What can Hierarchies do for Data Warehouses?

H. V. Jagadish[*]
U of Michigan, Ann Arbor
jag@eecs.umich.edu

Laks V. S. Lakshmanan[†]
IIT, Bombay
laks@cse.iitb.ernet.in

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

Data in a warehouse typically has multiple dimensions of interest, such as location, time, and product. It is well-recognized that these dimensions have hierarchies defined on them, such as "store-city-state-region" for location. The standard way to model such data is with a star/snowflake schema. However, current approaches do not give a first-class status to dimensions. Consequently, a substantial class of interesting queries involving dimension hierarchies and their interaction with the fact tables are quite verbose to write, hard to read, and difficult to optimize.

We propose the SQL($\mathcal{H}$) model and a natural extension to the SQL query language, that gives a first-class status to dimensions, and we pin down its semantics. Our model permits structural and schematic heterogeneity in dimension hierarchies, situations often arising in practice that cannot be modeled satisfactorily using the star/snowflake approach. We show using examples that sophisticated queries involving dimension hierarchies and their interplay with aggregation can be expressed concisely in SQL($\mathcal{H}$). By comparison, expressing such queries in SQL would involve a union of numerous complex sequences of joins. Finally, we develop an efficient implementation strategy for computing SQL queries, based on an algorithm for hierarchical joins, and the use of dimension indexes.

[*] Supported in part by NSF under grant IDM9877060.

[†] Currently on leave from Concordia University, Canada.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

Two key aspects of OLAP queries are aggregation and dimension hierarchies. Aggregations can involve multiple `GROUP BY`s as in `CUBE`, `ROLLUP`, and `DRILLDOWN` [8], or multiple levels of granularity [5, 20]. Several algorithms have been proposed for the efficient implementation of these queries (see, e.g., [8, 5, 9, 1, 23, 19, 20]). In contrast, work on dimension hierarchies has been sparse (see Section 1.2 for details). Dimension hierarchies arise naturally and are central to a large class of useful OLAP queries. In fact, `ROLLUP` and `DRILLDOWN` queries make sense only if there are dimension hierarchies with more than one level. This paper is a study of hierarchy in a data warehouse.

1.1 Contributions

We make the following contributions in this paper.

- We identify some key weaknesses (pertaining to modeling and query language) of the standard star/snowflake schemas, for common OLAP applications (Section 2).
- We propose a model for data warehouses, called the SQL($\mathcal{H}$) data model, which naturally extends the relational data model of SQL, by giving a first-class status to the notion of dimension hierarchies (Section 3).
- We present a simple but powerful extension to SQL, called the SQL($\mathcal{H}$) query language, which allows a user to regard dimension hierarchies as fundamental objects in themselves and express many useful OLAP queries concisely (Section 4).

  We show that a direct approach to expressing many SQL($\mathcal{H}$) queries in SQL will lead to an explosion in the size of the SQL query. This situation is analogous to the `CUBE` operator: even though `CUBE` is expressible in SQL, it is far less concise and harder to optimize.
- Finally, we develop algorithms that enable the direct computation of SQL($\mathcal{H}$) queries, and show that they lead to a significant speedup compared with the evaluation of the equivalent SQL queries (Section 5).

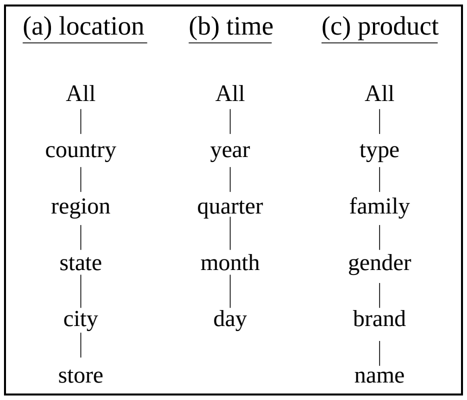


Figure 1: Dimension Hierarchies

1.2 Related Work

There has been a substantial amount of work on the general topic of data warehousing and OLAP (see, e.g., [22, 6]). For the sake of relevance and brevity, we discuss only those works that have addressed the issue of modeling and querying dimension hierarchies here.

Lehner [14] studies modeling issues in large scale OLAP applications and makes a case for heterogeneity arising within and across levels.

The well-known `CUBE` operator was extended by Baralis et al. [2] to deal with multiple aggregations on a data warehouse with hierarchical dimensions. The main issue they addressed was handling the redundancies among `GROUP BY`s arising from the presence of functional dependencies in dimensions. For example, if `zip` $\rightarrow$ `state` holds in dimension `location`, then any aggregation grouped by `zip, state` should yield the same result as one grouped by `zip` alone. This forms the basis for an optimization for computing the so-called hierarchical `CUBE` that they introduce.

Neither of the above two works addresses the technical issues in modeling or querying dimension hierarchies, two novel aspects of our contributions.

On the theoretical side, Cabibbo and Torlone [3] propose a model for multi-dimensional databases. Hurtado et al. [10] build on that model, and address the issue of maintaining materialized views on a data warehouse against updates to dimensions; this issue is completely orthogonal to the objective of this paper. Their models give a first-class status to dimensions, but only insofar as the model is concerned. Their assumption of explicit availability of rollup functions between successive levels of a dimension hierarchy suggests that query evaluation in their model is more likely to resemble that in the snowflake schema model.

2 Motivation

A typical data warehouse consists of *dimensions*, such as `location`, `time` and `product`, and *measures*, such as `dollarAmt` and `quantitySold`, being gauged as a function of the dimensions. The standard way to model and store such data is by means of the *star schema* [12, 6]. In the star schema model, there is a separate *dimension table* for information related to each dimension, and one or more *fact tables* that relate dimension values to measure values.

Star schemas do not model dimension hierarchies very well. They require the complete information associated with a dimension hierarchy to be represented in a single table, even when different levels of the hierarchy have different properties. To mitigate this, the *snowflake schema* was proposed [12, 6], which is obtained by normalizing the star schema, with respect to the various attribute dependencies. Intuitively, each level of a dimension hierarchy is typically represented in a separate table.

While the snowflake schema removes some of the shortcomings of the star schema, it continues to have severe limitations in its support for modeling hierarchies, and for concisely expressing many useful classes of OLAP queries. These limitations are discussed in the next two subsections. In the sequel, by the "traditional approach to data warehousing", we mean modeling data warehouses using snowflake schemas, and querying them using SQL or its extensions, including those with the `CUBE` operator and its variants.

2.1 Limitations on Query Support

Hierarchies enrich the semantics of data in a warehouse and correspondingly enhance the class of interesting and meaningful queries one can pose against it. It is important to be able to express such queries intuitively and concisely, and optimize them effectively. Since hierarchies are not given a first-class status in SQL, even very simple OLAP queries have to be expressed using complex sequences of joins, perhaps together with unions. This makes queries verbose and hard to read. It also makes an implementation of these queries very inefficient.

We now illustrate the above limitations on querying with an example. For the purpose of this example, we temporarily assume that a snowflake schema is adequate for modeling real-life dimension hierarchies, and focus on the limitations of conventional SQL for expressing natural OLAP queries against such a schema. In the next subsection, we shall revisit the modeling issue.

Example 2.1 [A Simple Data Warehouse]
Figure 2 shows a data warehouse consisting of a fact table and several dimension tables. The fact table `sales` represents the dollar-sale-amount measure corresponding to the three dimensions: `product, location, time`. (Note the use of the auxiliary `Id` attributes for realizing the hierarchy using a snowflake schema.) We

```
fact table: sales(storeId, prodId, timeId, dollarAmt).
dimension tables:
  location: locn1(storeId, manager, cityId), locn2(cityId, city, manager, stateId),
        locn3(stateId, state, manager, regId), locn4(regId, region, manager, ctryId),
        locn5(ctryId, country, manager).
  time: time1(timeId, day, mthId), time2(mthId, month, qtrId),
        time3(qtrId, quarter, yrId), time4(yrId, year).
  product: prod1(prodId, name, ptfgbId), prod2(ptfgbId, brand, ptfgId),
        prod3(ptfgId, gender, ptfId), prod4(ptfId, family, ptId), prod5(ptId, type).
```

Figure 2: Data Warehouse with Snowflake Schema

(Q1): find location(s) managed by 'john smith'.
(Q2): find the total sales of each location (at any level of the hierarchy) in the NE region of USA.
(Q3): find the total sales of each location (at any level of the hierarchy) that grossed a total sale over $100,000.
(Q4): for each location that grossed a total sale over $100,000, give a breakdown of the sales by its immediate sub-locations.
(Q5): find each location that grossed a total sale over $100,000, and immediate sub-locations that contributed < 10% to the gross value.
(Q6): find the products that performed poorly (grossed less than $1000) in at least one sub-location of every location that grossed over $100,000 dollars (over all products).

Figure 3: OLAP Queries against Snowflake Schema

will refer to the example queries in Figure 3 throughout the paper.[1]

Query Q1 can be expressed using the following SQL query:

```
SELECT storeId
FROM   locn1
WHERE  manager = john.smith
       UNION
       ...
       UNION
SELECT ctryId
FROM   locn5
WHERE  manager = john.smith
```

Depending on the number of levels of the `location` hierarchy, this can be somewhat tedious. Queries Q2 and Q3 are quite similar. We illustrate how to express query Q3 in SQL:

```
SELECT   locn1.storeId, SUM(dollarAmt)
FROM     locn1, sales
WHERE    locn1.storeId = sales.storeId
GROUP BY locn1.storeId
HAVING   SUM(dollarAmt) > 100000
         UNION
         ...
         UNION
SELECT   locn5.ctryId, SUM(dollarAmt)
FROM     locn5, locn4, locn3, locn2, locn1,
         sales
WHERE    locn5.ctryId = locn4.ctryId AND
         locn4.regId = locn3.regId AND
         locn3.stateId = locn2.stateId AND
         locn2.cityId = locn1.cityId AND
         locn1.storeId = sales.storeId
GROUP BY locn5.ctryId
HAVING   SUM(dollarAmt) > 100000
```

[1] We assume all the Id attributes have compatible domains.

Queries Q4 and Q5 are about as complex to express in SQL, except that they both involve a nested sub-query with each of the five single block sub-queries in Q3 above. Query Q6 makes use of the SQL `contains` predicate, and is very complex to express in SQL. *Note that, in all cases, the SQL queries need to be level sensitive.* ▮

2.2 Limitations on Modeling

We now return to the modeling issue. The snowflake schema, while more flexible than the star schema, still has two key modeling limitations.

- Each dimension hierarchy has to be *balanced*, i.e., the length of the path from the root to any leaf of the hierarchy must be the same.
- All nodes at any one level of a dimension hierarchy have to be *homogeneous*, i.e., they must possess the same set of attributes.

There are many real-life examples where neither of these assumptions is valid.

For example, in the `location` dimension, stores in USA and in Monaco are forced to be classified in a similar fashion, although it is natural to have a five level hierarchy of `store-city-state-region-country` for the USA stores, and only a two level hierarchy of `store-country` for the Monaco stores. Indeed, depending on the geographical distribution of a given enterprise, even within a single country (say USA), dissimilar hierarchies for different regions may be appropriate, since there may be, e.g., many stores in the north-east and fewer in the south.

Snowflake schemas allow for heterogeneity across levels. This enables, e.g., `city` to have the attribute `population`, while `state` may have both `population` and `capital`. However, there can also be considerable heterogeneity within a level, and the snowflake schema offers poor support for modeling such situations. For example, in the `product` dimension, if pants and shirts are at the same level of the dimension hierarchy, they are forced to have the same set of attributes in a snowflake schema, which is quite unnatural: the attribute `inseam` for pants is not applicable to shirts, and the attribute `collarSize` is not applicable to pants.

We conclude this section by noting that the traditional approach does not give a first-class status to the notions of dimension and dimension hierarchies and instead requires explicit manipulation of the set of tables that represent dimension information. Besides, it imposes unrealistic restrictions on the way dimensions can be modeled via such tables.

3 The SQL($\mathcal{H}$) Data Model

Our approach for modeling dimension hierarchies is to (potentially) have an *arbitrary set* of tables for each level in the dimension hierarchy, with no table straddling levels. This SQL($\mathcal{H}$) approach is more flexible than the star or snowflake schemas, and does not require that the dimension hierarchies represented in this fashion be balanced or be homogeneous: (i) Different nodes at the same level of the dimension hierarchy (e.g., ties, pants) could be in different tables, and (naturally) have heterogeneous sets of attributes; (ii) Different "sibling" subtrees in a dimension hierarchy may have different heights, allowing for structural heterogeneity.

Intuitively, we want to treat a *hierarchy* as a tree, where each node corresponds to a tuple over some set of attributes. We want to regard the hierarchy as being based on a special, hierarchical, attribute, in that tables corresponding to all levels share this attribute.

We need a few definitions for formalizing the notion of a dimension.

Definition 3.1 [Hierarchical Domain] A *hierarchical domain* is a non-empty set $\mathcal{V}_\mathcal{H}$ such that it satisfies the following conditions:

1. The only predicates defined on this domain are $=, <, <=, <<, <<=$.
2. The equality predicate $=$ has the standard interpretation of syntactic identity.
3. The predicate $<$ is interpreted as a binary relation over $\mathcal{V}_\mathcal{H}$ such that the graph $G_<$ over the nodes $\mathcal{V}_\mathcal{H}$, with an arc from u to v exactly when $u < v$ holds, is a tree.
4. The predicate $<<$ is interpreted as the transitive closure of (the relation that interprets) $<$.
5. For two elements $u, v \in \mathcal{V}_\mathcal{H}$, $u <= v$ (resp., $u <<= v$) holds iff either $u < v$ or $u = v$ (resp., $u << v$ or $u = v$). ∎

Intuitively, $\mathcal{V}_\mathcal{H}$ is an abstract data type corresponding to hierarchies. The predicate $<$ corresponds to the child/parent relationship: $u < v$ holds iff u is a child of v according to the tree represented by $<$. Similarly, $u << v$ holds iff u is a (proper) descendant of v. We stress that we do not assume any specific implementation of hierarchical domains. For the results and techniques developed in this paper to be applicable, all we require is that the implementation of $\mathcal{V}_\mathcal{H}$ support tests based on the predicates $=, <, <<$, etc.[2] As an example, one way to implement the `location` hierarchy shown in Figure 4 is to represent the location ids as lists of (attribute, value) pairs, as in `country=USA`, `country=USA/region=NE`, `country=USA/region=NE/state=NJ`, etc. As another example, the time ids in the `time` hierarchy can be represented as lists of values, as in `1999`, `1999/2`, etc. Here, the representation indicates ancestors and descendants.

Whenever the domain of an attribute A is a hierarchical domain, we say that A is a *hierarchical attribute* or that A's type is hierarchical. We refer to $dom(A)$ as hierarchical values and sometimes as nodes.

Definition 3.2 [Hierarchy Schema] A *hierarchy schema* is a triple $\mathbf{H} = (G, \mathcal{A}, \sigma)$ such that: (i) G is a rooted DAG, with the root being a special node *All*; (ii) $\mathcal{A}$ is an attribute set, containing a unique hierarchical attribute A_h; and (iii) $\sigma : G \rightarrow 2^{\mathcal{A}}$ is a function that assigns each node $u \in G$ an attribute set $\sigma(u) \subseteq \mathcal{A}$, such that $\forall u \neq All, A_h \in \sigma(u)$, and $\sigma(All) = \emptyset$. All nodes of G, except *All*, are required to share the unique hierarchical attribute A_h. ∎

As an example, consider the `location` hierarchy (see Figure 1). Let the attributes associated with this hierarchy be {`locId, manager, city, state, region, country`}, and suppose `locId` is the hierarchical attribute. Then the attribute sets associated with the successive nodes from the bottom up are {`locId, manager`}, {`locId, city, manager`}, {`locId, state, manager`}, {`locId, region, manager`}, {`locId, country, manager`}, and {} for the node *All*. Notice that `locId` simultaneously plays the role of `storeId`, `cityId`, `stateId`, `regId` and `ctryId`, and allows for tests of appropriate child/parent and descendant/ancestor relationships among hierarchy nodes.

[2] Preferably efficiently!

Definition 3.3 [Hierarchy] A *hierarchy* (*instance*) corresponding to a hierarchy schema $\mathbf{H} = (G, \mathcal{A}, \sigma)$ is a collection of tables $\mathcal{H}$, satisfying the following conditions: (i) each table $r \in \mathcal{H}$ corresponds to a unique node $u \in G$, $u \neq All$, and r is a table over $\sigma(u)$; (ii) for any table $r \in \mathcal{H}$ and any two hierarchical values $x, y \in \pi_{A_h}(r)$, neither $x << y$ nor $y << x$ holds;[3] (iii) whenever two tables $r, s \in \mathcal{H}$ correspond to a pair of nodes $u, v \in G$ such that v is a parent (resp., ancestor) of u, then $\forall$ tuples $t_r \in r : \exists t_s \in s : t_r[A_h] < t_s[A_h]$ (resp., $\forall t_r \in r : \exists t_s \in s : t_r[A_h] << t_s[A_h]$). ▮

The first condition says that in a tabular representation of a hierarchy, all tuples in a given level that are represented in one table are over the same attribute set. The second condition essentially says that no table can straddle hierarchy levels. The third condition says that for every tuple in a child (descendant) table, there must correspond a tuple in the parent (ancestor) table such that the hierarchical values in these two tuples are related by the child/parent (descendant/ancestor) relationship. It can be seen that our notion of hierarchy closely corresponds to the notion of hierarchy schema and instance as defined in [3, 10]. The main difference is that we do not assume that nodes (levels) in a hierarchy schema are necessarily named (although our examples show such names, for convenience).

We are now ready to formalize the notion of a dimension.

Definition 3.4 [Dimension] A *dimension schema* $D(\mathbf{H})$ is a name D (called the dimension name) together with a hierarchy schema $\mathbf{H} = (G, \mathcal{A}, \sigma)$. We refer to the attributes $\mathcal{A}$ as the attribute set associated with dimension D.

A *dimension instance* $D(\mathcal{H})$ over a dimension schema $D(\mathbf{H})$ is a dimension name D together with a hierarchy instance $\mathcal{H}$ of $\mathbf{H}$. ▮

Consider the hierarchy schema associated with dimension `location`, described above. An instance of this schema might consist of the tables `loc1`, ..., `loc5` (see Figure 4). Note the difference with the representation of this dimension using the five tables `locn1`–`locn5` in Figure 2. It is important to realize that in each table `loc`*i*, $1 \leq i \leq 5$, `locId` must be a key. The definition of a dimension instance permits having an arbitrary set of tables (subject to the conditions mentioned). So, in addition to the top-level country table, by introducing separate hierarchy nodes for various countries in the `location` hierarchy of Figure 1, we can have a set of four tables for locations inside USA, perhaps just three tables for locations inside Canada, and only one for those in Monaco. Alternatively, the table `loc1` can contain the tuples for all store locations in USA, Canada and Monaco.[4] Similarly, in the case of dimension `product`, one can have different tables for pants and shirts, even if they are at the same level of the `product` dimension hierarchy, thus permitting modeling of heterogeneous and unbalanced hierarchies with ease.

[3] Note that consequently, neither $u < v$ nor $v < u$ can hold.

[4] In this case, the hierarchy node corresponding to `loc1` may have multiple parents, i.e., the hierarchy schema graph is a DAG.

Definition 3.5 [Data Warehouse Schema] We define a *data warehouse schema* (DW schema) in the SQL($\mathcal{H}$) model as a set of dimension schemas $D_i(\mathbf{H}_i)$, with associated hierarchical attributes A_h^i, $1 \leq i \leq k$, together with a set of fact table schemas[5] of the form $f(A_h^{j_1}, ..., A_h^{j_n}, B_1, ..., B_m)$, where $D_{j_1}, ..., D_{j_n}$ are a subset of the dimensions $D_1, ..., D_k$, and $B_j, 1 \leq j \leq m$, are additional attributes, including the measure attributes. ▮

Recall that each dimension schema itself is composed of a collection of table schemas corresponding to that dimension. As an example, a DW schema corresponding to the data warehouse of Example 2.1 is shown in Figure 4. While the schemas for `location` and `time` closely correspond to the hierarchies shown in Figure 1, for illustrative reasons, we have chosen to make the schema for `product` heterogeneous and unbalanced. For instance, the top level *All* (not shown) of `product` is divided into "formal" and "casual",[6] which are then subdivided using quite different criteria. Notice the considerable heterogeneity in the structure and contents of the `product` dimension hierarchy shown in the figure, compared with that in Example 2.1. An instance of this DW schema would consist of tables over the various table schemas in the figure, subject to the conditions in Definition 3.5.

4 The SQL($\mathcal{H}$) Query Language

The SQL($\mathcal{H}$) model for a data warehouse is consistent with the relational model in that it is possible to query it using standard SQL. In particular, SQL queries continue to have their standard semantics against our extended model for data warehouses.[7] However, to take full advantage of the extended model, we propose simple but powerful extensions to SQL, which exploit the fact that dimensions and their associated hierarchies have a first-class status in the SQL($\mathcal{H}$) model. For simplicity of exposition, we present these extensions in a stage-wise manner.

4.1 Single Block SQL($\mathcal{H}$) Queries

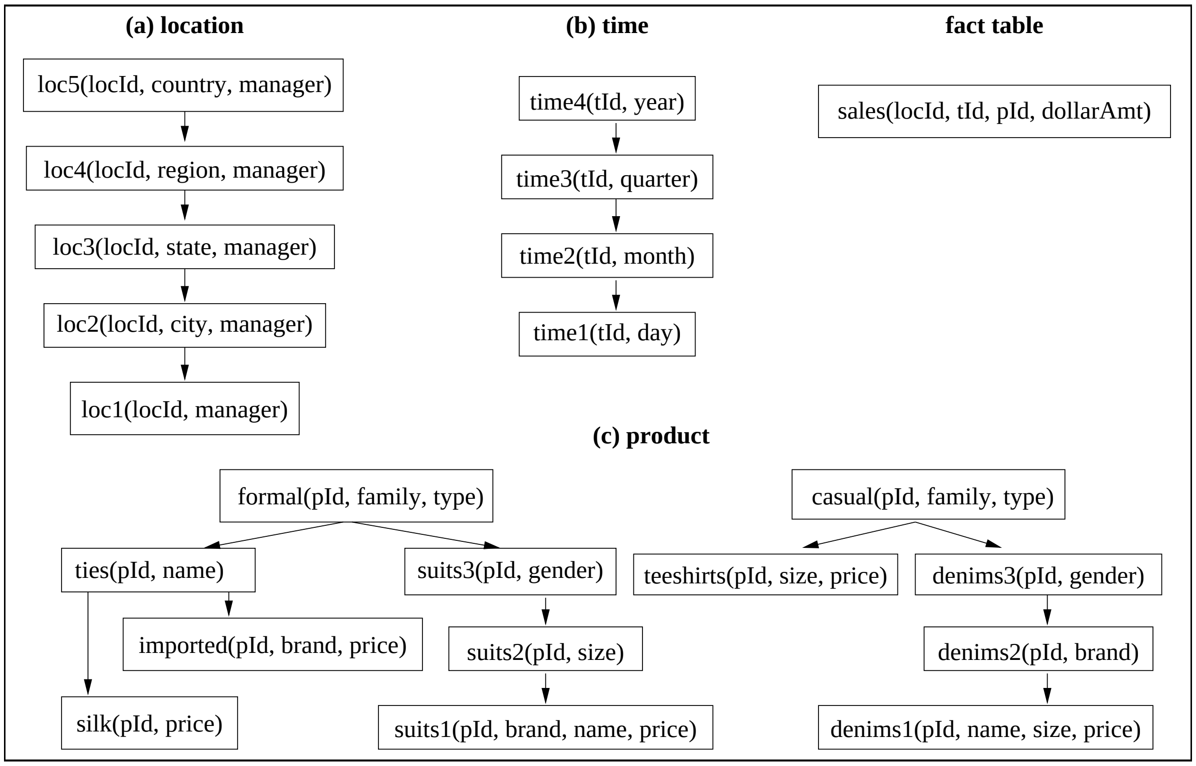
For single block queries, our extensions are of two types.

`DIMENSIONS` clause: We introduce a new `DIMENSIONS` clause in SQL($\mathcal{H}$) queries, where dimension names

[5] Following standard practice (e.g., [6]), we let a DW schema contain one or more fact tables.

[6] We assume all products are clothes.

[7] To perform star and snowflake joins, additional join attributes need to present in the dimension and the fact tables.



(a) location
loc5(locId, country, manager)
loc4(locId, region, manager)
loc3(locId, state, manager)
loc2(locId, city, manager)
loc1(locId, manager)
(b) time
time4(tId, year)
time3(tId, quarter)
time2(tId, month)
time1(tId, day)
fact table
sales(locId, tId, pId, dollarAmt)
(c) product
formal(pId, family, type)
ties(pId, name)
suits3(pId, gender)
imported(pId, brand, price)
suits2(pId, size)
silk(pId, price)
suits1(pId, brand, name, price)
casual(pId, family, type)
teeshirts(pId, size, price)
denims3(pId, gender)
denims2(pId, brand)
denims1(pId, name, size, price)

Figure 4: Data Warehouse Schema in the SQL($\mathcal{H}$) Model: Three Dimension Hierarchies and a Fact Table

of interest to the query can be listed, just like tables of interest are listed in the `FROM` clause.

Just as a table R mentioned in the `FROM` clause implicitly declares a tuple variable ranging over R, each dimension name D mentioned in the `DIMENSIONS` clause implicitly declares a dimension variable ranging over that dimension.

Hierarchical predicates: In SQL, domain expressions (DE) of the form $T.A$, T a tuple variable and A an attribute, can be used in various clauses including `SELECT, WHERE, HAVING, GROUP BY`. DEs can be compared with other DEs or with values of a compatible type, using the standard comparison operators $=, <=, <, >, >=, <>$.

In SQL($\mathcal{H}$), we extend the class of domain expressions to include those of the form $V.A$, where V is a dimension variable and A is an attribute associated with the relevant dimension. Such DEs can participate in comparisons just as in SQL.

We also extend domain expressions to include *hierarchical domain expressions* (HDEs), which are of the form $W.A_h$, where W is a tuple variable or a dimension variable and A_h is a hierarchical attribute. A HDE can be compared with another HDE or with a value of a compatible type using the hierarchical predicates $=, <, <=, <<, <<=$. $A < B$ means that A is a child of B in the hierarchy; $A << B$ means that A is a (proper) descendant of B in the hierarchy. The operators $<=$ and $<<=$ correspond to non-proper children and descendants, respectively.

To appreciate the power of these two simple constructs, we present a few examples below.

Example 4.1 [Dimensional Selection] The following single block SQL($\mathcal{H}$) query, `Q1'`, captures the query "find location(s) managed by john smith" (`Q1` in Figure 3).[8]

```
SELECT     L.locId
DIMENSIONS location L
WHERE      L.manager = john.smith
```

Note the simplicity of `Q1'` compared with the corresponding SQL expression.

Here, `L` is a dimension variable that ranges over the heterogeneous set of tuples in any of the tables associated with dimension `location`. One can drop the dimension variable `L` from this query. In this case, the attribute `manager` in the `WHERE` clause would implicitly refer to dimension `location`. As in SQL, when there is ambiguity, attributes must be preceded by table names or dimension names (or their aliases). ▮

[8] In general, we shall use the primed versions of query names, `Q1'`, `Q2'`, etc. in the text to indicate the SQL($\mathcal{H}$) expressions corresponding to queries `Q1`, `Q2`, etc.

Example 4.2 [Hierarchical Joins/Aggregation] The following single block SQL($\mathcal{H}$) query, `Q3'`, captures the query "find the total sales of each location that grossed a total sale over \$100,000" (`Q3` in Figure 3).[9]

```
SELECT     L.locId, SUM(dollarAmt)
DIMENSIONS location L
FROM       sales
WHERE      sales.locId <<= L.locId
GROUP BY   L.locId
HAVING     SUM(dollarAmt) > 100000
```

Note that the SQL($\mathcal{H}$) query does not involve (unions of) long sequences of joins, unlike the corresponding SQL expression. ∎

4.2 Nested SQL($\mathcal{H}$) Queries

Our two constructs introduced in the previous section are also very useful in nested SQL($\mathcal{H}$) queries.

Example 4.3 [Subqueries in the `WHERE` Clause] The following SQL($\mathcal{H}$) query, `Q4'`, captures the query "for each location that grossed a total sale over \$100,000, give a breakdown by its immediate sub-locations" (`Q4` in Figure 3).

```
SELECT     L1.locId, L2.locId, SUM(dollarAmt)
DIMENSIONS location L1 L2
FROM       sales
WHERE      sales.locId <<= L1.locId AND
           L2.locId < L1.locId AND
           L1.locId in (
              SELECT     L.locId
              DIMENSIONS location L
              FROM       sales S
              WHERE      S.locId <<= L.locId
              GROUP BY   L.locId
              HAVING     SUM(S.dollarAmt) >
                         100000)
GROUP BY   L1.locId, L2.locId ∎
```

Example 4.4 [Subqueries in the `FROM` Clause] The following SQL($\mathcal{H}$) query, `Q5'`, captures the query "find each location that grossed a total sale over \$100,000, and its immediate sub-locations that contributed to < 10% of the gross value" (`Q5` in Figure 3).

```
SELECT     T.locId, L2.locId
DIMENSIONS location L2
FROM       sales S2,
           (SELECT     L1.locId,
                       SUM(dollarAmt) AS total
           DIMENSIONS location L1
           FROM        sales S1
           WHERE       S1.locId <<= L1.locId
           GROUP BY    L1.locId
           HAVING      SUM(S1.sales) >
                       100000) T
WHERE      S2.locId <<= T.locId AND
           L2.locId < T.locId
GROUP BY   T.locId, L2.locId
HAVING     SUM(S2.dollarAmt) < 0.1*T.total
```

Query `Q5` can also be expressed by defining the result of query `Q3'` as a table view, and defining a regular SQL query that makes use of this view. ∎

In keeping with the use of hierarchical predicates for simple conditions in the `WHERE` clause, we also allow their use in nested queries. We mention two such extensions below.

The condition DE θ qual setQuery:

In SQL, one can use such a condition, where θ is a comparison operator, "qual" is one of *all, any,* and "setQuery" is a unary SQL query of a type compatible with DE.

We extend this by: (i) allowing the full class of DEs possible in SQL($\mathcal{H}$) in the above condition, and (ii) by allowing conditions of the form "HDE θ_h qual hierSetQuery", where θ_h is a hierarchical predicate and "hierSetQuery" is an SQL($\mathcal{H}$) query returning a set of hierarchical values from the same domain as HDE.

The `contains` predicate:

In SQL, one can express conditions involving universal quantifiers using expressions of the form "sqlQuery1 `contains` sqlQuery2", where "sqlQuery1" and "sqlQuery2" are SQL queries of the same type (and hence over the same number of attributes). The semantics of this condition is a test as to whether the result of "sqlQuery1" is a superset of that of "sqlQuery2".

In SQL($\mathcal{H}$), we additionally allow conditions of the form "hierSqlhQuery1 θ_h `contains` hierSqlhQuery2", where both "hierSqlhQuery1" and "hierSqlhQuery2" are SQL($\mathcal{H}$) queries returning a set of hierarchical values from the same domain, and θ_h is as before. The semantics of this condition is a test as to whether for every hierarchical value u in the result of "hierSqlhQuery2", there is a hierarchical value v in the result of "hierSqlhQuery1", such that v θ_h u holds.

Example 4.5 [Hierarchical `contains`] The following SQL($\mathcal{H}$) query, `Q6'`, captures the query "find the products that performed poorly (i.e., grossed less than \$1000) in at least one sub-location of every location that grossed over \$100,000 dollars (over all products)" (`Q6` in Figure 3).

```
SELECT     P.pId
DIMENSIONS product P
WHERE      (SELECT     L1.locId
           DIMENSIONS location L1
```

[9] The more natural query "find the *lowest* level locations, and their sales, among locations that grossed a total sale over \$100,000" can also be easily expressed in SQL($\mathcal{H}$), though not as a single block query.

```
FROM        sales S1
WHERE       S1.pId <<= P.pId AND
            S1.locId <<= L1.locId
GROUP BY    L1.locId
HAVING      SUM(S1.dollarAmt) <
            1000)

<<= contains

(SELECT     L2.locId
DIMENSIONS location L2
FROM        sales S2
WHERE       S2.locId <<= L2.locId
GROUP BY    L2.locId
HAVING      SUM(S2.dollarAmt) >
            100000)  ▮
```

4.3 Semantics of SQL($\mathcal{H}$) Queries

In this section, we define the semantics of single block SQL($\mathcal{H}$) queries.

Before we pin down the semantics, there is an issue to settle. The SQL($\mathcal{H}$) model supports two kinds of first-class objects: *tables* (corresponding to the fact tables) and *dimensions* (with their hierarchies). Even though for practical considerations, dimensions may be realized using a set of tables, they do have a first-class status in the query language, as illustrated in the preceding sections. Thus the input to an SQL($\mathcal{H}$) query consists of both tables and dimensions. If so, what should be the type of the output? In principle, both queries that produce tables as output (like conventional SQL) and those that produce dimensions (i.e., a set of tables that realize them) as output are meaningful. Indeed, we anticipate applications for both types of queries.

The various SQL($\mathcal{H}$) queries presented earlier in this paper all produce tables as output. The following variant of query `Q1'` is an example of an SQL($\mathcal{H}$) query that produces a dimension as an output.

```
SELECT     L.*
DIMENSIONS location L
WHERE      L.manager = john.smith
```

The result in this case is a hierarchy instance, where each table is a subset of the corresponding table in the input, corresponding to the condition `L.manager = john.smith`.

For lack of space, in this paper, we confine ourselves to SQL($\mathcal{H}$) queries that produce tables as output. A natural restriction on SQL($\mathcal{H}$) queries to ensure that the result is a table is that the `SELECT` clause contain only: (a) the hierarchical `Id` attributes (e.g., `locId`), (b) dimension attributes that appear in every table of the dimension (e.g., `manager`), and (c) aggregates of measure attributes (e.g., `SUM(dollarAmt)`). We refer to such queries as *uniform* SQL($\mathcal{H}$) queries.

Consider a generic uniform SQL($\mathcal{H}$) query:

```
Q: SELECT     domExpList, aggList
   DIMENSIONS dimList
   FROM       fromList
   WHERE      whereConditions
   GROUP BY   groupbyList
   HAVING     haveConditions
```

An important question is the following. A tuple variable (implicit or explicit) in SQL ranges over the set of tuples in the associated table. If so, what exactly does a dimension variable range over?

> We propose that a dimension variable should range over the set of nodes in the hierarchy associated with the dimension instance.

Given that each hierarchy (instance) node corresponds to a tuple, this means that a dimension variable ranges over a heterogeneous set of tuples corresponding to the various nodes in its associated hierarchy. While this is quite unlike the case with tuple variables (which always range over homogeneous sets of tuples in a given table), we shall show that uniform SQL($\mathcal{H}$) queries are always well-defined and "well-typed".

Following the notation used in [13], we define the semantics of a uniform SQL($\mathcal{H}$) query Q as a function

$$Q : \mathcal{D} \rightarrow \mathcal{R}$$

from input databases $\mathcal{D}$ to tables $\mathcal{R}$ over the output scheme of the query. For a database $\mathbf{D} \in \mathcal{D}$, let $\mathcal{T}_D$ denote the set of all tuples appearing in any table in $\mathbf{D}$, including fact and dimension tables. Let τ be the set of tuple variables in Q and η the set of dimension variables in Q. Define an *instantiation* as a function $\imath : (\tau \cup \eta) \rightarrow \mathcal{T}_D$, that maps each tuple variable to a tuple in the appropriate (fact) table,[10] and each dimension variable to a tuple in some table associated with that dimension. For each condition `cond` appearing in the `WHERE` clause, a notion of satisfaction is defined as follows. Let $\imath(X)[A]$, X being a tuple/dimension variable, denote the restriction of the tuple $\imath(X)$ to the attribute A.

- if `cond` is of the form $T.A\ \theta$ opnd, where T is a tuple variable, A is an attribute, and opnd is either a DE or a value of the appropriate type, then $\imath$ satisfies `cond` iff $\imath(T)[A]$ stands in relationship θ to $\imath$(opnd).[11] This is exactly the same as for SQL.

- if `cond` is of the form $V.A\ \theta$ opnd, where V is a dimension variable, then $\imath$ satisfies `cond` iff $\imath(V)$ is mapped to a tuple in a dimension table for which attribute A is defined, and further $\imath(V)[A]$ stands in relationship θ to $\imath$(opnd).

[10] Recall that there might be more than one fact table.

[11] We assume $\imath(c) = c$, for every constant value c.

- if `cond` is of the form $W.A_h\ \theta_h$ opnd, where W is a tuple or a dimension variable and A_h is a hierarchical attribute, then $\imath$ satisfies `cond` iff $\imath(V)[A_h]$ is a θ_h-relative of $\imath$(opnd), according to the hierarchical domain of A_h.

Let $\mathcal{I}_Q$ denote the set of all instantiations that satisfy the conditions `whereConditions`. Then a tuple assembly function can be defined as

$$tuple_Q(\imath) = \bigotimes_{\text{“}X.A\text{”} \in \texttt{domExpList}} \imath(X)[A]$$

Here, the predicate "$X.A$" $\in$ `domExpList` indicates the condition that the attribute denotation $X.A$ literally appears in the list of domain expressions `domExpList` in the `SELECT` statement. The symbol $\bigotimes$ denotes concatenation. For an instantiation $\imath$, $tuple_Q(\imath)$ thus produces a tuple over the attributes listed in the `domExpList` part of the `SELECT` statement. If Q is a query without aggregation (i.e. `aggList`, `GROUP BY` clause, and `HAVING` clause are empty), then the result of the query is captured by the function

$$Q(\mathbf{D}) = \{tuple_Q(\imath) \mid \imath \in \mathcal{I}_Q\}$$

Handling aggregation is done in the obvious manner, using an appropriately defined equivalence relation, just as is done for SQL. The following result is straightforward.

Proposition 4.1 *Let Q be any legal uniform SQL($\mathcal{H}$) query over a DW schema* **S**. *Then Q is well-typed, in the sense that it maps every instance* **D** *of* **S** *to a table over the output scheme of Q.*

Proof: Follows from the semantics of SQL($\mathcal{H}$) queries given above. ▌

5 Direct Evaluation of SQL($\mathcal{H}$) Queries

We present algorithms for the efficient evaluation of SQL($\mathcal{H}$) queries in this section. The two key challenges that we need to address, over standard SQL evaluation, are: (a) dealing with dimension selection conditions, such as "`L.manager = john.smith`" (in query `Q1'`), and (b) dealing with hierarchical joins, such as the one between `location` and `sales` using the predicate "`S.locId <<= L.locId`" (in queries `Q2'` and `Q3'`). We describe our solution for dealing with dimension selections in Section 5.1, and present an efficient approach for hierarchical joins in Sections 5.2 and 5.3.

5.1 Selections and Dimension Indexes

Database indexes available in current database systems, such as B-tree indexes [7], hash indexes [15] and bitmap indexes [16, 4], retrieve tuples of a *single table* with specified values for one or more attributes.

Since dimensions are represented by a set of heterogeneous tables, selection conditions in SQL($\mathcal{H}$) queries implicitly range over tuples in *multiple tables*. For example, "`L.locId <<= country=USA/region=NE`" in query `Q2'` will be satisfied by tuples corresponding to states, cities, and individual stores, which are all represented in different tables. Again, the condition "`L.manager = john.smith`" on the `location` dimension in query `Q1'` could be satisfied at any level of the dimension hierarchy. Using conventional (single table) indexes, determining all nodes in a dimension hierarchy that match a given condition would require access to multiple (possibly large number of) indexes, and can be quite inefficient.

Our solution to the problem of efficiently dealing with dimension selections involves construction of indexes, called *dimension indexes*, on (one or more) attributes of a single dimension. Since multiple dimension tables may share an attribute (e.g., the `locId` and the `manager` attributes are shared by all tables in the `location` dimension), dimension indexes are inherently multi-table indexes. Instead of associating the set of tuples in a single table with a keyvalue, dimension indexes (conceptually) associate a set of (table, tuple) pairs with a keyvalue. When multiple tuples in a single table match a keyvalue, factoring out the table can result in a potential for compression. Dimension indexes can be organized as B-trees, hash indexes, bitmap indexes, or any standard index structure.

Dimension indexes are similar, in spirit, to join indexes [21] which typically associate attribute values and tuples of two tables, and their generalizations. However, join indexes (as the name suggests) have been used only to support join processing, not multi-table selections.

5.2 Hierarchical Joins

Star joins have been used to denote the join of a fact table with multiple dimension tables, represented using a star schema. The star join conditions typically include selection conditions on the dimension tables as well. The specialized nature of star schemas makes join indexes especially attractive for OLAP queries [17, 6].

Different flavors of bitmap join indexes have been proposed to efficiently deal with star joins [17, 18]. In its simplest form [17], a bitmap join index is a bitmap index on the fact table T based on a single attribute A of a (star schema) dimension table V.

> The bitmap join index is useful precisely when the OLAP query specifies a selection condition on attribute A of dimension table V, and a join condition between the fact table T and the dimension table V.

For example, one can maintain a bitmap join index on the `sales` fact table based on the attribute `state` in the `location` dimension. Thus, the index entry for NJ is a bitmap that identifies the tuples in the sales table that correspond to stores whose state is NJ. This index entry can be used, e.g., in a star join OLAP query that specifies the selection condition "location.state = NJ" and the join condition "location.storeId = sales.storeId".

Modeling limitations of star schemas have led to snowflake schemas for data warehouses. However, there has not been much research reported in enhancing star join algorithms to efficiently perform snowflake joins. The approach of [17], of using bitmap join indexes, while applicable, is not as efficient for snowflake joins as in the case of star joins. To see why, consider the following query, which is one of the three subqueries in the SQL expression of Query Q2, using the snowflake schema of Figure 2.

```
SELECT   locn1.storeId, SUM(dollarAmt)
FROM     locn1, locn2, locn3, locn4, locn5,
         sales
WHERE    locn5.country = USA AND
         locn4.state = NE AND
         locn5.ctryId = locn4.ctryId AND
         locn4.regId = locn3.regId AND
         locn3.stateId = locn2.stateId AND
         locn2.cityId = locn1.cityId AND
         locn1.storeId = sales.storeId
GROUP BY locn1.storeId
```

Note that there is *no* join condition between the fact table (sales) and any of the dimension tables on which selection conditions have been specified (locn4 and locn5). Multiple bitmap join indexes (between tables corresponding to successive levels of the dimension hierarchy) would have to be built to support the above sequence of joins in an SQL query evaluation engine. Clearly, this approach is considerably less efficient for snowflake joins than in the case of star joins. For snowflake joins to have comparable efficiency to star joins, bitmap transitive join indexes could be built for this task. However, using them effectively would require the SQL query evaluation engine to have a detailed knowledge of the representation of dimension hierarchies as tables, and a substantial change to the query processing strategies.

Modeling dimension hierarchies using the SQL($\mathcal{H}$) model not only results in conceptual simplicity and elegance, but also allows for computational efficiency that is comparable to the use of bitmap join indexes for star joins. Our solution involves the use of a star join style algorithm for *hierarchical joins*, i.e., joins that use hierarchical predicates. The SQL($\mathcal{H}$) query Q2' can be expressed as follows, using the DW schema of Figure 4.

```
SELECT     L.locId, SUM(dollarAmt)
DIMENSIONS location L
FROM       sales
WHERE      sales.locId <<= L.locId AND
           L.locId <<= country=USA/region=NE
GROUP BY   L.locId
```

Note that the hierarchical join condition *is* between the fact table (sales) and the dimension on which a selection condition has been specified (location). Given a bitmap join index on the sales table based on the pair of attributes country and region of dimension location, *the hierarchical join could be performed as efficiently as the star join.* We next examine how bitmap join indexes that enable hierarchical joins can be constructed efficiently.

5.3 Bitmap Join Indexes for Hierarchical Joins

The bitmap join index used in computing the star join, as described in [17], is based on the assumption that the join between the fact table and the dimension table is an *equality join*. In SQL($\mathcal{H}$) queries, many different join predicates, such as $=, <, <=, <<, <<=$, can be used. To efficiently support each of these predicates between the Id attribute of the dimension and the corresponding attribute of the fact table, different bitmap join indexes may be needed. The choice of which ones are created depends on the predicates used in the anticipated query workload.

In this section, we present an efficient algorithm, called ComputeBitmapHJoinIndex, for computing a $(<<=, =)$-bitmap join index, which could be used for hierarchical join computation when the hierarchical predicate $<<=$ is used in the join condition between the fact table and the dimension (e.g., sales.locId <<= L.locId), and the equality predicate ($=$) is used in the selection condition on a dimension attribute (e.g., L.manager = john.smith). This algorithm is loosely based on the I/O efficient algorithm for computing hierarchical semijoins for querying network directories [11].

Let T denote the fact table, V the dimension, and A the dimension attribute on which the $(<<=, =)$-bitmap join index needs to be constructed. The resulting (uncompressed) bitmap join index would have $\mid A \mid$ bit-vectors, each with $\mid T \mid$ bits. A naive algorithm would compute this index as follows, assuming a total ordering on the VId attribute of the dimension and a total ordering on the tuple identifiers (RID's) of the fact table.

- First, allocate a 2-d bit-array VT with $\mid V \mid$ rows and $\mid T \mid$ columns, with each bit initialized to 0.[12]

  Examine each pair of (T tuple, V tuple). If $T.VId <<= V.VId$, set bit $VT[V.VId][T.RID] = 1$.

- Next, allocate a 2-d bit-array AT with $\mid A \mid$ rows and $\mid T \mid$ columns, with each bit initialized to 0.

[12]Zeroing out of an entire array of bits can be done very efficiently in most systems, and is not counted in our cost analysis.

```
Algorithm ComputeBitmapHJoinIndex $(T, V, A)$ {
    $T$ denotes the fact table, with $| T |$ tuples.
    $V$ denotes the dimension, with $| V |$ tuples
            in the various tables of dimension $V$.
    $A$ denotes $V$'s attribute, with $| A |$ values.
    Let $VId$ denotes the join attribute of $T$ and $V$.
    Let $L$ denote the leaf nodes of dimension $V$, and
            $| L |$ denote the number of such nodes.

    /* Phase 1: scan the fact table and build a
            bitmap index on $T.VId$ */
    Let $LT$ be a bit-array with $| L |$ rows and
            $| T |$ columns, initialized to 0's.
    for each tuple in $T$ with tuple identifier $T.RID$
        $LT[T.VId][T.RID] = 1$

    /* Phase 2: use the tree structure of $VId$'s domain
            to build a complete hierarchy bitmap */
    Let $V_s$ be a preorder traversal of the $VId$'s in $V$.
    Let $VT$ be a bit-array with $| V |$ rows and
            $| T |$ columns, initialized to 0's.
    /* stack $S$ identifies (parent, child) pairs */
    Initially stack $S$ is empty.
    node $v_l$ = firstElement($V_s$).
    repeat {
        if (stack $S$ is empty)
            push $v_l$ on top of stack $S$.
            $v_l$ = nextElement($V_s$).
        else {
            let $v_t$ be the node at the top of the stack $S$.
            if ($v_l < v_t$) /* is $v_l$ a child of $v_t$? */
                push $v_l$ on top of stack $S$.
                $v_l$ = nextElement($V_s$).
            else {
                if ($v_t$ is a leaf node)
                    $VT[v_t] = LT[v_t]$
                pop stack.
                if (stack $S$ is not empty) {
                    let $v_b$ be the node at the top of $S$.
                    /* bit-wise OR with child's bitmap */
                    $VT[v_b] = VT[v_b] \quad || \quad VT[v_t]$.
                }
            }
        }
    } until (all nodes in $V_s$ have been processed
            and stack $S$ is empty).

    /* Phase 3: build bitmap join index on $A$ values */
    Let $AT$ be a bit-array with $| A |$ rows and
            $| T |$ columns, initialized to 0's.
    /* many $V$ tuples may have the same $A$ value */
    for each tuple in $V$
        if ($V.A = val$)
            $AT[val] = AT[val] \quad || \quad VT[V.VId]$.
}
```

Figure 5: Computing a $(<<=, =)$-Bitmap Join Index

Examine each V tuple. If $V.A = val$, set $AT[val] = AT[val] \quad || \quad VT[V.VId]$, where $AT[val]$ and $VT[V.VId]$ represent bit-vectors of $| T |$ bits, and "$||$" is the bit-wise OR.

The naive algorithm scans $O(| V | * | T |)$ tuples, performs $| V | * | T |$ bit-level operations, and $| V |$ bit-vector operations. The overall cost is dominated by the cost of scanning tuples.

We describe Algorithm `ComputeBitmapHJoinIndex`, and analyze its computational complexity. In the first phase, the fact table tuples are scanned once, and a bitmap index on the VId attribute of the fact table T is constructed. In the second phase, the algorithm takes advantage of the hierarchical structure of the VId domain, and (recursively) computes bitmaps for parent nodes in the VId domain based on the bitmaps of their children nodes. (The leaf-level bitmaps are identical to the bitmaps constructed in Phase 1.) In this phase, each node's bit-vector is touched precisely once, without accessing any tuple in any table. In the third phase, the desired $(<<=, =)$-bitmap join index is constructed on fact table T based on the A attribute of dimension V. Algorithm `ComputeBitmapHJoinIndex` scans each fact table tuple and dimension table tuple exactly once, at the cost of a few more bit-vector operations than the naive algorithm.

Theorem 5.1 *Algorithm* `ComputeBitmapHJoinIndex` *correctly computes the $(<<=, =)$-bitmap join index. Further, it scans $| V | + | T |$ tuples, performs $| T |$ bit-level operations, and $2* | V |$ bit-vector operations.*
∎

Bitmap join indexes for other hierarchical predicates can be constructed just as efficiently.

6 Conclusions

By recognizing dimensions and dimension hierarchies as a first class construct in the context of a relational data warehouse, and reflecting this through suitable enhancements to SQL, we have demonstrated the considerable benefits that can be derived, both in ease of query expression and in computational efficiency of query evaluation.

Giving dimension hierarchies a first class status in the SQL($\mathcal{H}$) model permitted flexible modeling of structural and schematic heterogeneity, situations often arising in practice that cannot be modeled satisfactorily using the star/snowflake schemas. To take full advantage of the SQL($\mathcal{H}$) model, we proposed two key extensions to the SQL($\mathcal{H}$) query language: a novel `DIMENSIONS` clause, and the use of hierarchical predicates in conditions. The ease of expression of a large variety of OLAP queries is testimony to the power of these simple constructs. Finally, we complemented our extensions to the query language and data model with an algorithm for the efficient construction of a family of bitmap join indexes. These indexes enable the efficient computation of hierarchical joins, in the same way that bitmap indexes are instrumental in the efficient computation of star joins.

Our work establishes the foundations for further research in the important area of hierarchies in data warehouses.

References

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the International Conference on Very Large Databases*, pages 506–521, 1996.

[2] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. In *Proceedings of the International Conference on Very Large Databases*, 1997.

[3] L. Cabibbo and R. Torlone. Querying multidimensional databases. In *Proceedings of the 6th DBPL Workshop*, pages 253–269, 1997.

[4] C.-Y. Chan and Y. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 355–366, 1998.

[5] D. Chatziantoniou and K. A. Ross. Querying multiple features of groups in relational databases. In *Proceedings of the International Conference on Very Large Databases*, pages 295–306, 1996.

[6] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, Mar. 1997.

[7] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[8] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Datacube : A relational aggregation operator generalizing group-by, cross-tab, and subtotals. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 152–159, 1996. Also available as Microsoft Technical Report MSR-TR-95-22.

[9] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 205–216, 1996.

[10] C. Hurtado, A. Mendelzon, and A. Vaisman. Maintaining data cubes under dimension updates. In *Proceedings of the IEEE International Conference on Data Engineering*, 1999.

[11] H. V. Jagadish, L. V. S. Lakshmanan, T. Milo, D. Srivastava, and D. Vista. Querying network directories. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Philadelphia, PA, June 1999.

[12] R. Kimball. *The data warehouse toolkit*. John Wiley, 1996.

[13] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL – A language for querying and restructuring multidatabase systems. In *Proceedings of the International Conference on Very Large Databases*, pages 239–250, 1996.

[14] W. Lehner. Modeling large scale OLAP scenarios. In *Proceedings of the International Conference on Extending Database Technology*, 1998.

[15] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the International Conference on Very Large Databases*, pages 212–223, 1980.

[16] P. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, Lecture Notes in Computer Science, No. 359, pages 40–59, 1987.

[17] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, Sept. 1995.

[18] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 38–49, 1997.

[19] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proceedings of the International Conference on Very Large Databases*, pages 116–125, 1997.

[20] K. A. Ross, D. Srivastava, and D. Chatziantoniou. Complex aggregation at multiple granularities. In *Proceedings of the International Conference on Extending Database Technology*, 1998.

[21] P. Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, June 1987.

[22] J. Widom. Research problems in data warehousing. In *Proceedings of the Fourth International Conference on Information and Knowledge Management (CIKM)*, pages 25–30, Baltimore, MD, Nov. 1995.

[23] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 159–170, 1997.