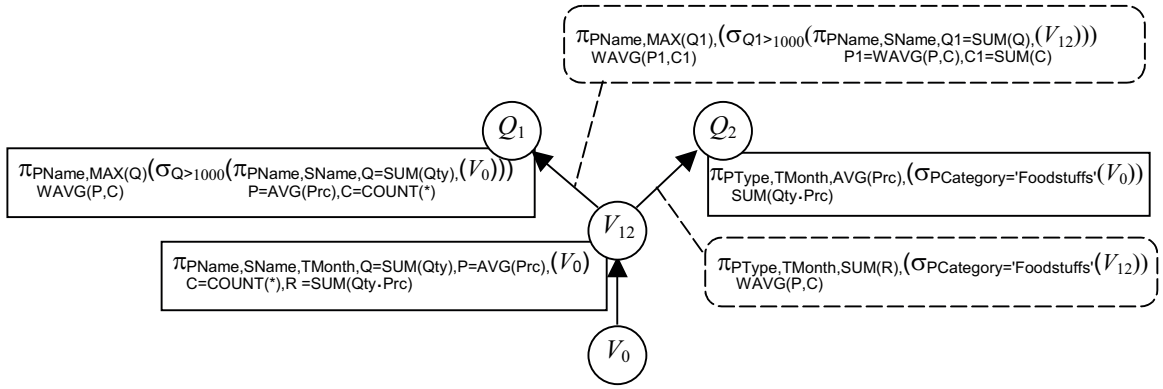

Figure 1: QVG for queries Q_1 and Q_2 (it is $V_0 : JS$)

Definition 5 (Candidate view) *Given a cube $\mathcal{C}$ and a workload $W = \{Q_1, \ldots Q_m\}$ on $\mathcal{C}$, a* candidate view *is defined by an NGPSJ expression V on $\mathcal{C}$ such that:*

$$(\exists Q_i \in W \mid V = Q_i) \vee (\exists Q_i, Q_j \in W \mid V = Q_i \oplus Q_j)$$

In [4] the authors prove that candidate views defined as above are the only relevant views for materialization, i.e., that materializing non-candidate views may never decrease the workload cost. In our approach, candidate views may:

- contain a subset of the tuples at a given pattern as a consequence of selections on dimension attributes and measures;
- contain only a subset of the measures in the base cube as a result of projections;
- include measures obtained by applying different aggregation sequences to the same measure;
- include derived measures and support measures necessary to support queries based on algebraic operators.

The widened definition of views given in our approach may lead to proliferation of measures; very large tables may entail poor performance. In order to avoid this, it may be convenient to adopt vertical partitioning techniques capable of splitting views into smaller fragments aimed at optimizing performance for a given workload [18].

The candidate views and the roll-up relationships between them can be represented in a graph:

Definition 6 (Query View Graph) *The* query view graph *(QVG) for workload W is the directed acyclic graph $\mathcal{G} = (VS, AS)$ such that:*

- *$VS = \{V_0, \ldots V_p\}$, where V_0 is the NGPSJ expression defining the base cube and $V_1, \ldots V_p$ are those defining the candidate views.*
- *Arc $A_{ij} \in AS$ denotes that $V_j \leq V_i$. Only the elemental roll-up relationships are represented in AS; those which can be obtained by transitivity are omitted.*

The QVG for a given workload is necessarily unique, since the set of candidate views is univocally determined by the set of queries. Figure 1 represents the QVG for queries Q_1 and Q_2 in Section 2.1; to help the reader, also the expressions which allow one view to be computed from another are shown (in dashed call-outs).

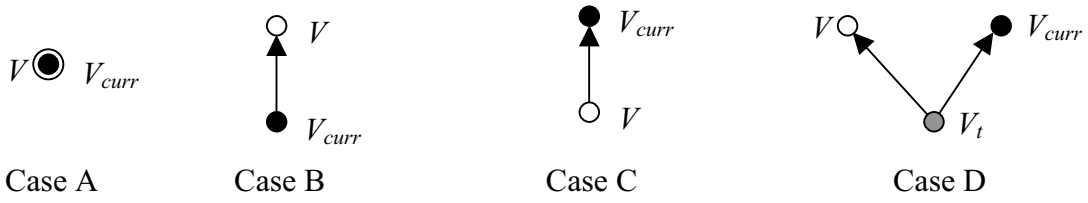

Figure 2: Different relationships between couples of views

2.6 Building the Query View Graph

The QVG for cube $\mathcal{C}$ and workload $W = \{Q_1, \ldots Q_m\}$ is built incrementally by merging each query with the QVG according to the expression which defines it:

$QVGBuilder(W, V_0)$:
{ $VS = \{V_0, Q_1\}$;
$AS = \{(V_0, Q_1)\}$;
$QVG = (VS, AS)$;
for each $Q_i \in W, i > 1$ **do**
{ $S = \{V_j \in VS \mid Child(V_j) = \emptyset\}$;
// S set of the leaves in QVG;
// function $Child(V_j)$ returns $\{V_k \in VS \mid (V_j, V_k) \in AS\}$
$VS \cup = \{Q_i\}$;
$Merge(S, Q_i, NULL)$;
}
}

Merging rules are aimed at determining if and how a query can be answered on a view belonging to the QVG. In general, the four possible relationships between two candidate views V and V_{curr} as entailed by Definition 4 are presented in Figure 2. In case A, V and V_{curr} are equivalent; in cases B and C, one of the views is less than the other. In case D no roll-up relationship can be established, and V_t is defined as $V \oplus V_{curr}$.

The problem of comparing two NGPSJ expressions is discussed in [19]. The procedure proposed can be seen as an iteration of the comparison between two GPSJ expressions, which was reported in [27]. Comparison proceeds from inside the two nested expressions, considering at each step two units having the form $\pi(\sigma(R))$; while in [27] R denotes a set of joins, here it denotes the result of the NGPSJ expressions analyzed up to the previous step.

The Merge algorithm finds the correct position for inserting V in the QVG, starting from its leaves and descending towards its root V_0. If V is already present (case A), no additional views must be inserted and Merge terminates. Otherwise, if while descending a path a roll-up relationship (case B) with V is found, the correct insertion position on that path has been reached; if a D relationship (roll-up relationship) among V and some view V_{curr} is found, the path is abandoned and V_{curr} is inserted into a set $Drel$ to be further examined. At the end, for each view V_j in $Drel$, a new candidate view $V_t = V_j \oplus V$ is determined and recursively merged into the QVG. The ExploreSubGraph function carries out a depth-first exploration of the sub-graph including all the predecessors of V_{curr}.

In Figure 3, the steps of the QVGBuilder algorithm for the queries proposed in Section 2.1 are shown. View V_{13} does not appear in the final QVG since it is equivalent to V_{12} (case A). As to V_{23}, the search is restricted to arc (Q_2, V_{12}) since a roll-up relationship (case B) is found and the other can be inferred.

If we restrict to considering only the aggregation patterns, the QVG resulting from the algorithm degenerates to the MDred Lattice presented in [4]. With reference to that approach, ours has a lower computation complexity (in terms of comparisons between expressions) since the roll-up relationships

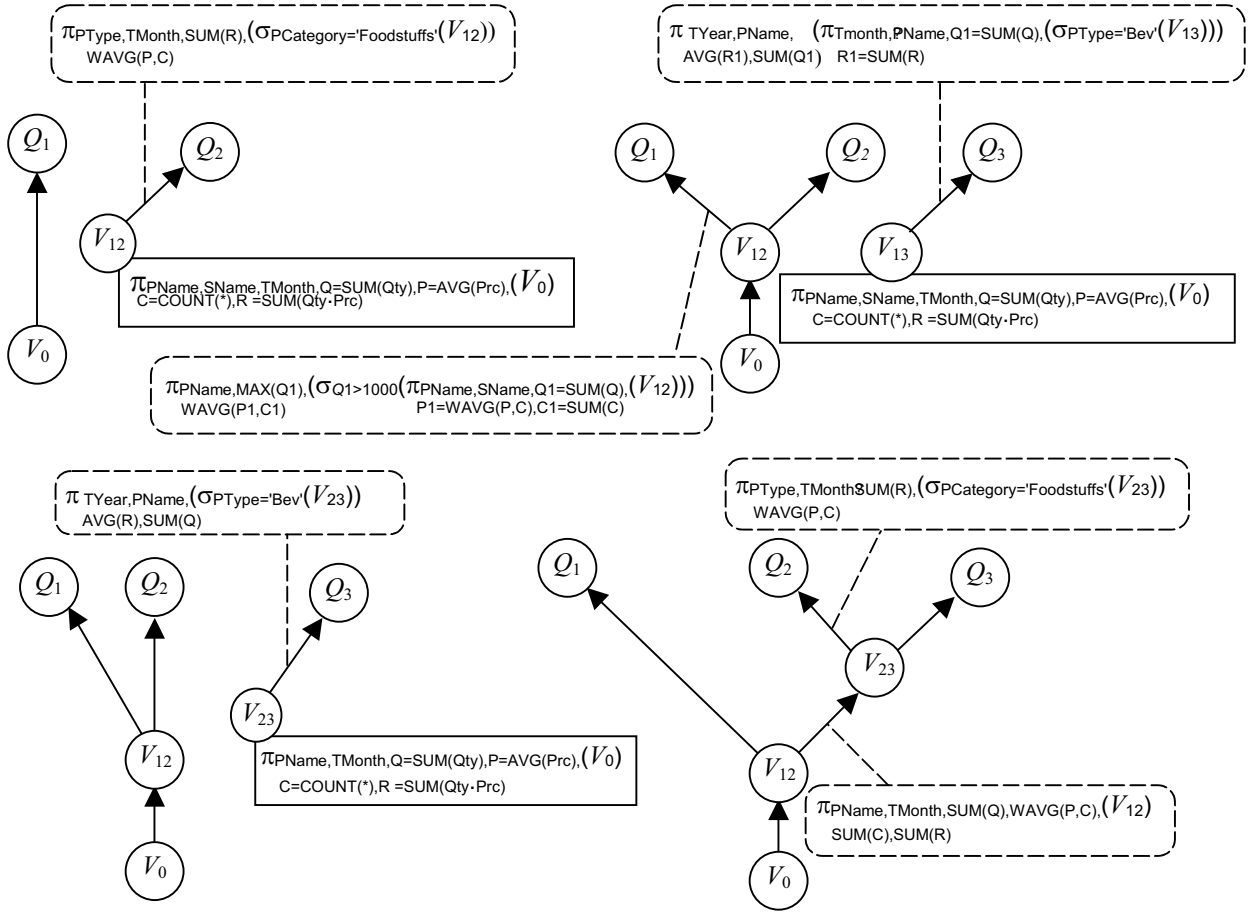

Figure 3: Construction of the QVG for queries Q_1, Q_2, Q_3

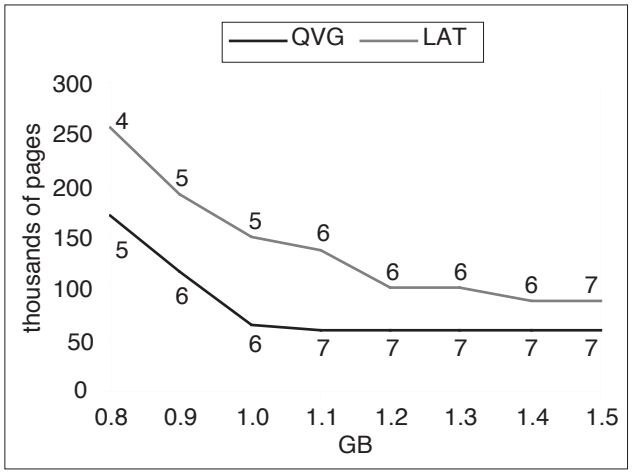

Figure 4: Execution costs for the QVG-based and the lattice-based approaches, on a sample workload, in function of the global space constraint. Besides each value point, the number of views materialized

expressed by the QVG are actively used to restrict the search to the sub-graph where the new vertex can be inserted. In fact, the complexity of Merge for adding the $(i+1)$-th query to the QVG reaches $O(2^i)$ only in the worst case, in which (1) the QVG already contains 2^i views and (2) all the candidate views solving both Q_{i+1} and each element in the power set of the i queries in the QVG must be created, i.e., relationship D holds for all the views in the QVG. In the best case the QVG degenerates into a path, and the complexity drops to $O(1)$. In the other cases, the complexity depends on the specific characteristics of the queries and on the order in which they are merged. The algorithm is efficient since the search space is pruned by neglecting all the vertices for which a roll-up relationship with the new vertex to be positioned can be inferred from the graph structure.

2.7 Experimental Tests

We tested our approach on the TPC-D benchmark; the workload includes four standard TPC-D queries and three additional NGPSJ queries. In order to demonstrate the effectiveness of our approach, we compared the results obtained by applying the same materialization heuristics [4] to our QVG and to the lattice defined in [4] (considering the same global space constraint). The cost function adopted expresses the total number of disk pages which must be accessed in order to solve each query, taking both the view size and the query selectivity into account. The results are summarized in Figure 4.

The low execution cost for the QVG-based approach is mainly due to the fact that, since NGPSJ views can express the nesting of aggregation operators, queries can be executed on views aggregated at a "higher" pattern. Besides, since NGPSJ views are more closely tailored on queries than classical views, more NGPSJ views typically fit the same space constraint. These considerations are further supported by observing how, in Figure 4, materializing the maximum number of useful NGPSJ views (7, one for each query in the workload) requires only 1.1 GB, and entails a lower cost than materializing the same number of classical views.

3 Cardinality Estimate

In this section, our approach to estimating the cardinality of views is described. After outlining our approach to estimation and showing its benefits with an example, we introduce the basic properties of bounds, propose an efficient bounding strategy, and sketch a branch-and-bound approach to determine the upper bound of a given view.

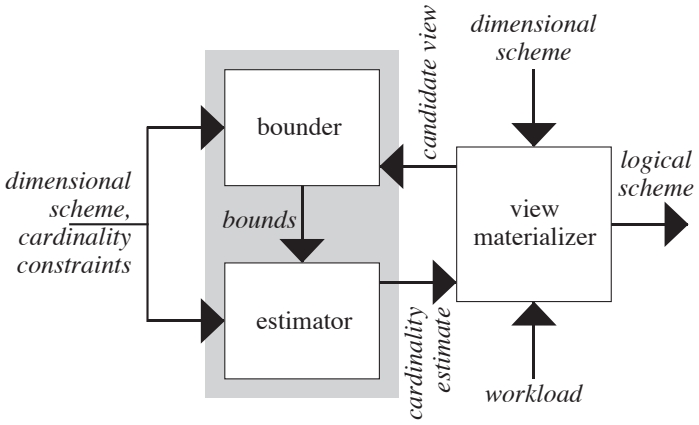

Figure 5: Overall architecture for logical design

3.1 Approach Overview and Motivating Example

The framework for this work is the logical design of data marts carried out off-line, i.e., assuming that the source data cannot be directly queried to estimate the cardinality of views. As sketched in Figure 5, whenever the materialization algorithm requires information about a candidate view V, our approach works in two steps. First, the *bounder* uses the set $\mathcal{I}$ of *cardinality constraints* supplied by the user to determine effective bounds for the cardinalities of a proper set of views; then, the *estimator* uses these bounds to derive a probabilistic estimate for the cardinality of V. Note that this two-steps approach generalizes well-known *parametric* models for the estimation of the cardinality of relational queries [40], and in particular those for projection size estimation [10], for which bounds are typically given as input parameters. Relevant examples of cardinality constraints that may be considered are the cardinality w of a view W, a lower and/or an upper bound (w^- and w^+, respectively) to the cardinality of W, and the ratio between the cardinalities of two views W_1 and W_2.

The set $\mathcal{I}$, together with cube $\mathcal{C}$, univocally determines two bounds for the cardinality of V, which are called the *greatest lower bound* and the *least upper bound*, denoted as v^- and v^+, respectively.[2] The interpretation of such bounds is as follows: (1) in each instance I of $\mathcal{C}$ that does not violate any constraint in $\mathcal{I}$, the cardinality v of V is such that $v \in [v^-, v^+]$; and (2) there exist two instances, both compatible with $\mathcal{I}$, where v equals v^- and v^+, respectively.

Computing the bounds implied by $\mathcal{I}$ turns out to be a challenging combinatorial problem, even for "simple" forms of cardinality constraints. For instance, it is known that the problem is NP-hard for arbitrary patterns of functional dependencies [9]. Furthermore, the actual computational effort needed to compute these bounds might limit applicability in real-world cases. For this reason, the bounder is built around the concept of *bounding strategy*. A bounding strategy s is characterized by a couple of bounding functions that, given $\mathcal{I}, \mathcal{C}$, and V, compute bounds v_{s}^- and v_{s}^+ such that $v_{\mathsf{s}}^- \leq v^-$ and $v^+ \leq v_{\mathsf{s}}^+$ both hold. In other terms, a bounding strategy never computes bounds which are more restrictive than the ones logically implied by the input constraints, trading-off accuracy for speed of evaluation. We say that a strategy s is *decoupled* iff computing v_{s}^+ for an arbitrary view V only requires the knowledge of upper bounds w_{s}^+ of other views W, but no knowledge of lower bounds w_{s}^-, and vice versa.

Turning to the estimator, our framework may support different *probabilistic models*. A probabilistic model is a function m that, given $\mathcal{I}, \mathcal{C}$, V, as well as bounds computed by the bounder, provides an estimate for the cardinality of V. In general, this step can use information from $\mathcal{I}$ that is not suitable to derive bounds. Typically this is the case where a cardinality constraint represents an average value (e.g., the number of calls originating on the average from a given number on each day is 10).

It is now necessary to report some considerations about the class of views we consider. The cardi-

[2] For simplicity of notation, we omit the dependence of bounds on $\mathcal{C}$ and $\mathcal{I}$.

nality v_1 of an NGPSJ view

$$V_1: \quad \sigma_{PredM_h}(\pi_{P_h,M_h}(\ldots\sigma_{PredM_1}(\pi_{P_1,M_1}(\sigma_{PredA\wedge PredM_0}(J)))\ldots))$$

is determined by three factors: its outer pattern P_h, the selection $PredA$ it operates on dimension attributes, and the selections $PredM_0, \ldots PredM_h$ it operates on measures at the different aggregation steps. Now, let $V_2 : \pi_{P_h,M_0}(J)$, where M_0 is the set of measures in the base cube, and let v_2 be its cardinality. While of course in general it will be $v_1 < v_2$ due to selection predicates, from the point of view of upper bounds we may assume that $v_1^+ \lesssim v_2^+$. In fact, for any predicate $PredM$ on measures we may write, there always exists an instance of the cube for which $PredM$ is true for all tuples. Similarly for the predicates on attributes. For this reason, in computing upper bounds we will consider only "simple" views including no selection predicates; thus, each view will be characterized only by its aggregation pattern. The Definitions of view and roll-up are then reformulated as follows:

Definition 7 (View) *Given a cube $\mathcal{C}$, a* view *on $\mathcal{C}$ is a pattern P on $\mathcal{C}$.*

Like for NGPSJ expressions, given two views V and W, both their sup and inf views always exist according to the roll-up partial ordering on patterns introduced in Definition 2; we will denote the sup with $V \oplus W$. Thus, roll-up determines a lattice, which we will call *multidimensional lattice* for $\mathcal{C}$, whose top and bottom elements are $dim(\mathcal{C})$ and the empty view $\emptyset$, respectively.

Example 2 A Calls cube modeling the phone calls supported by a telecommunication company might include:

$$U = \{\mathsf{date}, \mathsf{week}, \mathsf{month}, \mathsf{year}, \mathsf{sourceNumber}, \mathsf{sourceDistrict}, \mathsf{sourceState}, \mathsf{destNumber}, \mathsf{destDistrict}, \mathsf{destState}, \mathsf{timeSegment}\}$$
$$F = \{\mathsf{date} \rightarrow \mathsf{week}, \mathsf{date} \rightarrow \mathsf{month}, \mathsf{month} \rightarrow \mathsf{year}, \mathsf{sourceNumber} \rightarrow \mathsf{sourceDistrict}, \mathsf{sourceDistrict} \rightarrow \mathsf{sourceState}, \mathsf{destNumber} \rightarrow \mathsf{destDistrict}, \mathsf{destDistrict} \rightarrow \mathsf{destState}\}$$

thus having $dim(\mathcal{C}) = \{\mathsf{date}, \mathsf{sourceNumber}, \mathsf{destNumber}, \mathsf{timeSegment}\}$ as dimensions. Examples of views on the Calls cube are $V = \{\mathsf{month}, \mathsf{sourceNumber}, \mathsf{destState}\}$, $W = \{\mathsf{month}, \mathsf{sourceState}\}$ and $Z = \{\mathsf{year}, \mathsf{sourceDistrict}\}$. It is $W \oplus Z = \{\mathsf{month}, \mathsf{sourceDistrict}\}$. The roll-up relationships between these views are the following: $W \preceq W \oplus Z$, $Z \preceq W \oplus Z$, and $W \oplus Z \preceq V \preceq dim(\mathcal{C})$. □

Suppose that the telecommunication company domain serves 10^7 telephone numbers during a 10^3 days period on 5 daily time segments, and let $V = \{\mathsf{sourceNumber}, \mathsf{destNumber}, \mathsf{timeSegment}\}$. Using a simple bounding strategy, it is derived that $v \geq 10^7$, since at least one call is made from each number, and $v \leq 5 \times 10^{14}$, since at most each number calls each other number on each time segment. If the cardinality of the base cube is known, for instance it is 10^{11}, a bounding strategy could improve the upper bound of V to 10^{11}, since the cardinality of any aggregate view cannot exceed that of the base cube. Suppose now that the expert of the application domain is capable of providing an additional information: the number of distinct source-destination couples is at most 10^9. From this, we can infer that $v \leq 5 \times 10^9$.

3.2 A Decoupled Upper Bounding Strategy

In this subsection we focus on issues related to computing upper bounds by means of a decoupled bounding strategy and assuming that the set $\mathcal{I}$ just consists of a set of view cardinalities. In particular, for each $A_i \in U$ we assume that $card(A_i) = a_i \in \mathcal{I}$. This assumption, which is perfectly reasonable in all application domains, is necessary in order to guarantee that at least one upper/lower bound can be determined for each view. In addition, $\mathcal{I}$ may also include the cardinality w_j of some other view W_j. We say that a view is *ground* iff its cardinality is in $\mathcal{I}$.

The basic observation underlying the determination of effective bounds for view cardinalities is that the multidimensional lattice induces an isomorphic structure over such cardinalities. Let I be an instance of $\mathcal{C}$, and let v stand for the cardinality of view V in I. From Definition 2 it follows that $V \preceq W$ implies $v \leq w \; \forall I$, since $W \rightarrow V$ holds. This inequality also applies to upper bounds.

Lemma 1 *If $V \preceq W$, then $v^+ \leq w^+$.*

Proof: From Definition 2 it follows that $V \preceq W$ implies $v \leq w$ in each instance of $\mathcal{C}$, since $W \rightarrow V$ holds. Now, assume that $v^+ < w^+$. Then, there is an instance of $\mathcal{C}$ in which $v \leq v^+ < w \leq w^+$, thus $v < w$, which is a contradiction. □

The strategy we present strongly relies on the concept of *cover* of a view. Let $\mathcal{P}_\mathcal{C}$ be the set of all subsets of the views in cube $\mathcal{C}$.

Definition 8 (Cover) *Let V be a view on $\mathcal{C}$ and $S = \{W_1, \ldots W_m\} \in \mathcal{P}_\mathcal{C}$ be a set of views. S is called a* V-cover *iff $V \preceq \oplus(S)$. A V-cover is said to be* ground *when all the views it includes are ground.*

Example 3 In the Calls cube, let $V = \{\mathsf{sourceDistrict}, \mathsf{destDistrict}, \mathsf{timeSegment}, \mathsf{month}\}$ and $\mathcal{I} = \{w_0, w_1, w_2, w_3, w_4, w_5\}$ where

$$
\begin{aligned}
W_0 &= \{\mathsf{sourceNumber}, \mathsf{destNumber}, \mathsf{timeSegment}, \mathsf{date}\} \\
W_1 &= \{\mathsf{sourceNumber}, \mathsf{destDistrict}, \mathsf{timeSegment}, \mathsf{date}\} \\
W_2 &= \{\mathsf{sourceState}, \mathsf{destDistrict}, \mathsf{timeSegment}\} \\
W_3 &= \{\mathsf{timeSegment}, \mathsf{month}\} \\
W_4 &= \{\mathsf{sourceDistrict}, \mathsf{destDistrict}, \mathsf{date}\} \\
W_5 &= \{\mathsf{sourceNumber}, \mathsf{date}\}
\end{aligned}
$$

Some examples of ground V-covers are

$$
\begin{aligned}
S_1 &= \{W_1\} \\
S_2 &= \{W_2, W_4\} \\
S_3 &= \{W_3, W_4\} \\
S_4 &= \{W_2, W_5\};
\end{aligned}
$$

in fact, it is

$$
\begin{aligned}
V \preceq \oplus(S_1) &= \oplus(S_4) = W_1 \\
V \preceq \oplus(S_2) &= \oplus(S_3) = \{\mathsf{sourceDistrict}, \mathsf{destDistrict}, \mathsf{date}, \mathsf{timeSegment}\}
\end{aligned}
$$

In Figure 6 the roll-up relationships between these views are depicted. □

The notion of cover leads to generalize Lemma 1 to the case where several views at a time are used to bound from above the cardinality of V. Intuitively, this corresponds to viewing the problem as the one of determining the upper bound of the size of the (natural) join of the views in S.

Lemma 2 *Let V be a view and $S = \{W_1, \ldots W_m\}$ be a V-cover. Then: $v \leq u(S) \stackrel{\text{def}}{=} \prod_{j=1}^{m} w_j^+$.*

Proof: Immediate, since the least upper bound of a set of views corresponds to their natural join, and the size of the natural join of a set of views can never exceed that of their Cartesian product. □

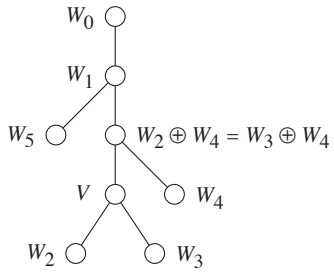

Figure 6: Roll-up relationships between views in Example 3

Coherently with Lemma 2, the *cover-based* strategy cb computes upper bounds as:

$$v^{+}_{\mathsf{cb}} = \begin{cases} v & \text{if } v \in \mathcal{C}, \\ \min\{u_{\mathsf{cb}}(S) \mid S \in \mathcal{P}_{\mathcal{C}}, V \preceq \oplus(S)\} & \text{if } v \notin \mathcal{C}. \end{cases}$$

where $u_{\mathsf{cb}}(S) = \prod_{j=1}^{m} w^{+}_{j,\mathsf{cb}}$.

Even for cubes with only a few attributes, computing v^{+}_{cb} by directly using this equation is not practically feasible, since the size of $\mathcal{P}_{\mathcal{C}}$ is $O(2^{2^N})$, where N is the number of attributes in $\mathcal{C}$. Fortunately, we can limit ourselves to consider only a restricted set of V-covers, which are called *minimal* V-covers and provide useful, non redundant, bounds. To see how minimal V-covers are determined, we need to consider two orthogonal aspects: a *domination* relationship between sets of views and the input information, $\mathcal{I}$. While the former induces a partial order on the bounds obtainable from V-covers, *regardless* of the specific input $\mathcal{I}$, the latter can be used to restrict the set of useful V-covers to only those consisting of ground views.

Definition 9 (Domination) *Let S_1 and S_2 be two sets of views. We say that S_1* dominates *S_2, written $S_1 \sqsubseteq S_2$, iff $u_{\mathsf{cb}}(S_1) \leq u_{\mathsf{cb}}(S_2)$ for every possible input set $\mathcal{I}$.*

Lemma 3 *Let $S_1 = \{W_{1,1}, \ldots, W_{1,i}, \ldots, W_{1,m}\}$ and $S_2 = \{W_{2,1}, \ldots, W_{2,j}, \ldots, W_{2,n}\}$ be two sets of views. If S_2 can be partitioned into m subsets $S_{2,1}, \ldots, S_{2,m}$ such that $W_{1,i} \preceq \oplus(S_{2,i})$ $\forall i = 1, \ldots, m$, then $S_1 \sqsubseteq S_2$.*

Proof: Immediate from Lemma 2 and from Definition 9. □

For instance, in Example 3 it is $S_1 \sqsubseteq S_4$, since $W_1 \preceq \oplus(S_4)$. Note that if $S_i \sqsubseteq S_j$ then $\oplus(S_i) \preceq \oplus(S_j)$ necessarily holds, whereas the opposite is not true in general (e.g., $S_3 \not\sqsubseteq S_4$ though $\oplus(S_3) \preceq \oplus(S_4)$).

Lemma 4 *Let S be a non-ground V-cover. Then there exists a ground V-cover S_1 such that $u_{\mathsf{cb}}(S_1) \leq u_{\mathsf{cb}}(S)$.*

Proof (sketch): Since S is not ground, at least one view in S is not ground. By recursively applying Lemma 2, $u_{\mathsf{cb}}(S)$ will be eventually expressed as a product of bounds in $\mathcal{I}$. The case of strict inequality ($u_{\mathsf{cb}}(S_1) < u_{\mathsf{cb}}(S)$) can arise since in this recursive process there is no guarantee that a given ground view will be generated just once, thus its least upper bound might appear more than once in $u_{\mathsf{cb}}(S)$. □

Definition 10 (Minimal Cover) *A ground V-cover S is* minimal *iff there is no other ground V-cover S_1 such that $S_1 \sqsubseteq S$ holds.*

The following theorem immediately derives from Definition 9 and Lemma 4.

Theorem 1 (Sufficiency of Minimal Covers) *It is:*

$$\min\{u_{\mathsf{cb}}(S) \mid S \text{ is a } V\text{-cover}\} = \min\{u_{\mathsf{cb}}(S) \mid S \text{ is a minimal } V\text{-cover}\}.$$

W_i	W_0	W_1	W_2	W_3	W_4	W_5
w_i	10^{11}	10^{10}	10^3	1.8×10^2	10^4	6×10^9
v^+_{cb}	10^{11}	10^{10}	9.36×10^6	9.36×10^6	1.8×10^6	1.8×10^6

Table 1: Improving upper bounds of v for increasing domain-derived information

Example 4 With reference to Example 3, S_1, S_2, S_3, and $S_5 = \{W_2, \{\mathsf{sourceDistrict}\}, \{\mathsf{month}\}\}$ are minimal V-covers. □

Although Theorem 1 states that v^+_{cb} can be determined by considering only minimal V-covers, Definition 9 does not directly provide a constructive rule to generate them. Nevertheless, based on the results obtained so far, it is possible to derive some rules aimed at reducing significantly the cardinality of the superset of minimal V-covers to be generated:

1. A ground view W such that $V \preceq W$ is a ground V-cover (from Definition 8).
2. A ground view W such that $arity(W) = 1$ and $W \cap V = \emptyset$ does not belong to any minimal V-cover[3] (from Definitions 9 and 10).
3. A ground view W such that $arity(W) > 1$ and $\forall W'$ for which $W' \preceq W$ it is $arity(W' \cap V) < 2$ does not belong to any minimal V-cover (since $\mathcal{C}$ includes the cardinalities of all the attributes).
4. If S is a ground V-cover, no set S' such that $S \subset S'$ is a minimal V-cover (from Definition 9).
5. If a minimal V-cover S contains a ground view W, it cannot contain any other ground view W' such that $W \preceq W'$ (from Definitions 9 and 10).

We approach the problem of computing the upper bound for a view V given a set of constraints $\mathcal{I}$ using a branch-and-bound algorithm which generates a superset of the minimal V-covers by solving a set of subproblems that repeatedly add new views to the partial solution obtained so far. Each subproblem is associated to: (1) a partial solution S containing all the ground views selected so far to build a ground V-cover; (2) an ordered set $T = \{W_i\}$ of the possible ground views to be added to S and compatible with S with reference to the rules above; (3) a function $lb(S)$ which returns a lower bound of the cardinality of the ground V-covers that can be obtained by extending S. The order on T is obtained by considering first the ground views for which the intersection with V has higher arity. It is remarkable that, if we impose that a partial solution containing a ground view W_i can be extended only with ground views W_j such that $j > i$, inducing a total order on T avoids the same cover to be generated twice; furthermore, the chosen order determines an heuristic criterion for generating the "most promising" covers first.

Example 5 We are interested in estimating $V = \{\mathsf{sourceDistrict}, \mathsf{destDistrict}, \mathsf{timeSegment}, \mathsf{month}\}$ in the Calls cube. Table 1 shows how the upper bound of v improves as additional cardinality constraints are supplied as input (see Example 3). The result when all six views are included in $\mathcal{I}$ (besides the cardinality of the views with arity 1) is obtained by building only 6 complete ground V-covers. Seven partial solutions are abandoned since dominated by the current best solution. □

4 Populating the DW

Population and refreshing of cubes are critical issues in DW initialization and management. Periodically, a number of batch queries are executed on the reconciled operational database. Note that these queries are typically very different from OLAP queries. Indeed, while the latter queries are executed on star schemes

[3] $arity(W)$ denotes the number of attributes in W.

(or similar simple schemes), these populating queries usually span several tables in the reconciled scheme in order to update both dimension and fact tables. Thus, they are very often long queries involving many join operations, plus selections, projections and, possibly, grouping and aggregate operators. In this context, the choice of a good query-execution strategy is therefore particularly relevant, because the differences among execution times can be several orders of magnitude large.

Query optimizers based on *quantitative methods* examine a number of alternative plans for answering a given query and then choose the best one, according to some cost model. These planners exploit information on the data, e.g., sizes of relations, indices, and so on. Recall that computing an optimal plan is an NP-hard problem and hence it is unlikely to find an efficient algorithm for selecting the best plans. Indeed, all the commercial DBMS just compute approximations of optimal query plans. See [50] for a short survey of quantitative methods and for further references.

A completely different approach to query answering is based on the identification of some structural properties of queries, rather than on quantitative information about data values. Exploiting such properties is possible to answer large classes of queries efficiently, i.e., with a polynomial-time upper bound. One of the most important and best studied class of tractable queries is the class of *acyclic queries* [6, 8, 12, 14, 15, 20, 36, 39, 43, 48, 49, 60, 61]. It was shown that acyclic queries coincide with the *tree queries* [5], see also [1, 38, 55]. The latter are queries which are representable by a *join tree* (or *join forest*) (see [50] for a formal definition).

By well-known results of Yannakakis [60], acyclic conjunctive queries are efficiently solvable. More precisely, all answers of an acyclic conjunctive query can be computed in time polynomial in the combined size of the input and the output. This is the best possible result, because in general the size of the answer may be exponentially greater than the size of the input. Recall that, for cyclic queries, even computing small outputs, e.g. just one tuple, or checking whether the answer of a query is non-empty (Boolean queries) requires exponential time (unless P = NP). The good results about acyclic conjunctive queries extend to very relevant classes of *nearly acyclic* queries, such as queries whose associated primal graph has *bounded treewidth* [44], a *cutset* [13] of bounded size, or a bounded *degree of cyclicity* [31]. Thus, a number of query answering techniques based on such structural properties have been proposed. Conceptually, all of them may be viewed as composed of two phases:

1. transform the given query into an equivalent acyclic query, and
2. solve this acyclic query.

We call these techniques *decomposition methods*, because they solve queries by decomposing them into acyclic ones.

It is worthwhile noting that quantitative and decomposition methods have been completely separated worlds. In particular, structural properties of queries have been deeply investigated in the literature, but most commercial DBMS only use quantitative methods for optimizing queries. However, many meaningful queries in our framework are quite long, but often acyclic or close to acyclic queries. Because of their sizes, it may happen that quantitative methods are unable to find optimal query plans and then take a large amount of time for answering queries that are structurally simple and easily solvable.

In the following sections we propose new techniques for answering queries efficiently by using the notion of hypertree decomposition [22]. It has been shown that the class of queries having bounded hypertree-width has the same desirable properties as the class of acyclic queries, and that this is the largest class of queries having these features (among the known methods described in both the AI and the database literatures) [16]. In particular, we first describe a pure decomposition method just based on the structure of the query, and then we show how to extend this technique in order to exploit quantitative information on relations, attribute selectivity, and so on.

For the sake of presentation, our description will focus on conjunctive queries, which are equivalent to SELECT-PROJECT-JOIN queries. However, our methods can be easily extended to more general queries, possibly involving aggregate operators.

4.1 Hypertree Decompositions and Tractable Queries

We next recall some basic definitions of queries, hypergraphs, and hypertree decompositions. For detailed descriptions of the latter notion, see [22, 25].

We will adopt the standard convention [1, 55] of identifying a relational database instance with a logical theory consisting of ground facts. Thus, a tuple $\langle a_1, \ldots a_k \rangle$, belonging to relation r, will be identified with the ground atom $r(a_1, \ldots, a_k)$. The fact that a tuple $\langle a_1, \ldots, a_k \rangle$ belongs to relation r of a database instance **DB** is thus simply denoted by $r(a_1, \ldots, a_k) \in \mathbf{DB}$.

A (rule-based) *conjunctive query* Q on a database schema $DS = \{R_1, \ldots, R_m\}$ consists of a rule of the form

$$Q: \ ans(\mathbf{u}) \leftarrow r_1(\mathbf{u_1}) \wedge \cdots \wedge r_n(\mathbf{u_n}),$$

where $n \geq 0$; $r_1, \ldots r_n$ are relation names (not necessarily distinct) of DS; ans is a relation name not in DS; and $\mathbf{u}, \mathbf{u_1}, \ldots, \mathbf{u_n}$ are lists of terms (i.e., variables or constants) of appropriate length. If $\mathbf{u}$ is an empty list, i.e., no variable occurs in the head, then Q is said a *Boolean* query. The set of variables occurring in Q is denoted by $var(Q)$. The set of atoms contained in the body of Q is referred to as $atoms(Q)$.

The *answer* of a non Boolean query Q on a database instance **DB** with associated universe U, denoted by $Q(\mathbf{DB})$, consists of a relation ans whose arity is equal to the length of $\mathbf{u}$, defined as follows. Relation ans contains all tuples $\mathbf{u}\theta$ such that $\theta : var(Q) \longrightarrow U$ is a substitution replacing each variable in $var(Q)$ by a value of U and such that for $1 \leq i \leq n$, $r_i(\mathbf{u_i})\theta \in \mathbf{DB}$. (For an atom A, $A\theta$ denotes the atom obtained from A by uniformly substituting $\theta(X)$ for each variable X occurring in A.) If Q is a Boolean query, then ans is a ground predicate and the answer $Q(\mathbf{DB})$ is "true" if there is at least one such a substitution θ satisfying the body of Q, otherwise $Q(\mathbf{DB})$ is "false."

If Q is a conjunctive query, we define the hypergraph $H(Q) = (V, E)$ associated to Q as follows. The set of vertices V consists of all variables occurring in Q. For each atom $r_i(\mathbf{u_i})$ in the body of Q, the set E contains a hyperedge consisting of all variables occurring in $\mathbf{u_i}$. Note that the cardinality of E can be smaller than the cardinality of *atoms(Q)* because two query atoms having exactly the same set of variables in their arguments give rise to only one edge in E. For example, the three query atoms $r(X, Y)$, $r(Y, X)$, and $s(X, X, Y)$ all correspond to a unique hyperedge $\{X, Y\}$.

Acyclic conjunctive queries were the object of a large number of investigations [6, 12, 14, 15, 20, 39, 43, 48, 49, 60]. In particular, it was shown that the class of acyclic queries coincides with the class of *tree queries* ([5], see also [1, 38, 55]). The latter are queries which are representable by a join tree (or join forest).

A query Q is acyclic if and only if its hypergraph $H(Q)$ is acyclic or, equivalently, if Q has a join forest.

A *join forest* for a query Q is a forest G whose set of vertices V_G is the set $atoms(Q)$ and such that, for each pair of atoms A_1 and A_2 in V_G having variables in common, the following conditions hold:

1. A_1 and A_2 belong to the same connected component of G, and
2. all variables common to A_1 and A_2 occur in every atom on the (unique) path in G from A_1 to A_2.

If G is a tree, then it is called a *join tree* for Q.

Acyclic queries have highly desirable computational properties:

1. Acyclic instances can be efficiently solved. Yannakakis provided a (sequential) polynomial time algorithm for answering acyclic queries[4] [60].
2. We have recently shown that answering queries is highly parallelizable on acyclic queries, as this problem (actually, the decision problem of answering Boolean queries) is complete for the low complexity class LOGCFL [21, 24]. Efficient parallel algorithms for Boolean and non-Boolean

[4]Note that, since both the database **DB** and the query Q are part of an input-instance, what we are considering is the *combined complexity* of the query [56].

queries have been proposed in [21, 24] and [23]. They run on parallel database machines that exploit the *inter-operation parallelism* [58], i.e., machines that execute different relational operations in parallel. These algorithms can be also employed for solving acyclic queries efficiently in a distributed environment.

3. Acyclicity is efficiently recognizable.

4. The result of a (non-Boolean) acyclic conjunctive query Q can be *computed* in time polynomial in the combined size of the input instance and of the output relation [60].

Intuitively, the efficient behavior of acyclic instances is due to the fact that they can be evaluated by processing any of their join trees bottom-up by performing upward semijoins, thus keeping small the size of the intermediate relations (which could become exponential if regular join were performed).

A new class of tractable conjunctive database queries, which generalizes the class of acyclic queries, has recently been identified [22]. This is the class of queries having a bounded-width hypertree decomposition [22]. These queries have all the properties described above for acyclic queries [22].

A *hypertree for a hypergraph* $\mathcal{H}$ is a triple $\langle T, \chi, \lambda \rangle$, where $T = (N, E)$ is a rooted tree, and χ and λ are labelling functions which associate to each vertex $p \in N$ two sets $\chi(p) \subseteq var(\mathcal{H})$ and $\lambda(p) \subseteq edges(\mathcal{H})$. If $T' = (N', E')$ is a subtree of T, we define $\chi(T') = \bigcup_{v \in N'} \chi(v)$. We denote the set of vertices N of T by $vertices(T)$, and the root of T by $root(T)$. Moreover, for any $p \in N$, T_p denotes the subtree of T rooted at p.

Definition 11 (Hypertree decomposition) A *hypertree decomposition* of a hypergraph $\mathcal{H}$ is a hypertree $HD = \langle T, \chi, \lambda \rangle$ for $\mathcal{H}$ which satisfies all the following conditions:

1. for each edge $h \in edges(\mathcal{H})$, there exists $p \in vertices(T)$ such that $h \subseteq \chi(p)$ (we say that p *covers* h);

2. for each variable $Y \in var(\mathcal{H})$, the set $\{p \in vertices(T) \mid Y \in \chi(p)\}$ induces a (connected) subtree of T;

3. for each $p \in vertices(T)$, $\chi(p) \subseteq var(\lambda(p))$;

4. for each $p \in vertices(T)$, $var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.

Note that the inclusion in Condition 4 is actually an equality, because Condition 3 implies the reverse inclusion.

An edge $h \in edges(\mathcal{H})$ is *strongly covered* in HD if there exists $p \in vertices(T)$ such that $var(h) \subseteq \chi(p)$ and $h \in \lambda(p)$. In this case, we say that p strongly covers h.

A hypertree decomposition HD of hypergraph $\mathcal{H}$ is a *complete decomposition* of $\mathcal{H}$ if every edge of $\mathcal{H}$ is strongly covered in HD.

The *width* of a hypertree decomposition $\langle T, \chi, \lambda \rangle$ is $max_{p \in vertices(T)} |\lambda(p)|$. The `HYPERTREE` *width* $hw(\mathcal{H})$ of $\mathcal{H}$ is the minimum width over all its hypertree decompositions. A c-width hypertree decomposition of $\mathcal{H}$ is *optimal* if $c = hw(\mathcal{H})$.

The acyclic hypergraphs are precisely those hypergraphs having hypertree width one. Indeed, any join tree of an acyclic hypergraph $\mathcal{H}$ trivially corresponds to a hypertree decomposition of $\mathcal{H}$ of width one. Furthermore, if a hypergraph $\mathcal{H}'$ has a hypertree decomposition of width one, then, from this decomposition, we can easily compute a join tree of $\mathcal{H}'$, which is therefore acyclic [22].

Remark 1 From any hypertree decomposition HD of $\mathcal{H}$, we can easily compute a complete hypertree decomposition of $\mathcal{H}$ having the same width. For any "missing" edge h, choose a vertex q of T such that $var(h) \subseteq \chi(q)$ (such a vertex must exist by Condition 1), and create a new vertex p as a child of q with $\lambda(p) = h$ and $\chi(p) = var(h)$. Assuming the use of suitable data structures, this computation can be done in $O(\|\mathcal{H}\| \cdot \|HD\|)$ time, where $\|HD\|$ denotes the size of a hypertree decomposition, i.e., the

number of bits needed for encoding HD (that is, for encoding the rooted tree of HD and, for each vertex v of this tree, the labellings χ and λ for v, encoded as a list of variables and a list of edge identifiers, respectively).

Intuitively, if $\mathcal{H}$ is a cyclic hypergraph, the χ labelling selects the set of variables to be fixed in order to split the cycles and achieve acyclicity; $\lambda(p)$ "covers" the variables of $\chi(p)$ by a set of edges.

Example 6 Consider the following conjunctive query Q_1:

$$ans \leftarrow j(J, X, Y, X', Y') \wedge a(S, X, X', C, F) \wedge b(S, Y, Y', C', F');$$
$$c(C, C', Z) \wedge d(X, Z) \wedge e(Y, Z) \wedge f(F, F', Z') \wedge g(X', Z') \wedge h(Y', Z').$$

Let $\mathcal{H}_1$ be its corresponding hypergraph. Since $\mathcal{H}_1$ is cyclic, $hw(\mathcal{H}_1) > 1$ holds. Figure 7 shows a (complete) hypertree decomposition HD_1 of $\mathcal{H}_1$ having width 2, hence $hw(\mathcal{H}_1) = 2$.

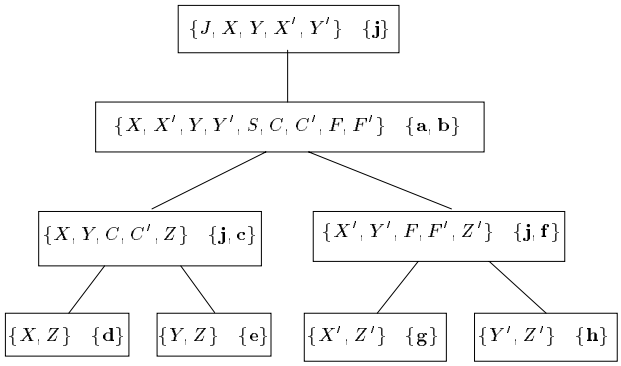

Figure 7: A 2-width hypertree decomposition of $\mathcal{H}_1$

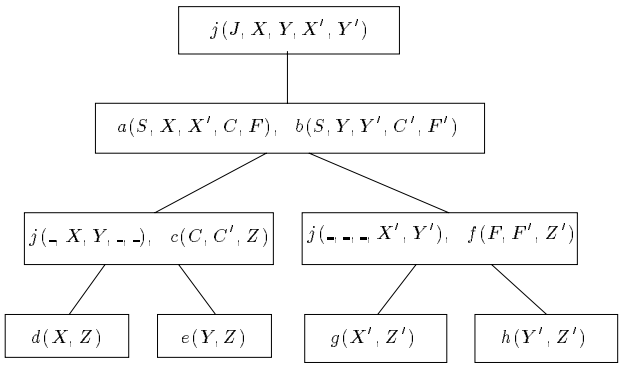

Figure 8: Hyperedge representation of hypertree decomposition HD_1

In order to help the intuition of what a hypertree decomposition is, we also present an alternative representation, called *hyperedge representation*. (Also, "atom representation.") Figure 8 shows the hyperedge representation of the hypertree decomposition HD_1 of $\mathcal{H}_1$. Each node p in the tree is labeled by a set of hyperedges representing $\lambda(p)$; $\chi(p)$ is the set of all variables, distinct from '_', appearing in these hyperedges. Thus, the anonymous variable '_' replaces the variables in $var(\lambda(p)) - \chi(p)$.

Using this representation, we can easily observe an important feature of hypertree decompositions. Once an hyperedge has been covered by some vertex of the decomposition tree, any subset of its variables can be used freely in order to decompose the remaining cycles in the hypergraph. For instance, the variables in the hyperedge corresponding to atom j in $\mathcal{H}_1$ are jointly included only in the root of the decomposition. If we were forced to take all the variables in every vertex where j occurs, it would not

be possible to find a decomposition of width 2. Indeed, in this case, any choice of two hyperedges per vertex yields a hypertree which violates the connectedness condition for variables (i.e., Condition 2 of Definition 11).

Let k be a fixed positive integer. We say that a CQ instance I has k-bounded HYPERTREE width if $hw(\mathcal{H}_I) \leq k$, where $\mathcal{H}_I$ is the hypergraph associated to I. From the results in [22], it follows that k-bounded hypertree width is efficiently decidable, and that a hypertree decomposition of width k can be efficiently computed (if any).

4.2 Efficient Query Evaluation

In this section, we show how to exploit hypertree decompositions for answering conjunctive queries having bounded hypertree-width. In particular we show that, for any constant k, conjunctive queries having hypertree width at most k can be efficiently answered, as they can be transformed, via a logspace procedure, to an equivalent acyclic conjunctive query.

The following lemma shows that, starting from a hypertree decomposition of a query Q, we can efficiently build a join tree of an acyclic query Q' equivalent to Q.

Lemma 5 *Let Q be a Boolean conjunctive query over a database* **DB***, and HD a hypertree decomposition of Q of width k. Then, there exists Q', $\mathbf{DB}'$, JT such that:*

1. *Q' is an acyclic (Boolean) conjunctive query evaluating to "true" on database $\mathbf{DB}'$ iff the answer of Q on* **DB** *is "true".*
2. *$||\langle Q', \mathbf{DB}', JT\rangle|| = O((||Q|| + ||HD||) \cdot r^k)$, where r denotes the maximum relation-size over the relations in* **DB**.
3. *JT is a join tree of the query Q'.*
4. *$\langle Q', \mathbf{DB}', JT\rangle$ is logspace computable from $\langle Q, \mathbf{DB}, HD\rangle$.*

Proof. Let Q be a Boolean conjunctive query over a database **DB**, and HD a hypertree decomposition of Q of width k. Without loss of generality, we assume that Q does not contain any atom A such that $var(A) = \emptyset$. We first transform HD, into a k-width complete hypertree decomposition $\hat{HD} = \langle T, \chi, \lambda\rangle$ of Q. This transformation is feasible in logspace, and $||\hat{HD}|| = O(||Q|| + ||HD||)$.

The query Q evaluates to *true* on **DB** if and only if $\bowtie_{A \in atoms(Q)} rel(A)$ is a non-empty relation, where $rel(A)$ denotes the relation of **DB** associated to the atom A, and $\bowtie$ is the natural join operation (with common variables acting as join attributes).

For each vertex $p \in vertices(T)$ define a Boolean query $Q(p)$ and a database $\mathbf{DB}(p)$ as follows. For each atom $A \in \lambda(p)$:

- If $var(A) \subseteq \chi(p)$, then A occurs in $Q(p)$ and $rel(A)$ belongs to $\mathbf{DB}(p)$;
- if $var(A) \not\subseteq \chi(p)$ and $(var(A) \cap \chi(p)) \neq \emptyset$, then $Q(p)$ contains a new atom A' such that $var(A') = var(A) \cap \chi(p)$, and $\mathbf{DB}(p)$ contains the corresponding relation $rel(A')$, which is the projection of $rel(A)$ on the set of attributes corresponding to the variables in $var(A')$;

Now, consider the following Boolean query $\bar{Q}$ on the database $\bar{\mathbf{DB}} = \bigcup_{p \in vertices(T)} \mathbf{DB}(p)$.

$$\bar{Q} : \bigwedge_{p \in vertices(T)} Q(p)$$

By the associative and commutative properties of natural joins, and by the fact that $\hat{HD}$ is a complete hypertree decomposition, it follows that $\bar{Q}$ on $\bar{\mathbf{DB}}$ is equivalent to Q on **DB**. To see this, note that $\bar{\mathbf{DB}}$ just contains some new relations which are projections of relations already occurring in **DB**, and $\bar{Q}$ contains

the atoms corresponding to these relations. Thus, no tuple can be lost by taking the additional joins corresponding to these relations. It follows that if Q evaluates to *true* on **DB**, then also $\bar{Q}$ evaluates to *true* on $\bar{\mathbf{DB}}$. On the other hand, since $\hat{HD}$ is a complete decomposition, every atom A of Q also occurs in $\bar{Q}$ (as A must occur in $\lambda(p)$ for some vertex p of T). Thus, if $\bar{Q}$ evaluates to *true* then also Q evaluates to *true*.

We build $\langle Q', \mathbf{DB}', JT\rangle$ as follows. JT has exactly the same tree shape of T. For each vertex p of T, there is precisely one vertex p' in JT, and one relation P' in $\mathbf{DB}'$. Then p' is an atom having $\chi(p)$ as arguments and its corresponding relation P' in $\mathbf{DB}'$ is the natural join of all atoms in $Q(p)$. $Q' = \bigwedge_{p\in JT} p'$ is the conjunction of all atoms corresponding to some vertex of JT.

Q' on $\mathbf{DB}'$ is clearly equivalent to $\bar{Q}$ on $\bar{\mathbf{DB}}$, and hence to Q on **DB**. JT is a join tree of Q', because the connectedness condition holds in the hypertree decomposition HD and thus in JT, by construction. Figure 9 shows the join tree JT_1 of the acyclic query Q'_1 corresponding to query Q_1 of our running example.

Each relation in $\mathbf{DB}'$ is the join of (at most) k relations of **DB**, and its size is $O(r^k)$, where r denotes the maximum relation-size over the relations in **DB**. Moreover, the number of relations in $\mathbf{DB}'$ is equal to the number of vertices of T. Therefore, $||\mathbf{DB}'|| = O(||\hat{HD}||\cdot r^k)$. Since $||JT|| = O(||\hat{HD}||)$ and $||Q'|| = O(||\hat{HD}||)$, we have $||\langle Q', \mathbf{DB}', JT\rangle|| = O((||Q||+||HD||)\cdot r^k)$ (recall that $||\hat{HD}|| = O(||Q||+||HD||)$, where the term $||Q||$ comes from the "completion" of the decomposition HD – see above).

The described transformation mainly involves a join of at most k relations for each node of the tree. Since the join of a constant number of relations can be computed in logspace, the transformation is feasible in logspace. ∎

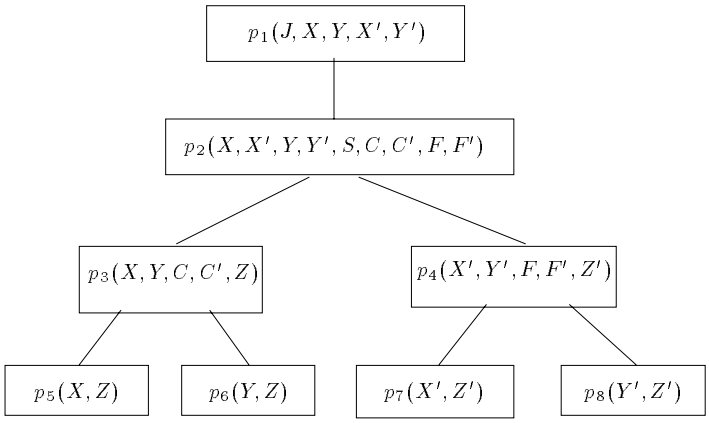

Figure 9: Join tree JT_1 computed for query Q'_1

Answering conjunctive queries having bounded hypertree width has the same complexity as answering acyclic queries, as stated by the following theorem.

Theorem 2 ([22]) *Given a database* **DB**, *a Boolean conjunctive query Q, and a k-width hypertree-decomposition of Q for a fixed constant $k > 0$, deciding whether Q evaluates to* true *on* **DB** *is* LOGCFL-*complete.*

Using the same construction as in the proof of Lemma 5 and well-known results on the sequential complexity of acyclic conjunctive-query evaluation [60], we get the following result for non-Boolean conjunctive queries.

Theorem 3 *Given a database* **DB**, *a (non-Boolean) conjunctive query Q, and a k-width hypertree decomposition of Q for a fixed constant $k > 0$, the answer of Q on* **DB** *can be* computed *in time polynomial in the combined size of the input instance and of the output relation.*

Remark. In this section we demonstrated that k-bounded hypertree-width queries are efficiently computable, once a k-width hypertree decomposition of the query is given as (additional) input. In particular, we have seen that the given decomposition in fact determines the query plan for executing the query. In the following section, we face the problem of computing such a decomposition.

4.3 Computing a Hypertree Decomposition

Given a query Q and a small-width hypertree decomposition HD for Q, we know that there is a polynomial time upper bound for answering Q, while in general this problem is NP-hard and all the available algorithms requires exponential time in the worst case. However, HD is not just a theoretical indication of tractability for Q. Rather, after the construction in Lemma 5, HD represents a (partially specified) query plan for Q. Indeed, the decomposition suggests to proceed in two phases: first compute the join operations among relations occurring together in some vertex of HD; then, compute just semi-joins among the intermediate relations computed in the previous phase, following some topological ordering of the vertices of HD, as in Yannakakis's algorithm. Note that such a query plan is not completely specified. For instance, no actual join method (merge, nested-loop, etc.) is chosen. However, this final step can be easily implemented using well-known techniques that take into account indices, size of relations, and so on. We remark that such optimizations are to be executed just on relations belonging to the same vertex, and hence on k relations at most, for some fixed and usually very small $k > 0$. Thus, also optimal methods based on dynamic programming or sophisticated heuristics can be employed, as the size of the problem is small, by the choice of k.

Given a query Q, we next focus on the problem of computing a hypertree decomposition of its hypergraph $H(Q)$ having the smallest possible width.

First, we introduce a new normal form for hypertree decompositions, which is a bit stronger than the normal form proposed in [22]. This stronger form allows us to develop a more efficient procedure for computing optimal hypertree decomposition.

Let $\mathcal{H}$ be a hypergraph, and let $V \subseteq var(\mathcal{H})$ be a set of variables and $X, Y \in var(\mathcal{H})$. X is $[V]$-adjacent to Y if there exists an edge $h \in edges(\mathcal{H})$ such that $\{X, Y\} \subseteq (h - V)$. A $[V]$-path π from X to Y is a sequence $X = X_0, \ldots, X_\ell = Y$ of variables such that: X_i is $[V]$-adjacent to X_{i+1}, for each $i \in [0...\ell\text{-}1]$. A set $W \subseteq var(\mathcal{H})$ of variables is $[V]$-connected if $\forall X, Y \in W$ there is a $[V]$-path from X to Y. A *[V]-component* is a maximal $[V]$-connected non-empty set of variables $W \subseteq (var(\mathcal{H}) - V)$. For any $[V]$-component C, let $edges(C) = \{h \in edges(\mathcal{H}) \mid h \cap C \neq \emptyset\}$.

Let $HD = \langle T, \chi, \lambda \rangle$ be a hypertree for a hypergraph $\mathcal{H}$. For any vertex v of T, we will often use v as a synonym of $\chi(v)$. In particular, $[v]\text{-}component$ denotes $[\chi(v)]\text{-}component$; the term $[v]$-path is a synonym of $[\chi(v)]$-path; and so on.

Definition 12 (Normal form) A hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ of a hypergraph $\mathcal{H}$ is in *normal form* (*NF*) if for each vertex $r \in vertices(T)$, and for each child s of r, all the following conditions hold:

1. there is (exactly) one $[r]\text{-}component$ C_r such that $\chi(T_s) = C_r \cup (\chi(s) \cap \chi(r))$;
2. $\chi(s) \cap C_r \neq \emptyset$, where C_r is the $[r]$-component satisfying Condition 1;
3. for each $h \in \lambda(s)$, $h \cap var(edges(C_r)) \neq \emptyset$, where C_r is the $[r]$-component satisfying Condition 1;
4. $\chi(s) = var(edges(C_r)) \cap var(\lambda(s))$, where C_r is the $[r]$-component satisfying Condition 1.

Note that, from Condition 4 above, the label $\chi(s)$ of any vertex s of a hypertree decomposition in normal form can be computed just from its label $\lambda(s)$ and from the $[r]$-component satisfying Condition 1 associated to its parent r.

Theorem 4 *For each k-width hypertree decomposition of a hypergraph $\mathcal{H}$ there exists a k-width hypertree decomposition of $\mathcal{H}$ in normal form.*

The normal form theorem above immediately entails that for each optimal hypertree decomposition of a hypergraph $\mathcal{H}$ there exists an optimal hypertree decomposition of $\mathcal{H}$ in normal form.

Figure 10 shows the Algorithm `opt-`k`-decomp` that, given a hypergraph $\mathcal{H}$, computes an optimal hypertree decomposition of $\mathcal{H}$ in normal form, whose width is bounded by some fixed $k > 0$ (if any). The algorithm returns "failure" if no such a decomposition exists (i.e, if $hw(\mathcal{H}) > k$). We will call k-vertex any set of edges of $\mathcal{H}$ having cardinality at most k. The Algorithm `opt-`k`-decomp` is composed of the two procedures *Compute-CG* and *Weight-CG*, called one after the other.

The procedure *Compute-CG* computes a directed weighted graph CG whose nodes are partitioned in two sets N_e and N_a. Each node in N_e is a pair (R, C) where R is a k-vertex of $\mathcal{H}$, and C is an $[R]$-$component$. The node $(root, V)$ is special, because V is the set of all vertices of the hypergraph $\mathcal{H}$ and $root$ does not correspond to any k-vertex of $\mathcal{H}$. It will be the root vertex of any hypertree decomposition computed by `opt-`k`-decomp`. This node has no outgoing arcs.

Intuitively, for every node $(R, C) \in N_e$, we solve the subproblem associated to the $[R]$-$component$ C. Then, from the partial solutions computed for these subproblems, we are able to determine an optimal hypertree decomposition of the whole hypergraph. Each node $(R, C) \in N_e$ has some incoming arcs from nodes in N_a. Let $(S, C) \in N_a$ be any of these nodes, where S is a k-vertex. Then S represents a candidate for "breaking" the $[R]$-$component$ C of the hypergraph. The evaluation of this choice requires the solution of smaller subproblems, one for each $[S]$-$component$ C' of $\mathcal{H}$ such that $C' \subseteq C$. For this reason, (S, C) has an incoming arc from any such a node $(S, C') \in N_e$.

Node are weighted by the bottom-up procedure *Weight-CG*. Nodes having no incoming arcs are the starting point of the procedure and are weighted as follows: for each node $(S, C) \in N_a$ with no incoming arcs, set *weight*$((S, C)) = |S|$, i.e., its weight is the number of hyperedges in S; for each node $(R, C) \in N_e$ with no incoming arcs, set *weight*$((R, C)) = \infty$, because in this case there is no k-vertex S that can be used to decompose the $[R]$-component C. Then, for any vertex (R, C) in N_e such that all of its incoming nodes have been weighted, set *weight*$((R, C))$ = *weight*$((S, C))$, where (S, C) is its best-weighted incoming node. For any vertex (S, C) in N_a such that all of its incoming nodes have been weighted, set *weight*$((S, C))$ = *weight*$((R, C))$, where (R, C) is its worst-weighted incoming node. Intuitively, the best weighted incoming node for a node $(R, C) \in N_e$ represents the k vertex chosen for decomposing the $[R]$-component C. Any node $(S, C) \in N_a$ behaves differently, because there is no choice to be done. Indeed, every $[S]$-$component$ $C' \subseteq C$ must be decomposed, hence the weight of (S, C) will be determined by the worst subproblem, i.e., by the worst-weighted incoming node.

Note that, in particular, the weight of the special node $(root, V)$ will be the hypertree width of $\mathcal{H}$ if $hw(\mathcal{H}) \leq k$ and ∞ otherwise.

After the execution of the procedure *Weight-CG*, the weighted graph CG contains enough information to compute every optimal hypertree decomposition of $\mathcal{H}$ in normal form, if $hw(\mathcal{H}) \leq k$. Figure 11 shows the procedure *Compute-hypertree*, which selects one optimal hypertree decomposition of $\mathcal{H}$ in normal form. It proceeds top-down, starting from the special node $(root, V)$ and recursively descending along the graph, choosing at each step the best-weighted nodes. Indeed, these nodes corresponds to the best solutions of the subproblems associated to the nodes of CG.

`opt-`k`-decomp` runs in $O(n^{2k}m^2)$ time, where n and m are the number of edges and the number of vertices of $\mathcal{H}$, respectively. This bound can be easily obtained by inspecting the procedures of `opt-`k`-decomp`. Note that there are $O(n^k)$ k-vertices of $\mathcal{H}$, and for each k-vertex R, $O(m)$ $[R]$-components. It follows that the graph CG has $O(n^k m)$ nodes in N_e and $O(n^{2k}m)$ nodes in N_a. Moreover, each node in N_e has $O(n^k)$ incoming arcs at most, and each node in N_a has $O(m)$ incoming arcs at most.

4.4 Cost-based Query Decompositions

We have described an algorithm that computes an optimal hypertree decomposition HD for a given query Q, and we observed that such a decomposition determines a partial query plan for Q, which can be completed using well-known techniques. Denote by $QP(HD)$ the completely specified query plan

ALGORITHM opt-k-decomp
Input: A Hypergraph $\mathcal{H} = (V, H)$.
Output: An optimal hypertree decomposition
of $\mathcal{H}$, if $hw(\mathcal{H}) \leq k$; $failure$, otherwise.

Var
$CG = (N_e \cup N_a, A, \textit{weight})$: weighted directed graph;
$HD = \langle T, \chi, \lambda \rangle$: hypertree of $\mathcal{H}$;

Procedure *Compute-CG*
N_e:= $\{(root, V)\} \cup \{(R, C)$: R is a k-vertex
and C is an $[R]$-component $\}$;
$N_a := \emptyset$; $A := \emptyset$;
For each $(R, C) \in N_e$ **Do**
Let $ec := \bigcup_{h \in edges(C)} var(h)$;
Let $rc := ec \cap var(R)$;
For each k-vertex S **Do**
If $var(S) \cap C \neq \emptyset$ And $rc \subseteq var(S)$ And $(\forall h \in S, h \cap ec \neq \emptyset)$ **Then**
$N_a := N_a \cup \{ (S, C) \}$;
Add an arc from (S, C) to (R, C) in A;
For each $(S, C') \in N_e$ s.t. $C' \subset C$ **Do**
Add an arc from (S, C') to (S, C) in A;
EndIf
endFor
endProcedure;

Procedure *Weight-CG*
weight$((R, C)) := \infty$, for any $(R, C) \in N_e$
having no incoming arcs in CG;
weight$((S, C)) := |S|$, for any $(S, C) \in N_a$
having no incoming arcs in CG;
(* For any $p \in N_e \cup N_a$, let $in(p)$ denote the number of
arcs coming into p from unweighted nodes *)
While there is some unweighted node in $N_e \cup N_a$ **Do**
Let $p = (S, C)$ be an unweighted node s.t. $in(p) = 0$;
If $p \in N_e$ **Then**
weight$(p) = min(\{$*weight*$(q) | (q, p) \in A\})$;
Else (* $p \in N_a$ *)
weight$(p) = max(\{|S|\} \cup \{$*weight*$(q) | (q, p) \in A\})$;
EndWhile
endProcedure;

begin*(* MAIN *)*
Compute-CG;
Weight-CG;
If *weight*$((root, V)) = \infty$ **Then**
Output *failure*;
Else
Create a vertex $root'$ in T;
$\chi(root') := \emptyset$; $\lambda(root') := \emptyset$;
Compute-hypertree$((root, V), root')$;
Output HD
end.

Figure 10: Computing an optimal hypertree-decomposition

Procedure *Compute-hypertree*($p : CG$node; $r : HD$vertex)
  Choose a minimum-weighted predecessor (S, C) of p;
  Create a new vertex s as a child of r in T;
  $\lambda(s) := S$;
  $\chi(s) := var(S) \cap (C \cup \chi(r))$;
  For each predecessor q of (S, C) **Do**
    Compute-hypertree(q, s);
endProcedure;

Figure 11: Procedure *Compute-hypertree*

for Q computed from the decomposition HD, i.e., the relational expression specifying how to compute $Q(\mathbf{DB})$.[5] Note that the plan obtained by using Algorithm `opt-`k`-decomp` works well if we have no accurate information on the relations involved in the query. However, if we are given sizes of relations, attributes' selectivity, etc., we can do much better. Indeed, in this case, we can guide the search for hypertree decompositions towards those decompositions that guarantee the lowest (estimated) execution cost for Q.

Let $k > 0$ be a fixed bound on the width of hypertree decompositions we are interested in. Let **DB** be a database and assume we are given quantitative information on the data in **DB**. Let *cost* be a function that, given a relational expression (or a plan) E on **DB**, computes an estimation of executing E on **DB** according to the given quantitative information. We next present a new algorithm `cost-`k`-decomp` that, given a Boolean query Q and the information on **DB**, returns *failure* if $hw(\mathcal{H}(Q)) > k$, otherwise computes a *cost-based optimal hypertree decomposition* of $\mathcal{H}$, i.e., a decomposition HD of with at most k such that $cost(QP(HD))$ is the minimum over all k-bounded hypertree decompositions in normal form. This algorithm can be applied also to non-Boolean queries Q whose set of output (head) variables is included in some atom belonging to the body of Q, with a slight modification of the algorithm. Indeed, in this case the choice of the root of the hypertree in no longer arbitrary, but depends on such output variables. Furthermore, it can be easily extended to general non-Boolean queries. We deal with these issues in [41].

Note that for answering queries we need complete decompositions, where each atom occurs at least once. However, our algorithms are designed to compute decompositions in normal form that, in general, are not complete. Even if a complete decomposition can be easily and efficiently computed from any hypertree decomposition, this is not sufficient in the cost-based approach. Indeed, it may happen that the computed optimal decomposition is no longer optimal after we make it complete. To overcome this problem, we may assume without loss of generality that every query Q satisfies the following condition: each atom p of Q has a variable X_p which does not occur in any other atom of Q. We call *unambiguous* the queries satisfying this property. Clearly, if a query is not unambiguous, it can be immediately transformed into a query in this form by adding a fresh variable to each atom. Typically, quantitative information on the database is not affected by this modification (or it can be properly extended, in a straightforward way). Once a query plan has been computed for the transformed query, it can be cleaned from such artificial fresh variables, thus obtaining a query plan for the original query. It is easy to verify that all hypertree decompositions of unambiguous queries are complete, from Condition 1 in Definition 11. Moreover, for these queries, there is a one-to-one correspondence between atoms of Q and hyperedges of $\mathcal{H}(Q)$. Therefore, if p is an atom of an unambiguous query Q on a database **DB**, we will refer to the corresponding edge in $\mathcal{H}(Q)$ using the same name p and viceversa, as p is uniquely identified by its variables. Recall that $rel(p)$ denotes the relation in **DB** that corresponds to the atom (edge) p.

Figure 12 shows Algorithm `cost-`k`-decomp`. This algorithm is similar to `opt-`k`-decomp`, but for the procedure *Weight-CG* that assigns weights to the nodes of the graph CG. Roughly, in `opt-`k`-decomp` the weight of a node p is the maximum width of any node in the subtree rooted at

[5] More generally, plans also specify the algorithms to be used for each operator's occurrence, and other low-level details that we skip here for the sake of presentation. However, they can be easily included in a full implementation of our techniques.

p, while in `cost-k-decomp` it is a measure of the cost of evaluating the partial plan represented by the subtree rooted at p, by using the given cost model.

Let $HD = \langle T, \chi, \lambda \rangle$ be a hypertree decomposition in normal form for a query Q on a database **DB**. For any vertex p of T, let $E(p)$ denote the relational expression $E(p) = \bowtie_{h \in \lambda(p)} \prod_{\chi(p)} rel(h)$, i.e., the join of all relations corresponding to hyperedges in $\lambda(p)$, suitably projected onto the variables in $\chi(p)$. Again, nodes having no incoming arcs are the starting point of the bottom-up procedure *Weight-CG*, and are weighted as follows: for each node $p \in N_a$ with no incoming arcs, set *weight*$(p) = cost(E(p))$, i.e., its weight is the estimate of the cost of computing the expression $E(p)$; for each node $(R, C) \in N_e$ with no incoming arcs, set *weight*$((R, C)) = \infty$, because in this case there is no k-vertex S that can be used to decompose the $[R]$-component C. For any vertex $p = (S, C)$ in N_a such that all of its incoming nodes have been weighted, set

$$\textit{weight}(p) = cost(E(p)) + \sum_{q \in inc(p)} \min_{p' \in inc(q)} \{cost(E(p) \ltimes E(p')) + \textit{weight}(p')\},$$

where $inc(p)$ denote the set of the source nodes of all the arcs coming into p, i.e., $inc(p) = \{q \mid (q, p) \in A\}$, *weight*$(p')$ holds the estimate of evaluating the subtree rooted at the descendant p', $cost(E(p))$ is the cost of evaluating the relational expression for node p, and, for each $q \in inc(p)$, $cost(E(p) \ltimes E(p'))$ is the cost of evaluating the semi-join of the already computed relational expression $E(p')$ of the "best" child p' of q with $E(p)$. Recall that all predecessors $q = (S, C')$ in $inc(p)$ correspond to components $C' \subset C$ to be decomposed. Intuitively, the above expression says that, for each of these nodes, we add to the estimate for p the weight determined by the predecessor $p' = (R, C')$ of q that guarantees the minimum evaluation-cost (w.r.t. p).

The following example shows how we can employ Algorithm `cost-k-decomp` for computing a query plan of a conjunctive query, given data information as the size of relations and the selectivity of attributes, and a cost model for estimating the cost of evaluating relational expressions. This is a crucial issue in query optimizers, and the literature reports many techniques for estimating such costs under different assumptions on the data. See, e.g., [17], for algorithms and further references. In our example, we employ a simple cost model where costs are measured in terms of the size of the relations computed for answering the query. However, note that Algorithm `cost-k-decomp` is general enough to be used with more sophisticated cost models, taking into account indices, number of disk accesses, and so on.

Example 7 Consider again query Q_1 in Example 6, and assume we have to evaluate this query on the randomly generated database **DB**, whose quantitative information on the data are reported in Tables 2 and 3. In particular, Table 2 shows, for each atom p occurring in Q_1, the number of tuples in the relation $rel(p)$ associated to p; Table 3 shows, for each variable X occurring in any atom p in Q_1, the selectivity of the attribute corresponding to X in $rel(p)$, i.e., the number of *distinct* values X takes on $rel(p)$ or, equivalently, the cardinality of $\prod_X rel(p)$.

To use Algorithm `cost-k-decomp` we have to choose the function *cost*, i.e., to choose a cost model for estimating the cost of evaluating each relational expression. In this example, costs are measured in terms of the size of the relations computed. In particular, for any vertex p in CG, we have to estimate the cost of the relational expression $E(p)$, which is a join operation involving at most k relations, suitably projected onto a subset of their attributes. Of course, there are different ways to perform these join operations. In our preliminary testing of `cost-k-decomp`, we estimate the cost of the k-way join associated to each vertex assuming that the *join selectivity criterion* described in [17] is used to perform this operation. However, since k is typically very small, also exhaustive techniques are viable for choosing the best plan for $E(p)$. A similar selectivity criterion has been used for choosing the best ordering for the semi-join operations associated to the arcs coming into a node p of CG. See [41] for all details, including the estimates we used for every kind of relational operation (join, semi-join, projection, etc.).

Once we have fixed the cost model and a bound $k > 0$, we use `cost-k-decomp` for computing a cost-based hypertree decomposition HD whose associated query plan $QP(HD)$ for answering Q_1 on

ALGORITHM `cost-`k`-decomp`
Input: A query Q on a database **DB**, quantitative information on Q and **DB**,
and the hypergraph $\mathcal{H} = (V, H)$ associated to Q.
Output: A cost-based optimal hypertree decomposition
of $\mathcal{H}$, if $hw(\mathcal{H}) \leq k$; $failure$, otherwise.

Var
$CG = (N_e \cup N_a, A, \textit{weight})$: weighted directed graph;
$HD = \langle T, \chi, \lambda \rangle$: hypertree of $\mathcal{H}$;

Procedure *Compute-CG*
N_e:= $\{(root, V)\} \cup \{(R, C) \ : \ R$ is a k-vertex and C is an $[R]$-component $\}$;
$N_a := \emptyset$; $A := \emptyset$;
For each $(R, C) \in N_e$ **Do**
Let $ec := \bigcup_{h \in edges(C)} var(h)$;
Let $rc := ec \cap var(R)$;
For each k-vertex S **Do**
If $var(S) \cap C \neq \emptyset$ And $rc \subseteq var(S)$ And $(\forall h \in S, h \cap ec \neq \emptyset)$ **Then**
$N_a := N_a \cup \{ \ (S, C) \ \}$;
Add an arc from (S, C) to (R, C) in A;
For each $(S, C') \in N_e$ s.t. $C' \subset C$ **Do**
Add an arc from (S, C') to (S, C) in A;
EndIf
endFor
endProcedure;

Procedure *Weight-CG*
weight$((R, C)) := \infty$, for any $(R, C) \in N_e$ having no incoming arcs in CG;
weight$((S, C)) := cost(E(p))$, for any $(S, C) \in N_a$ having no incoming arcs in CG;
(* For any $p \in N_e \cup N_a$, let $inc(p)$ denote the set of the source nodes of all the arcs coming into p,
i.e., $inc(p) = \{q \mid (q, p) \in A\}$ *)
While there is some unweighted node in $N_e \cup N_a$ **Do**
Let $p = (S, C)$ be an unweighted node s.t. all nodes in $inc(p)$ have been weighted;
If $p \in N_e$ **Then**
weight$(p) = min_{q \in inc(p)}\{$*weight*$(q)\}$;
Else (* $p \in N_a$ *)
If $\exists q \in inc(p)$ s.t. *weight*$(q) = \infty$ **Then**
weight$(p) := \infty$;
Else
weight$(p) = cost(E(p)) + \sum_{q \in inc(p)} min_{p' \in inc(q)}\{cost(E(p) \ltimes E(p')) +$ *weight*$(p')\}$;
EndWhile
endProcedure;

begin*(* MAIN *)*
Compute-CG;
Weight-CG;
If *weight*$((root, V)) = \infty$ **Then**
Output *failure*;
Else
Create a vertex $root'$ in T;
$\chi(root') := \emptyset$; $\lambda(root') := \emptyset$;
Compute-hypertree$((root, V), root')$;
Output HD
end.

Figure 12: Computing a cost-based optimal hypertree-decomposition

ATOM	CARDINALITY
$a(S, X, X', C, F)$	4606
$b(S, Y, Y', C', F')$	2808
$c(C, C', Z)$	1748
$d(X, Z)$	3756
$e(Y, Z)$	3554
$f(F, F', Z')$	2892
$g(X', Z')$	4573
$h(Y', Z')$	3390
$j(J, X, Y, X', Y')$	4234

Table 2: Cardinality of the relations in Example 7

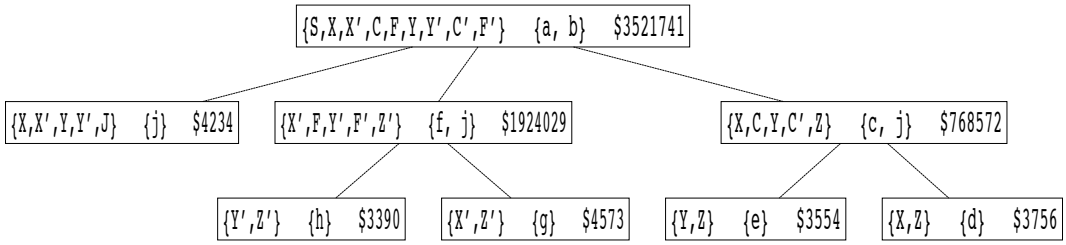

Figure 13: A cost-based optimal hypertree-decomposition of width 2 for Q_1

DB guarantees the minimum estimated cost over all (normal form) k-bounded hypertree decomposition of Q_1.

Figure 13 shows an optimal cost-based hypertree decomposition of Q_1 (w.r.t. **DB**) if we fix the bound k to 2. In the figure, each vertex v has a new label marked by the symbol **$** that represents the estimated cost for evaluating the subtree rooted at v. In particular, for each leaf ℓ, this number will be equal to $cost(E(\ell))$; for the root r of the hypertree, this number gives the estimated cost of the whole evaluation of Q_1(**DB**). In our example, this cost is 3521741.

Recall that Q_1 is a cyclic query, having hypertree width 2, thus there is no hypertree decomposition of width 1 for Q_1, and $k = 2$ is the lowest bound such that `cost-k-decomp` is able to compute decomposition for Q_1. However, any value larger than 2 is feasible, and so we have run `cost-k-decomp` with k ranging from 2 to 5. Figures 14 and 15 show two optimal cost-based decompositions for $k = 3$ and $k = 4$, whose estimated costs are 1373879 and 854867, respectively. For $k = 5$ we get the same value as for $k = 4$.

It turns out that, even if the hypertree width of Q_1 is 2, for the given quantitative information on **DB** and the cost model we have chosen, the bounds 4 and 5 lead to the best query plans. This is not surprising, as a larger bound k allows us to explore more decompositions, containing larger vertices with more associated join operations. Note that these vertices are not optimal from a purely structural point of view, and are in fact discarded by Algorithm `opt-k-decomp`, which just looks for low-width decompositions.

5 Conclusions and Open Issues

In this paper we have presented an original approach to view materialization in which NGPSJ expressions are used to model both queries and candidate views. An algorithm for selecting the minimal set of candidate views by incrementally inserting queries into a directed acyclic graph has been proposed; the algorithm is efficient since, every time a query is inserted, only a portion of the graph is visited.

We have also shown how cardinality constraints derived from the application domain may be employed to determine effective bounds on the cardinality of aggregate views. In order to devise a compre-

ATOM	VARIABLE	SELECTIVITY
$a(S, X, X', C, F)$	S	14
$a(S, X, X', C, F)$	X	24
$a(S, X, X', C, F)$	X'	16
$a(S, X, X', C, F)$	C	21
$a(S, X, X', C, F)$	F	15
$b(S, Y, Y', C', F')$	S	17
$b(S, Y, Y', C', F')$	Y	5
$b(S, Y, Y', C', F')$	Y'	12
$b(S, Y, Y', C', F')$	C'	20
$b(S, Y, Y', C', F')$	F'	7
$c(C, C', Z)$	C	18
$c(C, C', Z)$	C'	7
$c(C, C', Z)$	Z	19
$d(X, Z)$	X	18
$d(X, Z)$	Z	7
$e(Y, Z)$	Y	21
$e(Y, Z)$	Z	13
$f(F, F', Z')$	F	20
$f(F, F', Z')$	F'	7
$f(F, F', Z')$	Z'	6
$g(X', Z')$	X'	22
$g(X', Z')$	Z'	16
$h(Y', Z')$	Y'	15
$h(Y', Z')$	Z'	12
$j(J, X, Y, X', Y')$	J	18
$j(J, X, Y, X', Y')$	X	8
$j(J, X, Y, X', Y')$	Y	18
$j(J, X, Y, X', Y')$	X'	22
$j(J, X, Y, X', Y')$	Y'	10

Table 3: Selectivity of the attributes for the database **DB** in Example 7

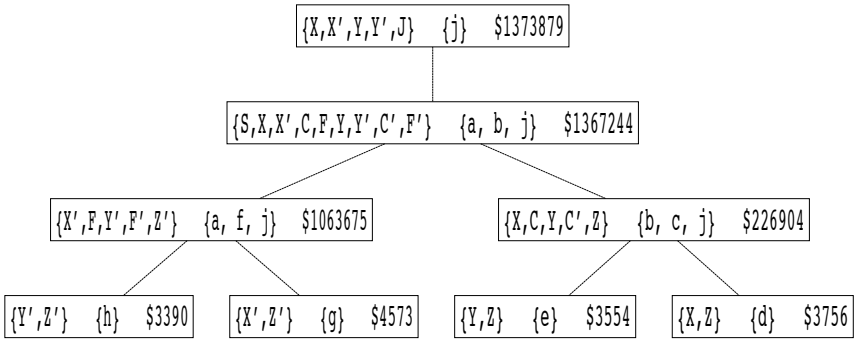

Figure 14: A cost-based optimal hypertree-decomposition of width 3 for Q_1

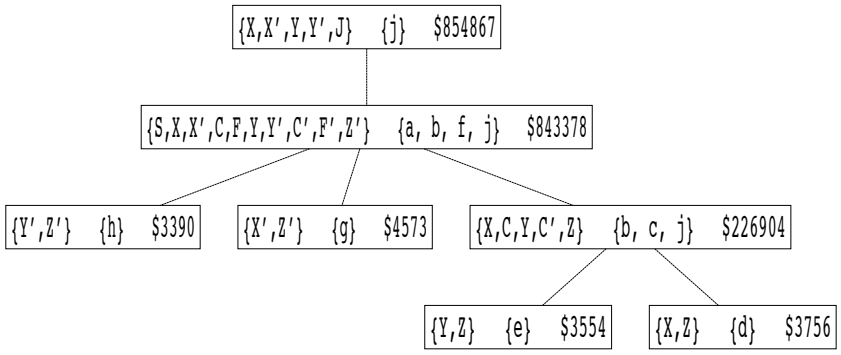

Figure 15: A cost-based optimal hypertree-decomposition of width 4 for Q_1

hensive approach, several issues still need to be investigated. In particular, it is necessary to develop a probabilistic model which, given a complex NGPSJ view, estimates its cardinality by properly taking its selection predicates into account.

Finally, we faced the problem of populating and refreshing data warehouses. This operation typically requires a number of complex and long queries. We presented a technique for answering efficiently such queries, choosing query plans that guarantee low execution costs. This technique combines structural and quantitative methods and is based on the notion of hypertree decomposition. We are currently developing efficient implementations of the algorithms we presented in this paper, and we are tuning these algorithms in order to deal with more general queries.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling Multidimensional Databases. In *Proc. ICDE*, pages 232–243, Birmingham, UK, 1997.

[3] M. Akinde and M. Böhlen. Constructing GPSJ view graphs. In *Proc. DMDW*, Heidelberg, Germany, 1999.

[4] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in multidimensional databases. In *Proc. 23rd VLDB*, pages 156–165, Athens, Greece, 1997.

[5] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, July 1983.

[6] P.A. Bernstein and N. Goodman. The power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.

[7] L. Cabibbo and R. Torlone. A framework for the investigation of aggregate functions in database queries. In *Proc. ICDT*, Jerusalem, Israel, 1999.

[8] Ch. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211–229, 2000.

[9] P. Ciaccia and D. Maio. On the Complexity of Finding Bounds for Projection Cardinalities in Relational Databases. *Information Systems*, 17(6):511–515, 1992.

[10] P. Ciaccia and D. Maio. Domains and Active Domains: What This Distinction Implies for the Estimation of Projection Sizes in Relational Databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):641–655, 1995.

[11] S. Cohen, W. Nutt, and A. Serebrenik. Algorithms for rewriting aggregate queries using views. In *Proc. DMDW*, Heidelberg, Germany, 1999.

[12] A. D'Atri and M. Moscarini. Recognition algorithms and design methodologies for acyclic database schemes. In P. Kanellakis, editor, *Advances in Computing Research*, pages 43–68. JAI press, 1986.

[13] Rina Dechter. Constraint networks. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 276–285. Wiley, 1992. Volume 1, second edition.

[14] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM*, 30(3):514–550, 1983.

[15] R. Fagin, A.O. Mendelzon, and J.D. Ullman. A simplified universal relation assumption and its properties. *ACM Transactions on Database Systems*, 7(3):343–360, 1982.

[16] Gottlob G., Leone N., and Scarcello F. A comparison of structural csp decomposition methods. *Artificial Intelligence*, 124(2), 2000.

[17] H. Garcia-Molina, J. Ullman, and J. Widom. *Database system implementation*. Prentice Hall, 2000.

[18] M. Golfarelli, D. Maio, and S. Rizzi. Applying vertical fragmentation techniques in logical design of multidimensional databases. In *Proc. DaWaK*, pages 11–23, 2000.

[19] M. Golfarelli and S. Rizzi. Comparing Nested GPSJ Queries in Multidimensional Databases. In *Proc. DOLAP*, pages 65–71, Washington, DC, 2000.

[20] N. Goodman and O. Shmueli. Tree queries: a simple class of relational queries. *ACM Transactions on Database Systems*, 7(4):653–6773, 1982.

[21] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science(FOCS-98), Los Alamitos, CA*, pages 706–715, November 1998.

[22] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems – PODS'99*, pages 21–32, May 31st – June 2nd 1999. To appear in *Journal of Computer and System Sciences*.

[23] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Advanced parallel algorithms for processing acyclic conjunctive queries, rules, and constraints. In *Proceedings of the 2000 Conference on Software Engineering and Knowledge Engineering (SEKE'00), Chicago*, pages 167–176, July 2000.

[24] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM*, 48(3):431–498, 2001.

[25] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: A survey. In *In Proceedings of MFCS'2001*, pages 37–57, 2001.

[26] J. Gray, A. Bosworth, A. Lyman, and H. Pirahesh. Data-Cube: a relational aggregation operator generalizing group-by, cross-tab and sub-totals. Technical Report MSR-TR-95-22, Microsoft Research, 1995.

[27] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data-warehousing environments. In *Proc. 21st VLDB*, Zurich, Switzerland, 1995.

[28] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *Proc. ICDT*, pages 98–112, Delphi, Greece, 1997.

[29] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *Proc. ICDE*, pages 208–219, Birmingham, UK, 1997.

[30] H. Gupta and I.S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *Proc. ICDT*, Jerusalem, Israel, 1999.

[31] M. Gyssens, P.G. Jeavons, and D.A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Journal of Algorithms*, 66:57–89, 1994.

[32] M. Gyssens and L.V .S. Lakshmanan. A Foundation for Multi-Dimensional Databases. In *Proc. 23rd VLDB*, pages 106–115, Athens, Greece, 1997.

[33] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. In *Proc. ACM Sigmod Conf.*, pages 205–216, Montreal, Canada, 1996.

[34] W. Hou and G. Özsoyoglu. Statistical Estimators for Aggregate Relational Algebra Queries. *ACM Transactions on Database Systems*, 16(4):600–654, 1991.

[35] R. Kimball. *The data warehouse toolkit*. John Wiley & Sons, 1996.

[36] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. In *Proceedings of Symposium on Principles of Database Systems (PODS'98), Seattle, Washington*, pages 205–213, 1998.

[37] H.J. Lenz and A. Shoshani. Summarizability in OLAP and statistical databases. In *Proc. Statistical and Scientific Database Management*, Washington, 1997.

[38] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1986.

[39] F.M. Malvestuto. Modelling large bases of categorical data with acyclic schemes. In *Proceedings of the First International Conference on Database Theory, Rome, Italy,*, 1986.

[40] M. V. Mannino, P. Chu, and T. Sager. Statistical Profile Estimation in Database Systems. *ACM Computing Surveys*, 20(3):191–221, 1988.

[41] A. Mazzitelli. Progettazione e sviluppo di tecniche di decomposizione per la valutazione efficiente di interrogazioni congiuntive di basi di dati. Master's thesis, Universitá della Calabria, 2002.

[42] M. Muralikrishna and D.J. DeWitt. Equi-depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. In *Proc. ACM Sigmod Conf.*, pages 28–36, Chicago, IL, 1988.

[43] C.H. Papadimitriou and M. Yannakakis. On the complexity of database queries. In *Proceedings of Symposium on Principles of Database Systems (PODS'97), Tucson, Arizona*, pages 12–19, 1997.

[44] N. Robertson and P.D. Seymour. Graph minors ii. algorithmic aspects of tree width. *Journal of Algorithms*, 7:309–322, 1986.

[45] K. Ross and D. Srivastava. Fast Computation of Sparse Datacubes. In *Proc. 23rd VLDB*, pages 116–125, Athens, Greece, 1997.

[46] K. A. Ross, D. Srivastava, and D. Chatziantoniou. Complex aggregation at multiple granularities. In *Proc. ICDE*, 1998.

[47] K. A. Ross, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Foundations of aggregation constraints. *Theoretical Computer Science*, 193(1-2):149–179, 1998.

[48] D. Saccà. Closures of database hypergraphs. *Journal of the ACM*, 32(4):774–803, 1985.

[49] Y. Sagiv and O. Shmueli. Solving queries by tree projections. *ACM Transactions on Database Systems*, 18(3):487–511, 1993.

[50] Francesco Scarcello. Answering queries: Tractable cases and optimizations, d2r3-cofin2000. Technical report, DEIS, 2001.

[51] A. Shoshani. OLAP and statistical databases: similarities and differences. In *Proc. CIKM*, Rockville, Maryland, 1996.

[52] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. In *Proc. 22nd VLDB*, pages 522–531, Mumbai, India, 1996.

[53] D. Theodoratos and M. Bouzeghoub. A General Framework for the View Selection Problem for Data Warehouse Design and Evolution. In *Proc. DOLAP*, pages 1–8, Washington, DC, 2000.

[54] D. Theodoratos and T. Sellis. Data warehouse configuration. In *Proc. 23rd VLDB*, pages 126–135, Athens, Greece, 1997.

[55] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.

[56] M. Vardi. Complexity of relational query languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing, (STOC '82)*, pages 137–146, 1982.

[57] P. Vassiliadis. Gulliver in the land of data warehousing: practical experiences and observations of a researcher. In *Proc. DMDW*, pages 12/1–12/16, Stockholm, Sweden, 2000.

[58] A.N. Wilschut, J. Flokstra, and P. M.G. Apers. Parallel evaluation of multi-join queries. In *Proceedings of SIGMOD'95, CA, USA*, pages 21–32, 1995.

[59] W.P. Yan and P. Larson. Eager and lazy aggregation. In *Proc. 21st VLDB*, pages 345–357, Zurich, Switzerland, 1995.

[60] M. Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the 7th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Zaniolo and Delobel(eds)*, September 1981.

[61] C.T. Yu and M.Z. Özsoyoğlu. On determining tree-query membership of a distributed query. *Infor*, 22(3):261–282, 1984.