Design of data warehouses using metadata

L. Wu\textsuperscript{a}, L. Miller\textsuperscript{a}, S. Nilakanta\textsuperscript{b,*}

\textsuperscript{a}Department of Computer Science, College of Liberal Arts and Sciences, 201 Atanasoff, Iowa State University, Ames, IA 50011, USA
\textsuperscript{b}Department of Logistics, Operations and Management Information Systems, College of Business, 300 Carver Hall, Iowa State University, Ames, IA 50011, USA

Received 5 October 1999; revised 21 June 2000; accepted 18 July 2000

Abstract

Data warehouses have become an instant phenomenon in many large organizations that deal with massive amounts of information. Drawing on the experiences from the systems development field, we surmise that an effective design tool will enhance the success of warehouse implementations. Thus, we present a CASE tool designed to generate the SQL queries necessary to build a warehouse from a set of operational relational databases. The warehouse designer simply specifies a list of attribute names that will appear in the warehouse, conditions if any are desired, and a description of the operational databases. The tool returns the queries needed to populate the warehouse table. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Data warehouses; Metadata; SQL queries

1. Introduction

Organizations today record and track every single detail of their transaction history hoping that valuable business insights can be unearthed. With millions of transactions captured in their systems, organizations are experiencing an explosion of data. Early attempts to use the burgeoning transaction database (now commonly called the online transaction processing system or OLTP) for decision support has been a forgettable mistake for many organizations. Recent interest in data extraction and on line analytical processing of extracted and aggregated data has given rise to a new creed of information system, namely the data warehouse system [17]. Data warehouses now appear to offer the much heralded database support for decision support.

Several different definitions have been used to describe what a data warehouse is [3,8,9,14,17]. McFadden and Watson [14] define a data warehouse to be a collection of integrated, subject-oriented databases, designed to support decision enhancing activities, where each unit of data is relevant to some moment in time. The data in the warehouse is not updated in real time, but refreshed periodically. Business managers, who use the warehouse data for business analysis, access the data for querying purposes only. Since the warehouse database is physically segregated from the operational databases, and is a collection of summaries and details, it provides a faster forum to answer the ad hoc information requirements of decision makers.

Modeling a data warehouse is one of the most critical component in the warehouse design process [31]. The process must include logical data modeling, physical data modeling, and metadata management. Though similarities might exist between database modeling and data warehouse modeling, there also exist significant differences. A data warehouse model must satisfy rapid information retrieval and ad hoc query formulations. Star schema and dimensional schema are often employed here. In addition to the logical data model the warehouse design must maintain a metadatabase consisting of information that will aid in administering the warehouse. Not only can metadata enable the design of data warehouses it also supports the user interface to the warehouse [13]. Design tools that utilize metadata to build warehouse designs are a natural evolution of the warehouse design process. In this paper we present the conceptual foundations and implementation details of such a tool. The tool makes use of the metadata for a set of relational databases and returns a set of SQL queries that define the data needed from the data sources to create the desired warehouse table.

In the next section we briefly look at the data warehouse model. Section 3 examines the tool implementation. The conceptual framework of the tool is given in Section 4 and Section 5 looks at the basic issues in partitioning queries.
2. The warehouse model

McFadden and Watson [14] present one of the earliest architectures for data warehouse design. In their model, the metadata or data about the underlying operational data is at the base of the model. All warehouses are built on this base information. At the highest level of the architecture user interfaces and its interaction with the warehouse model is ascribed.

The data warehousing industry has, however, seen several different architectures. Among the most popular is perhaps the one by Kimball et al. [30]. The model is shown in Fig. 1.

The model shows two major components, namely, the back room and the front room technologies. The back room area focuses on the data sources and staging of the relevant data prior to extraction, transformation and loading (ETL). The flow of data from source files to the warehouse tables is driven by the metadata kept in the metadata catalog. This process is where much attention is paid by a large number of vendors. Ensuring a seamless and correct set of procedures to ensure the ETL is key to the success of many warehouse implementations.

The front room component is concerned with user access to the data in the warehouse. Once again several vendors have developed a number of solutions to enable the easy access of the warehouse data.

2.1. Warehouse design

Major data warehouse vendors such as Oracle, IBM, Microsoft, and Informatica have introduced systems that help design and build warehouses. These systems enable the professional developer to assemble the data sources and carry out the ETL tasks in a systematic fashion. Though these systems do use metadata catalogs and graphical navigation and selection tools, the developer is expected to have an intimate knowledge of the underlying structure of the source databases or files. The proposed system on the other hand, requires the designer to only have a cursory knowledge of the structure of the data sources. This is accomplished by allowing more information in the metadata. For example, a typical vendor tool makes use of only the key dependencies, while the proposed system can incorporate any available functional dependencies and also makes use of the universal join dependency. The incorporation of these dependencies allows us to test for lossless joins even in cases where joins are not done over keys.

Designing and building a data warehouse, given the need to understand the corporate data sources and their structure, is a challenging task. Vendors as well as customers of data warehouses continue to uncover new approaches and tools to design and model their warehouse requirements. It is in this spirit that we present our design tool, which is built on grounded database theory. Unlike other tools or methods, our design tool, which is theory based, enables us to prove the validity and accuracy of the relational queries that may be used to design the resulting data warehouse.

A great deal of research on warehouses is focused on maintaining materialized views. The Stanford Data Warehouse Project (WHIPS) [8,25] has primarily looked at methods to update a relational data warehouse given that the operational databases are relational. Quass et al. [19], Wiener et al. [26], and Zhuge et al. [27] investigate algorithms for such view maintenance. Gupta [6] has recently studied the selection of views to materialize. Similarly, Baekgaard and Roussopoulos [1] at the University of Maryland have explored data warehouse view refreshment.

If recent announcements in the trade press and practitioner conferences are any indication, there is a growing need for tools that will facilitate the design and development

![Fig. 1. High level technical architecture for data warehouse (adapted from Kimball et al. [30]).](image)
of warehouses and data marts more than any other effort [28]. History of systems development has shown that the use of structured methodologies and software tools (CASE) has a positive impact on system success and maintainability [29]. Borrowing from this experience it can be argued that similar results can be expected in the area of warehouse development also if similar support is available. Unfortunately, there are no such CASE tools that a warehouse designer can use today. In this paper we present the conceptual development and the implementation of a CASE tool designed to generate the SQL queries necessary to build a warehouse from a set of operational relational databases. The warehouse designer simply specifies a list of attribute names that will appear in the warehouse, conditions if any are desired, and a description of the operational databases. The tool returns the query needed to build the warehouse table.

In the next section we look at the interface of the warehouse modeler. The following section will describe the interface of our CASE tool. The foundations and a current implementation of the warehouse modeler are examined in Section 4 and we conclude by looking at the issues associated with query partitioning.

3. Interface design and implementation

3.1. Introduction

In this section, we introduce the design and the implementation of the user interface of the data warehouse modeler system. We present the interface of our prototype before the foundations of our approach to set the stage for the type of information that the modeler has to work with. In particular, our system is designed to work with a very simple view of the source data. The interface only shows a list of attribute names to represent the data available in the data sources. In the event that names conflict, either views renaming the attributes can be used or the names can be qualified. The metadata (e.g. functional dependencies, tables, etc.,) is assumed to have been supplied by the information manager or developed through simple extraction programs. The specific types and format of the metadata is not important in the context of our presentation. Nevertheless, the type and format are of concern to any data warehouse administrator because there is a need for compatibility among the different database and data warehouse vendors’ products that the organization might use. To this effect, emerging standards in the area of metadata interchange will help. In Section 3.2, we present a general
3.2. Interface design

The DWQTI is a user-friendly interface for the data warehouse modeler presented in this paper. The layout is shown in Fig. 2. We briefly look at the main parts of the interface to help set up the discussions of the modeler.

3.2.1. Operational DATABASE attributes

After typing in the operational metadata file name in the small text field labeled with “Operational MetaData File:”, a scrolling list of all attribute names in the given operational metadata file will be generated. This list will not be editable and will be used for selecting warehouse attributes and composing warehouse conditions. The default operational metadata file name is “demo” in the current prototype. If the typed in operational metadata file name does not exist in the data files, error information will be presented in this region.

3.2.2. Warehouse attributes

This is the region for writing a proposed list of warehouse attributes corresponding to the SELECT part of an operational database query in SQL format. All the attributes listed in this region must be from the Operational DATABASE Attributes list. Clicking on an attribute in the list of operational database attributes will place it in the list of warehouse attributes in this region, and double clicking on an attribute in the list of operational database attributes will remove it from the list of warehouse attributes. Clicking on the button “Reset” in this region can clear the selected list of warehouse attributes.

3.2.3. Warehouse conditions

This region is an editable text area for users to compose a warehouse condition. We assume that a warehouse condition is of the form C1 & C2 & … & Cm, where each Ci is a simple condition or the union of a few simple conditions. By a simple condition we mean an expression of the form ‘attribute’ op ‘attribute or constant’, where op is one of the six comparison operators: =, >, <, >=, <=, and !=. For example,

\[(A = “Ames”)&(B <=120|B > 500)&(C E)\]

is a legal warehouse condition under our assumption. Users can check if some warehouse condition composed in this text area is legal by clicking on the button labeled with “Legal Conditions?”. The small text field under this button will display either “Yes!” or some error message.

Any attribute in a warehouse condition can be either typed in or placed by first clicking on the relevant attribute in the list of operational database attributes and then clicking on the button labeled with “Get One Attribute”. Clicking on some operator in the list of Operators will place it to the current cursor position in this text area. All constants (either real numbers or strings) should be typed in. Clicking on the button labeled with “Clear” can clear the text area. Note that some attributes in a warehouse condition may not appear in the list of Warehouse Attributes, but must have appeared in Operational DATABASE Attributes.

3.2.4. Operational DATABASE query

This is a non-editable text area for displaying the operational database query in SQL. After clicking on the button labeled with “Get Operational DATABASE Query”, users will see either the desired operational database query in SQL format (i.e. SELECT, FROM, WHERE) corresponding to the given warehouse attributes and conditions, or the information of “A lossy join is made!” together with the computed optimal join sequence for user’s reference purpose. The join sequence can then be used to manually derive an SQL. The issue is that the user will understand that the join as given will need to be considered lossy and also the design of the warehouse will need to be reconsidered or the join sequence will have to be expanded.

3.3. Interface implementation

3.3.1. General description

The DWQTI is a user-friendly interface for the data warehouse modeler. It is implemented in JAVA (JDK 1.0.2). The reason for not using JDK 1.1.x is that some computer systems do not support JDK 1.1.x at this moment. This JAVA program is a standalone application rather than an applet because it needs to read some data files about the database information from the user’s machine. That is, it is implemented as a public class (named DWQTI) which extends the JAVA AWT class Frame.

The class DWQTI consists of the main( ) method, fourteen
new-designed methods, and additional methods inherited from the AWT class Frame. Also, seven utility classes and six data structure classes are designed and implemented to support the fourteen methods. Table 1 shows the classes used in the JAVA program.

3.4. Example

We briefly look at an example of using the modeler interface.

Example 1:

Operational MetaData File: demo
Warehouse Attributes: dname, ename, iname
Warehouse Condition: inum = 12 & mbirth \leq \textit{9/9/54} \text{ OR } mbirth \geq \textit{10/22/60} \text{ AND } ocity = \text{"Ames"}

Modeler Database Query:

\[
\text{SELECT dept.dname, employee.ename, item.iname FROM owner, dept, employee, manager, store_info, sale, item WHERE sale.inum = 12 AND (manager.mbirth \leq \textit{9/9/54} \text{ OR } manager.mbirth \geq \textit{10/22/60}) AND owner.ocity = \text{"Ames" AND owner.storenum = dept.storenum AND owner.onum = store_info.onum AND owner.storenum = store_info.storenum AND owner.storenum = sale.storenum AND dept.mnum = employee.mnum AND dept.mnum = manager.mnum AND dept.dnum = sale.dnum AND employee.enum = sale.enum AND sale.inum = item.inum}
\]

Fig. 2 shows the information required from the user. The user has chosen lname, ename and iname from the list given in Operational DB attributes. The attributes are selected by scrolling down the list and clicking on the desired attribute. The warehouse conditions are created by either typing the condition or using a combination of typing and clicking on the operators and/or Operational DB attribute names. The metadata for this example consists of the relation schemes in the data sources and the set of functional dependencies defined over the union of the set of relation schemes.

In the example the “yes!” response comes from clicking on the legal condition button after the condition was formulated. When the required information (Fig. 2) has been entered, the user can get the query needed to generate the warehouse tables (the Operational DB query) by simply clicking on the “Get Operational DB Query” button. The resulting screen is shown in Fig. 3.
4. Modeler foundations and implementation

4.1. Introduction

In this section, we discuss the foundation and the implementation issues of the modeler or engine for generating the data warehouse query.

In Section 4.2, we describe the modeler design issues which include the approaches and/or algorithms for processing query translations, creating complete intersection graphs (CIGs), generating adjusted breadth first search (ABFS) trees, finding join sequences, and testing the losslessness of the join sequences.

In Section 4.3, we introduce the modeler implementation issues. The functionality and structures of two utility classes and four data structure classes will be introduced.

4.2. Modeler foundations

4.2.1. Query translation process

To create a warehouse query, we must translate the request for the warehouse design into our hypergraph space. The resulting source (warehouse) query must then be mapped to the target data space (the hypergraph representing the collection of connected operational databases). Finally, the target query hypergraph is mapped to an SQL query. To look at this process in more detail, we consider the basic data structures and algorithms. To start the discussion, we briefly look at the notion of a hypergraph and its complete intersection graph.

4.2.2. Hypergraph

A hypergraph is a couple $H = (N, E)$, where $N$ is a set of vertices and $E$ is a set of edges which are non-empty subsets of $N$.

4.2.3. Complete intersection graph

Let $H = (U, R)$ be a hypergraph where $U = \{A_1, A_2, \ldots, A_n\}$ is a set of attributes and $R = \{R_1, R_2, \ldots, R_p\}$ is a set of relation schemes over $U$. Fig. 4 shows an example of a hypergraph. The CIG [10,18] is an undirected graph $(R, E)$ where $E = \{(R_i, R_j) : R_i \cap R_j \neq \emptyset, R_i \in R, R_j \in R, i \neq j\}$.

Note that the edge $(R_i, R_j)$ between vertices (or nodes) $R_i$ and $R_j$ exists if and only if $R_i$ and $R_j$ have at least one attribute in common. The edge $(R_i, R_j)$ will be labeled as $R_{ij}$ where $R_{ij} = R_i \cap R_j$. The complete intersection graph for the hypergraph shown in Fig. 4 is given in Fig. 5.

4.2.4. Adjusted breadth first search

The adjusted breadth first search (ABFS) [10,18] is a variation of the breadth first search (BFS) used to determine the join sequence for a target hypergraph. An ABFS supplements the traditional BFS by including a path label for each node and an adjustment set in the search tree so that the search is more efficient. The resulting search tree is called an ABFS tree [18]. The node from which the search is started is called the root of the ABFS tree.

The path label for an ABFS tree node is the union of all warehouse attributes on this ABFS tree node and its ancestors on the search path. So the path label of an ABFS tree node should be a superset of its parent’s path label. In the process of creating an ABFS tree, the path labels will be used to prune or delay the expansion of subsets where the unused nodes that are adjacent to the current endpoint of the search path do not contribute any new warehouse attributes to the path label. Any nodes falling into this class will be stored in the adjustment set [18] (denoted by ASet) with a pointer to the position where they could be added to the ABFS tree during the further search or expansion. The relevant CIG can be applied to determine which nodes are adjacent to the current endpoint of the search path.

The expansion of the ABFS tree will continue until the union of the path labels of all the leaves in the current ABFS tree contains all of the warehouse attributes. If the ABFS tree cannot be expanded and the union of the path labels of all the leaves in the current ABFS tree does not contain all the warehouse attributes, then a node can be taken from the adjustment set and the process can be restarted from the position pointed to by this node. Note that this process of creating an ABFS tree will terminate successfully in a finite number of steps since all the warehouse attributes are in the hypergraph and can be reached eventually. An ABFS tree...
for the CIG shown in Fig. 5 is given in Fig. 6 where the set of warehouse attributes \{B,E,F\} and ABC is the root of the ABFS tree. Since all of the warehouse attributes appear in the label for AEF, the required join sequence is defined by the single path ABC–AEF. The node in the adjustment set (BF) is not needed for this simple example and is ignored by the process. It is only required in the event that the search process fails to include all warehouse attributes in the path labels in the ABFS tree. The root is chosen in our approach by choosing the scheme that leads to the most compact join sequence.

Several different ABFS trees with the same root may be generated. This is because the order of search is not unique. Also, there are more than one way (such as FIFO, LIFO, or randomly) to select nodes from the adjustment set. In Fig. 6 BF is placed in the adjustment set due to the fact that CDF was examined first.

4.2.5. Join sequence

Finding an optimal join sequence for the selected warehouse attributes (including the attributes appeared in the warehouse condition) is a crucial part in the modeler design and implementation.

Once the ABFS tree with a given root is created, we can determine the join sequence defined by this tree. The approach is to select a set of paths connected to the root such that the union of the path labels contains all of the desired warehouse attributes. We use the following procedure to select the appropriate paths [18]:

(Step 1.0) Set \( W \) = the set of warehouse attributes. Go to (Step 1.1).

(Step 1.1) Mark every leaf and its ancestors if its path label has a warehouse attribute that appears only once in the path labels of all leaves in the ABFS tree. Remove the warehouse attributes included in the path labels of the marked nodes from \( W \). If \( W \) is empty, stop; otherwise, go to (Step 1.2).

(Step 1.2) If there is a contributing warehouse attribute in more than one path labels of the unmarked leaves with the same parent, then mark one (and only one) of those leaves and its ancestors. Remove the warehouse attributes in the path labels of the marked nodes from \( W \). If \( W \) is empty, stop; otherwise, go to (Step 1.3). (By a contributing warehouse attribute we mean a warehouse attribute that occurs in the path label of a leaf but does not occur in its parent’s path label.)

(Step 1.3) If there is a leaf which contains a remaining warehouse attribute in \( W \) with the lowest frequency, then mark this leaf and its ancestors. In case of tie, choose the leaf with the shortest path and mark the nodes on this path.

The approach described in the previous section does not guarantee to create the optimal ABFS tree with a given root since the order of search in that approach is not necessarily optimal. The so-called optimal ABFS tree with a given root is actually the one with the minimum weight over all possible ABFS trees for this root. By weight of an ABFS tree we mean the length of the join sequence defined by the ABFS tree.

The creation of a non-optimal ABFS tree with some given root does not cause serious problems. Our goal is to generate an optimal join sequence which is the one with the minimum weight over the optimal ABFS trees for all roots. The probability of creating non-optimal ABFS trees for all roots is very low. One can remove a redundant join from the resulting join sequence at a later stage (e.g. just prior to query optimization).

The algorithm to find a join sequence for a given hypergraph and a set of warehouse attributes is summarized as follows:

(Step 2.0) Create the CIG for the hypergraph. Find the set \( L_R \) of all legal roots in the CIG. Set \( \text{minweight} = \text{the number of nodes in CIG} \). Go to (Step 2.1).

(Step 2.1) If \( L_R \) is empty, then stop. Otherwise, choose a root \( r \in L_R \), set \( L_R := L_R - \{r\} \), and go to (Step 2.2).

(Step 2.2) Create an ABFS tree with root \( r \). Find the weight and the corresponding join sequence for this tree. If the weight is smaller than \( \text{minweight} \), then save this join sequence as the current best one, and replace \( \text{minweight} \) with the weight. Go to (Step 2.1).

4.2.6. Lossless test

After a join sequence for the target hypergraph is found, the modeler goes to the next crucial part — lossless test of the join sequence which can be considered as a subhypergraph in the target hypergraph. The lossless test of a join sequence is basically to determine if a subhypergraph defines an embedded join dependency (EJD), and the whole procedure can be divided into two stages.

A hinge always defines an EJD and the converse is also
true except in two trivial cases [16]. In the absence of functional dependencies, testing if a subhypergraph defines an EJD basically reduces to testing for a hinge. Furthermore, since a connected subhypergraph is a hinge if and only if it is not externally connected, the test can be reduced to check if the subhypergraph has a separating edge for each component connected with respect to this subhypergraph. The detailed procedure, based on Sciore’s axioms [20], for performing such a test in the first stage is listed as follows:

(Step 3.0) Suppose the join sequence is represented by the subhypergraph \((U', R')\) of the target hypergraph \((U, R)\). The set \(W\) of warehouse attributes is contained in \(U'\). Go to (Step 3.1).

(Step 3.1) Create the attribute sets (as new hyperedges) \(G_1, G_2, \ldots, G_m\) by uniorning the hyperedges connected outside of \(R'\). Go to (Step 3.2).

(Step 3.2) For each \(i = 0, \ldots, m\) put \(G_i := G_i - (U - U')\). (That is, eliminate the attributes that are not in \(U'\) from each \(G_i\).) If all \(G_i's\) are empty, then the join sequence is lossless, and stop; otherwise, go to (Step 3.3).

(Step 3.3) Without loss of generality, suppose \(G_1, \ldots, G_p\) are non-empty where \(1 \leq p \leq m\). For each \(k = 1, \ldots, p\), if \(G_k\) is a subset of some element in \(R'\), then set \(G_k\) to be empty. If all \(G_i's\) are empty, then the join sequence is lossless, and stop; otherwise, the join sequence is lossy in the absence of FDs, and send the non-empty \(G_i's\) to the second stage for further testing when FDs are available.

In the second stage, we first create a tableau representing the subhypergraph \((U', R')\). Suppose there exists a set \(F\) of functional dependencies embodied in the hyperedges of the subhypergraph \((U', R')\). Then we apply the FDs in \(F\) to the tableau by adopting the Chase method [11,22], and let \(T^*\) be the resulting tableau. It is shown in Ref. [16] that under above assumption, the join sequence represented by \((U', R')\) is lossless if and only if \(T^*\) contains at least one row with all the distinguished variables for the attributes in each separating set \(G_i\) \((i \in \{1, 2, \ldots, q\}, 1 \leq q \leq p)\) sent from the first stage. Based on this result, we use the following procedure to finish the second stage of the lossless test:

(Step 4.0) Create the initial tableau \(T\) representing the subhypergraph \((U', R')\). Apply the FDs repeatedly to \(T\) by following the Chase method. Let \(T^*\) be the resulting tableau. Let \(G_1, \ldots, G_p\) be the separating sets sent from the first stage. Put the counter \(k = 1\). Go to (Step 4.1).

(Step 4.1) If \(k = q\), go to (Step 4.2). Otherwise, the join sequence is lossless, stop.

(Step 4.2) Examine the rows in \(T^*\) for distinguished variables for the attributes in \(G_i\). If there does not exist a row containing all the distinguished variables for the attributes in \(G_i\), the join sequence is lossy, stop. Otherwise, increment \(k\) by 1 and go to (Step 4.1).

Any lossless test is based on the universal relation assumption. A critical aspect of this assumption is that the attributes in the operational database tables that are equivalent have the same names [22]. This is easily remedied by employing views at the operational database sites to rename any offending attributes.

### 4.3. Modeler implementation

#### 4.3.1. General description

As mentioned in Section 3.3, the implementation of this modeler or engine part for the DWQTI Java program includes two new utility classes and four new data structure classes. The functionality and relations of these six classes and their methods are shown in Tables 2 and 3.

The implementation of this modeler also extensively uses the following utility classes and data structure classes introduced in Section 3.3: class SortDelVector, class CharSet, class RelationOfTwoSets, and class RelNode.

#### 4.4. Validation

To address the correctness of the process used in the design tool it is necessary to look at the correctness of the query that is generated.

**Theorem 1.** The query generated by the design tool is correct.
Proof. The list of attributes and the selection conditions are taken directly from the user interface, thus the only issue is that the join sequence used in the query is lossless. The design tool only generates a query when the test for losslessness of the join sequence returns true. The test for losslessness used by the design tool has been proven correct in Ref. [16]. Therefore, the general query is correct. □

5. Query partitioning

The modeler query is used directly when the warehouse table is drawn exclusively from a single database. A more interesting case comes when more than one operational database is used to generate a warehouse table. The question of interest is what query should be generated. It is easy to show that the best approach (from the point of view of generating a lossless join design) is to allow the modeler to operate on the connected hypergraph making up the warehouse data space as though there is a single database and then partition the query. For example, in Fig. 7 we have two databases that are required to create a warehouse table consisting of the attributes C, D, and F. The hypergraph used by the modeler is given in Fig. 8.

**Database 1:** \{r1(AB), r2(BC), r3(CD)\}
**Database 2:** \{s1(DE), s2(EF), s3(FG)\}

If the designer wants to impose the condition B > F, then the modeler would generate the following query.

SELECT C, D, F FROM r2, r3, s1, s2 WHERE r2.C \^ r3.C AND r3.D \^ s1.D AND s2.E and r2.B > s2.F

The resulting query defines a lossless join, but cannot be directly applied to the existing operational databases. We need to partition the query to be able to make use of it. The rule for partitioning the query is to create one new query for each operational database, where the attribute list consists of all of the attributes that are mentioned in the modeler query and appear in the given database. Let X1, ..., Xm be the attribute sets for the m operational databases in the connected hypergraph used by the modeler. Let attr(Q) be the set of attribute names mentioned in the query Q generated by the modeler. Let Ri be the set of relations for the ith operational database and rel(Q) be the relation names mentioned in the query Q. Then we can characterize the query for the ith individual operational database as

SELECT Xi \cap attr(Q) FROM Ri \cap rel(Q) WHERE f

and f is the portion of the query Q condition that relates to the ith operational database. The resulting queries can then be processed on the m operational databases resulting in m relations y1, y2, ..., ym. The operational database queries for the example in Fig. 7 are shown in Fig. 9. To generate the warehouse query requires the use of substitution of relation names and a reduction in the size of the condition. The final warehouse query for combining the m relations to satisfy Q is simply

SELECT W FROM y1, y2, ..., ym WHERE f1

where W is the set of attributes in the warehouse table and f1 is the condition resulting from the following process.

1. remove any subconditions used in the operational database queries.
2. replace any relation name r from a given operational database with the name of the relation generated by the operational database query for that operational database.

The resulting query for the example in Fig. 7 is shown in Fig. 10.

SELECT B, C, D FROM r2, r3 WHERE r2.C = r3.C
SELECT D, E, F FROM s1, s2 WHERE s1.E = s2.E

Fig. 7. Operational database schemes and their hypergraphs.

Fig. 8. The connected hypergraph for Fig. 7.

Fig. 9. Operational queries generating the relations y1 and y2, respectively, from the example in Fig. 7.
SELECT C, D, F
FROM y1, y2
WHERE y1,D = y2,D AND y1,B > y2,F

Fig. 10. The query needed to generate the warehouse table from the results of the operational queries.

The result is that it is straightforward to partition the query generated by the modeler for application to the operational databases. It is also easy to prove that if Q defines a lossless join, then the final warehouse query (e.g. Fig. 10) will also produce a lossless join.

6. Conclusions

The Data Warehouse Query Translation Interface (DWQTI) has been implemented in JAVA (JDK 1.0.2). It provides a user-friendly interface and an efficient modeler to let the users obtain the desired operational database query information in SQL after they select the warehouse attributes and compose the warehouse condition on the interface. It is also shown that the resulting query can be easily partitioned for multiple operational databases. A significant contribution of our work is that using metadata (attributes, functional dependencies, and join dependencies) specified in plain ASCII format that a data warehouse administrator can quickly use to design a warehouse table and generate the required set of relational queries that are provably correct! With growing interest in data marts our design tool would become a powerful aid for the warehouse administrator. Future implementations will address heterogeneous data sources such as legacy file systems and non-relational databases. The future work also looks at methods to directly construct the warehouse tables and incorporate data quality issues.

This JAVA program has been run and tested across a few platforms including UNIX (in DEC Alpha workstations and HP-UX workstations), Window 95, and Window NT 4.0. It runs well over these platforms on all test data. The data structure classes and utility classes in this program are well designed so that the implementation is relatively simple and efficient.

References


Further reading

