Uniform Management Of Data And Metadata

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

Yannis Velegrakis
University of Trento
velgias@disi.unitn.eu

ABSTRACT

There is a growing need to associate a variety of metadata with the underlying data, but a simple, elegant approach to uniformly model and query both the data and the metadata has been elusive. In this paper, we argue that (1) the relational model augmented with queries as data values is a natural way to uniformly model data, arbitrary metadata and their associations, and (2) relational queries with a join mechanism augmented to permit matching of query result relations, instead of only atomic values, is an elegant way to uniformly query across data and metadata. We describe the architecture of a system we have prototyped for this purpose, demonstrate the generality of our approach and evaluate the performance of the system, in comparison with previous proposals for metadata management.

Categories and Subject Descriptors:
H.2.1 [Database Management]: Logical Design – Data Models

General Terms: Management.

Keywords: Queries as data, metadata management, intensional associations, annotations.

1. INTRODUCTION

In recent years we have witnessed a tremendous proliferation of databases in many fields of endeavor, ranging from corporate environments and scientific domains to supporting a diverse set of applications on the web. These databases are becoming increasingly complex, both in their internal structure (e.g., thousands of tables) and in their interactions with other databases and applications (e.g., mediators and workflows). There is a consequent need for understanding, maintaining, querying, integrating and evolving these databases. In successfully performing these tasks, metadata plays an important role. Metadata is data about data, a secondary piece of information that is separate in some way from the primary piece of information to which it refers. Metadata examples include schema, integrity constraints, comments about the data [4], ontologies [1], quality parameters [28, 16], annotations [5, 12], provenance [24], and security policies [3].

Metadata is used in many different fields. In corporate environments, databases deployed at the core of important business operations may contain erroneous, inaccurate, out-of-date, or incomplete data with a significant impact on query results [8]. In such applications, data can be tagged with quality parameters to communicate suitability, accuracy, freshness or redundancy [27, 28], and schema structures can be annotated with textual description to communicate their semantics. In scientific domains where data may be collected from various sources, cleansed, integrated and processed to produce new forms of data enhanced with new analysis results [19], provenance can be provided as metadata in the form of annotations [4] and schema and mapping information can be stored in special structures to allow users to apply their own judgment to assess credibility of query results [24]. In heterogeneous environments where different sources may use different structures to represent the same real world entity, or the same representation to model different concepts, metadata can clarify semantics, prevent misinterpretations or misuses of data, achieve interoperability [10] and allow the retrieval of the data that is best suited to the task at hand [21]. In the Internet domain, superimposed information, i.e., data "placed over" existing information sources, is used to help organize, access, connect and reuse information elements in those sources [15]. Similarly, security related policies can be associated to the data to control access in various environments [3].

Kind	Ref.	Method Used
Annotations	[12]	Atomic value annotations attached to a block of values within a tuple. They accompany the values as retrieved. Relational algebra query language.
Provenance	[5]	Atomic data values carry their provenance, which propagates with them as they are retrieved. Query language supports predicates on provenance.
Quality Parameters	[16]	Data values are associated with quality parameters (accuracy, freshness, etc.). SQL is extended to retrieve data using these parameters.
Schema & Mapping Information	[24] [14] [29]	Explicit modeling of schema and mapping information, and associations of it with portions of the data. SQL extension to retrieve data and metadata that satisfy certain metadata properties.
Security	[3]	Credential-based access control. System reads complex security profiles and returns data results accordingly.
Superimposed Information	[15]	Loosely-coupled model of information elements, marks and links used to represent superimposed information. It has no specific schema, but is in relational model and can be queried using SQL.
Time	[6]	Creation and modification time is recorded with the data and used in query answering. Query language supports predicates on time values.

Table 1: Metadata management proposals

1.1 Metadata Management Approaches

Over the years, numerous proposals have been made by researchers for augmenting the data model and the query capabilities of a database in order to facilitate metadata management. Table 1 provides a list of such proposals. While the list is by no means exhaustive, it provides a good sample of the kinds of metadata that have been considered of interest and have been studied. It also characterizes the way the problem has been approached. Based on these approaches one can observe the great variety of metadata, the different structures, the kind of data associated with metadata, and the way metadata has been used in queries. Some of this metadata is described by single atomic values, e.g., the creation time of an element in the database [6]. Others have a more complex structure, like the schema mapping information [24] or security [3]. Furthermore, metadata can be associated either to individual data values [5, 16, 6] or to a group of values, such as a subset of the attributes (i.e., a block) of a tuple [12] or a complex XML element [3].

A common denominator of the approaches in Table 1 is the use of metadata in querying. Some use it to restrict query results, which may [6, 16] or may not [5] include metadata alongside the actual data results. Others query and retrieve metadata independently of the data to which it is associated [15].

It is also worth observing that each entry in Table 1 is tailored to specific kinds of metadata, and is not directly applicable to other kinds, at least not without some major modifications. Past attempts at building generic metadata stores (e.g., [13, 2]) have employed complex modeling tools for this purpose: [13] explicitly represented the various artifacts using Telos [17], and [2] employed data repositories (intended for shared databases of engineering artifacts). A simple, elegant approach to uniformly model and query data, arbitrary metadata and their association has been elusive.

Another observation from Table 1 is that metadata has to be explicitly associated with each data item it refers to. There are, however, practical scenarios in which an *intensional* association may be more appropriate. For example, in an application that contains reviews about restaurants [4], one may need to describe a property that holds for all the restaurants in New York. Using previous proposals this information would have had to be associated explicitly to every New York restaurant in the database. Furthermore, future tuples cannot be accommodated without explicit association. For instance, if ten new restaurants are opened in New York, the property that holds for all the restaurants in New York should also hold for them. However, this cannot be achieved unless the property gets explicitly associated to these ten restaurants. An alternative approach (which we advocate) is to use an intensional description of the restaurants (data) that have that property (metadata), in the same way virtual views use queries over base tables to describe their instances, and future tuples that are inserted or deleted from the base tables, are automatically included or removed from the instances of the views.

1.2 Our Approach

In this work, we describe a study that aims to accommodate in one simple framework the different kinds of metadata, the different structures, the different ways that metadata is associated with data and the different ways in which it is used in queries. We argue that in order to achieve this we need a framework that is simple and abstracted from the specifics of each kind of metadata. Having observed that the operations one needs to perform on metadata are similar to those people do with data, we propose the use of standard data management techniques for metadata, so that both data and metadata can be managed in one single framework. We advocate that the relational model is adequate for such a purpose. Metadata with complex structures can easily be modeled through relations and attributes. These relations have no special semantics, thus, the same piece of information can be viewed either as data or as metadata. It can also be queried using a relational query language, even independently of whether or not it is associated to some data.

Our philosophy is that although, at the conceptual level, there may be a distinction between data and metadata, at the database level, for the purpose of management, everything is represented as a relation. This approach is not different from the one followed by the relational model, where at the conceptual level there may be a distinction between entities and relationships, but at the database level, everything is represented through relations.

The main mechanism used in the relational model to associate data in different relations is the join on one or more attributes. If data and metadata have been modeled as relations, then the same mechanism can be used to describe the association between data and metadata. Unfortunately, the relational join operation has two main limitations that make it inadequate for this intended use: (i) The association is always at the tuple level, i.e., it is not possible to associate a metadata tuple with only a subset of attributes of another tuple in a different table, since there is no way to specify the attributes that the metadata refers to; (ii) It requires explicit association between the tuples through the join attributes, as previously mentioned in the New York restaurants example. To cope with these issues, we propose the use of queries as values in the relational tables. In particular, we show how attributes of type "query", can achieve the required functionality.

Using queries as data values is not entirely new. Relational DBMSs already store in the catalog tables the definition queries of their views. In this case, however, queries are considered schema information and despite the fact that they can be queried using SQL, they are not considered part of the instance data. Our proposal raises such metadata to the level of data, and provides a unified mechanism for modeling and querying across data and metadata. Apart from the system catalog tables, queries as values have been proposed in INGRES [23], with a similar functionality adopted by Oracle [11]. They have also been studied in the context of relational algebra [18], and the Meta-SQL system [9]. Here we show how this idea can be used in the service of metadata management. A key difference is that previous approaches require the existence of an $eval$ operator that evaluates the queries stored as values at run time, possibly resulting in nested relations, or in some cases the computation of the outer-union of these results. In contrast, we only need a new kind of predicate that makes use of $eval$, leading to a more efficient implementation. Our use of queries as values is similar to the role of RDF resource descriptions [26], but our approach is much more generic, since we can use as a resource description the full power of SQL queries.

Our contributions can be summarized as follows:

1. We elevate metadata to first class citizens of the database and the query language, without requiring any special semantics. The approach allows metadata management without any modification of the semantics of the relational model and SQL, and without having to alter existing tables, since we use stored queries to refer to the data which the metadata entries describe.

2. We extend the traditional join mechanism of the relational model to support joins that are based not on single values, but on a relation specified by a query stored in one of the attributes. This allows intensional specification of the data to which the metadata is associated. Furthermore, it allows metadata tuples to be associated to not just whole tuples, but also to portions of them.

Customers

Name	Type	Loc	PhoneLine	CircuitID
AFLAC	bus	NJ	4078417332	245-6983
J. Lu	res	NY	2019394460	245-7363
H. Ford	res	NJ	2159537607	245-7564
AMEX	bus	NY	3178763540	343-5002
NJC	bus	NJ	9730918327	981-5002
BCT	bus	NJ	9734858504	273-6019
...	...	...	...	...

Provenance

Rf1	Source	IP	Protocol
q1	NJDB	147.52.7.8	http
q2	3State	148.62.1.11	ftp
...	...	...	...

Permissions

Rf2	Users
q11	Administrators
q12	Guests
...	...

q1: select Name,Type,PhoneLine
from Customers where Loc='NJ'
q2: select Loc,PhoneLine,CircuitID
from Customers where Type='business'

q11: select * from Provenance
where IP LIKE '147.%'
q12: select Name from Customers
where Loc='NY'

XJ

Qc	Qt
q_{C1}	q_{T1}
q_{C2}	q_{T2}
q_{C3}	q_{T3}
...	...

Technicians

Name	Contact	Company
W. Farkas	4804978353	AT&T
S. Gilbert	3178757627	Verizon
M. Henry	8187167852	AT&T
C. Urs	7739735713	AT&T
Y. James	7344676191	CISCO

q_{C1}. select CircuitID from Customers where Type='residence'
q_{T1}. select * from Technicians where Company='CISCO'
q_{C2}. select PhoneLine,CircuitID from Customers where Type='business'
q_{T2}. select * from Technicians where Company='Verizon'
q_{C3}. select PhoneLine from Customers
q_{T3}. select * from Technicians where Company='AT&T'

Figure 1: A database with metadata information stored in regular tables as data.

3. We explore alternative implementation mechanisms that allow the use of queries as data values in modern relational databases and also allow joins based on such values. We present pure rewriting-based strategies, as well as techniques that can effectively use and update indexes for this purpose.
4. We describe the architecture of the Metadata Management System (MMS) we have prototyped. We experimentally evaluate the performance of MMS, and compare it with previous proposals for metadata management. Our results validate the generality and practicality of our uniform approach to metadata management.

The structure of the paper is as follows. Section 2 provides a running example that identifies the issues and illustrates our solution. Section 3 defines the semantics of query expressions as data values in relational tables, and their use in query conditions. Section 4 describes how attributes of this type can be implemented using existing relational database technology. Finally, Section 5 presents experimental results to validate our methodology.

2. ILLUSTRATIVE EXAMPLE

We describe in this section a realistic example that illustrates the need for a uniform way of managing different kinds of metadata and their associations to data, and also illustrates our solution.

EXAMPLE 2.1. *Consider a communications company database with the table* Customers *shown in Figure 1. The table contains information about the phone lines (*PhoneLine*) of the customers (*Name*), their location (*Loc*), whether a customer is a business or a residence (*Type*) and the circuit (*CircuitID*) used by the phone line. The contents of the table are generated by integrating data from a number of physically distributed sources. When a mistake is detected in the table, it is important to know its origin in order to correct it. To make this information available to the user, the data in the* Customers *table needs to be annotated with its provenance information. This includes the origin database name (*Source*), its IP address (*IP*), and the communication protocol used to access it (*Protocol*). One way to achieve this is to alter the table* Customers *by adding three new columns for each of its attributes [4]. Such a solution may affect the way existing applications use the table, may degrade performance, or may not even be implementable due to lack of authorization for such a change.*

*An alternative solution is to to store the provenance information in a separate table (*Provenance*) as illustrated in Figure 1. Column* Rf1 *can be used to specify the relationship between the specific tuple and the data it annotates. It may contain* Name *values assuming that* Name *is the key in* Customers*. For instance, tuple* [BCT, NJDB, 147.52.7.8, http] *in* Provenance *would indicate that the* BCT *customer data tuple was obtained from the* NJDB *source.*

This modeling approach has two main drawbacks. First, it has a lot of information repetition. Assume that it has been asserted that all the New Jersey customers originate from the same data source. To record that, a tuple like the one just mentioned will have to be repeated in table Provenance *for every New Jersey customer. The second drawback is that this mechanism cannot be used to model the fact that a* Provenance *tuple may not refer to the whole* Customers *tuple but only to a subset of its attributes.*

What we propose is to allow queries to be used as values in the table columns. In particular, to have some columns recording query expressions used to intensionally describe data a metadata tuple is associated to. To find whether a particular data value is associated with a given (metadata) tuple, one only needs to check if the data value is part of the relation described by the query expression.

EXAMPLE 2.2. *In the example database of Figure 1, column* Rf1 *of table* Provenance *contains queries instead of atomic values. The first tuple with query* $q1$ *in column* Rf1 *intensionally describes that the provenance of all the customers with location 'NJ' is the* NJDB *data source. Furthermore, through the attributes of its* select *clause, it specifies that this is true only for attributes* Name, Type *and* PhoneLine*. It states nothing about attributes* Loc *and* CircuitID*. In a similar fashion, the second tuple specifies that data source '3State' is the origin of the* Loc, PhoneLine *and* CircuitID *values of all the business* Customers*.*

EXAMPLE 2.3. *The data in table* Customers *often needs to be verified for their consistency. This is common practice in large database applications where errors appear frequently [5]. In the current application, this is done by a number of technicians from various companies. Not all technicians are qualified to verify the correctness of every data element in the* Customers *table. There are certain rules that govern this qualification. For example, any* CISCO *technician can verify that the circuit id recorded in the database for any residential customer is correct, any* Verizon *technician can verify the correctness of the recorded phone number or circuit id (or their association) of any business customer, and any* AT&T *technician can verify the correctness of the phone*

number of any customer. The owner of the database would like to annotate the data in the `Customers` *table with the technicians that are eligible for performing the verification task, so that given some data values, it is easy to find who can be called to perform the verification. To do that, the technician information is recorded in a new database table* `Technicians`*. The relationship between technicians and customers is modeled through a new table* `XJ` *with columns* `Qc` *and* `Qt`*, both containing queries as values. Their contents are presented in Figure 1. The tuples in the table* `XJ` *model these rules.*

With the proposed mechanism, one can easily introduce new metadata on top of other existing metadata. It is only a matter of creation of a new table and of specifying the right queries as values in one of its columns. Also, different metadata tuples in the same metadata table can refer to different data or metadata tables. These are important features since the distinction between data and metadata is usually blurred. The same piece of information may be viewed as data by one application and as metadata by another.

EXAMPLE 2.4. *Suppose that a set of security policies need to be specified for some of the data. For simplicity, assume that these policies include only the group of users who can access the relevant data. The system administrator would like to annotate both the* `Customers` *and* `Provenance` *tables with the access permissions information. To achieve it, she creates a new* `Permissions` *table as illustrated in Figure 1. The first tuple of that table, through query* q_{11} *stored as a value in column* `Rf2`*, indicates that records in the* `Provenance` *table whose IP is in the 147.* domain can be accessed only by an administrator. The second tuple, through query* q_{12}*, indicates that the* `Name` *field in the* `Customers` *table with location 'NY' can be accessed by a guest user.*

It is also important to note that through the proposed modeling of metadata information as data, and of the associations between tables through attributes with queries as values, metadata can have any complex structure. In particular, a piece of metadata information may have multiple columns with different types.

3. QUERIES AS DATA VALUES

This section formally defines the semantics and the use of query expressions as data values and the operators on them.

3.1 Query-Types

The adopted type system is the one of the relational model extended with a new user defined atomic type called Q-type. Q-types provide the means to store queries as values in relational tables, in a fashion similar to INGRES [23] or Meta-SQL [9]. User defined types are used the same way any other primitive atomic type is used. The ability to define and use such types is part of the SQL Standard and is currently supported by most commercial database management systems. A value of type Q-type, referred to as Q-value, is a relational query expression.

To be able to dynamically execute queries stored as values, we assume the existence of a function $eval$ whose role is to evaluate a query expression that is provided to it as argument. However, in contrast to other approaches that use query expressions as data values [23, 18], we do *not* propose to extend relational algebra to include this function as an operator. Such an extension would have two major implications. The first is that it would have required the use of a nested relational model, instead of the simpler flat (first normal form) relational model. If, for instance, $eval$ was part of the extended relational algebra, applying it on a Q-type column

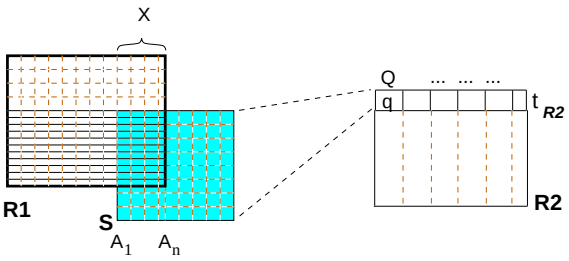

Figure 2: Q-type joins

would have returned a relation in which the contents of that column would have been relations, i.e., the result would have been a non-first normal form relation. The second disadvantage is that it would have created results with an unspecified schema. Different Q-values have query expressions that can return different numbers and types of attributes. Evaluating the Q-type column would have resulted in multiple non union-compatible relations.

EXAMPLE 3.1. *Consider the query* select * from `Provenance` *applied on the database instance of Figure 1. If the Query column* `Rf1` *is evaluated prior to retrieval, the result will be a relation with the last 3 attributes of* `Provenance` *table and the attributes returned by the execution of the query in column* `Rf1`*. Queries* q_1 *and* q_2 *are not union-compatible (they have different* select *clauses). Due to this, in the execution of query* select * from `Provenance`*, if an unnest operation is applied on the returned result relation of queries* q_1 *and* q_2*, the final result will have tuples with different number and kind of attributes, hence, it would not be a relation.*

The only *additional* functionality that we need is a new kind of predicate (i.e., conditional expression) that makes use of the $eval$ function. Such a predicate can be a parameter of the standard operators of relational algebra, including selection and join, yielding functionality that can effectively use Q-values for associating metadata tables with data (or other metadata) tables, as described next.

The functionality described here can also be achieved in the nested relational model, where an entire relation is stored as a value of an attribute. That model is much more powerful but, unfortunately, its richer functionality comes with a much higher cost. Our goal is to achieve what we want with minimal cost. We chose to implement our solution to the relational model that currently dominates the commercial DBMS world.

3.2 Selections on Q-values

Although for storage and retrieval purposes Q-type column contents are viewed as atomic types, for comparison purposes, we would like to view Q-values as relations. Since a Q-value is nothing more than an intensional description of a virtual relation, the functionality needed is the one that allows checking whether certain values exist in the relation described by a Q-value. If they do, it is said that the Q-value *references* these values.

DEFINITION 3.2. *For a relation* R *with a Q-type column* Q*, let* t *be a tuple of that relation and* q *the value of that tuple on column* Q*. Assume that* $\mathcal{A}$ *is the set of attributes of the result relation* $eval(q)$*. If* $v_1, \ldots, v_n$ *are atomic values and* $A_1, \ldots, A_n$ *are attribute names, expression* $R.Q[A_1, \ldots, A_n] \doteq [v_1, \ldots, v_n]$ *evaluates to true for* $R.Q{=}q$ *if the following conditions are satisfied.*

1. $\forall i = 1..n\ A_i \in \mathcal{A}$*, and*
2. $\exists t' \in eval(q)$ *such that* $t'[A_i] = v_i, \forall i = 1..n$.

EXAMPLE 3.3. *Assume that a data administrator would like to know what data sources are related to customer 'AFLAC'. This translates to selecting from the* Provenance *table those tuples having a* Q*-value in attribute* Rf1 *that references* Name *and that name is 'AFLAC'.*

This can be expressed as:

select distinct p.Source from Provenance p
where p.Rf1[Name] $\doteq$ [$'AFLAC'$].

Note that the semantics of the select operation have not changed. The only new part is the introduction of the "$\doteq$" conditional expression (predicate) for values of type Q. The way this condition is evaluated is a topic of a subsequent section. The symbol "$\doteq$" is used instead of "=" to emphasize the different predicate.

Another point worth clarifying is that Q-values can specify multiple conditions on multiple attributes. Our examples are kept simple for expository purposes.

3.3 Joins Using a Q-value

The values $v_1, \ldots, v_n$ in Definition 3.2 can be either constant values as is the case in Example 3.3, or relational atomic expressions that take values during query evaluation. The ability to use such expressions provides the means to form joins that are based on Q-type columns, allowing tables representing metadata to be associated to the data tables.

EXAMPLE 3.4. *A data administrator has discovered that in the database in Figure 1 the customer names starting with "A" violate the format policy and would like to know the source from where they originate. She knows that the provenance information is stored in the* Provenance *table where attribute* Rf1 *specifies the association between the data and the metadata. She issues the following query:*

select distinct p.Source from Customers c, Provenance p
where p.Rf1[Name] $\doteq$[c.Name] and c.Name *LIKE 'A%'*

What the query does is to select all the customers whose name starts with letter "A". For every such tuple c it checks if there is a tuple p in Provenance *with a* Q*-value q in p.*Rf1 *such that relation $eval(q)$ has an attribute* Name *and there is at least one tuple in it with the value in column* Name *equal to the value of c.*Name*. If yes, then tuples c and p pair up. The answer of the above query on the instance of Figure 1 is the tuple* [$'$NJDB$'$].

Note again that this has the same semantics as the regular SQL join operator (only the join condition is different since it is based on a Q*-type column). The result of its execution if the* select *clause was* select *"*", would consist of the attributes of* Customers *concatenated to the attributes of* Provenance*. The* Q*-type attribute values will be the query expression presented as a string.*

Figure 2 provides a visual explanation of when a tuple in one relation R_2 can form a join with tuples in a relation R_1 when the join is based on its Q-type column Q and the set of attributes $X = \{X_1, \ldots, X_n\}$ of R_1. This is represented through the condition $R_2.Q[A_1, \ldots, A_n] \doteq [R_1.X_1, \ldots, R_1.X_n]$. The shaded square represents the result relation S of $eval(q)$ where q is the value of column $t_{R_2}[Q]$ of a tuple t_{R_2} of relation R_2. A tuple t_{R_1} of R_1 pairs up with tuple t_{R_2} if all the attributes in X also exist in S, and there is at least one tuple in S with which t_{R_1} agrees on all the attributes in X. Those tuples of R_1 that will join with tuple t_{R_2} are illustrated in R_1 by the thick horizontal lines.

EXAMPLE 3.5. *The data administrator wants to find the names of the technicians who can verify the consistency of the circuit id of New Jersey business customers. To determine that, she is looking for the* Technicians *annotations that have been placed over New Jersey business customer* CircuitID *numbers. The association between the* Customers *data and its* Technicians *metadata information is done through a many-to-many relationship implemented by table* XJ*. Note that in contrast to traditional join approaches, there are no common attributes between* XJ*,* Customers*, and* Technicians*. The join is achieved through the intensional description of the queries stored in the two* Q*-type columns of table* XJ*. First, a query is constructed to retrieve the New Jersey business customers.*

select * from Customers c
where Loc=*'NJ'* and c.Type=*'business'*

It is then enhanced to also retrieve the tuples of table XJ *that contain some intensional reference to the circuit ids of these* Customers*. This is achieved through a join between the* Q*-values in attribute* Qc *and the attribute* CircuitID *of* Customers*. The query becomes:*

select * from Customers c, XJ j
where c.Loc=*'NJ'* and c.Type=*'business'* and
j.Qc[CircuitID]$\doteq$[c.CircuitID]

From the tuples that appear in XJ*, the first is for residential customers, so it cannot satisfy the query specifications. The third tuple does not mention* CircuitID *in its* select *clause, so it cannot satisfy the query specifications either. The second tuple of* XJ *will form a join pair with every* Customers *tuple that agrees on the* CircuitID *attribute value with at least one tuple in the evaluation of the query q_{C2}.*

The result of the join between Customers *and* XJ *will also have to be associated with the* Technicians *tuples. This is done in a similar way through a join on column* Qt *of* XJ *and the attributes of* Technicians*. The final query is:*

select t.Name from Customers c, XJ j, Technicians t
where c.Loc=*'NJ'* and c.Type=*'business'* and
j.Qc[CircuitID]$\doteq$[c.CircuitID] and
j.Qt[Name]$\doteq$[t.Name]

If the administrator was interested in the names of the technicians that can verify either *the phone and the circuit id of business customers, then the final query would have been:*

select t.Name from Customers c, XJ j, Technicians t
where c.Loc=*'NJ'* and c.Type=*'business'* and
(j.Qc[PhoneLine]$\doteq$[c.PhoneLine] or
j.Qc[CircuitID]$\doteq$[c.CircuitID]*)*
and j.Qt[Name]$\doteq$[t.Name]

In this case, both the second and the third tuples of XJ *would have been relevant. If the administrator was interested in technicians that can verify* both *the phone and the circuit id, then the final query would have been:*

select t.Name from Customers c, XJ j, Technicians t
where c.Loc=*'NJ'* and c.Type=*'business'* and
j.Qc[PhoneLine, CircuitID]$\doteq$[c.PhoneLine, c.CircuitID]
and j.Qt[Name]$\doteq$[t.Name]

which is not equivalent to the query:

select t.Name from Customers c, XJ j, Technicians t
where c.Loc=*'NJ'* and c.Type=*'business'* and
j.Qc[PhoneLine]$\doteq$[c.PhoneLine] and
j.Qc[CircuitID]$\doteq$[c.CircuitID]
and j.Qt[Name]$\doteq$[t.Name]

This is because, in principle, different *tuples in the evaluation of a* Q*-value query expression can satisfy the two conditions linked by the* and*, while in the former query the* same *tuple in the evaluation of the* Q*-value query expression has to satisfy the joint condition on phone and circuit id.*

THEOREM 3.6. *Given a* Q*-value* q*, the* equivalent class *of* q *is the set of* Q *values whose query is equivalent to the query of* q*. The results of selections and joins are not affected by the use of different* Q*-values assuming that they all belong to the same equivalent class.*

To realize why this is true one needs to observe that selections and joins on Q-type attributes are based on the data instances of the relations generated by the evaluation of the Q-values. Thus, replacing a Q-value with an equivalent one, will not affect any query that is based on that Q-value. The same applies if a tuple t containing a Q-value q is replaced by a set of tuples that differ from t only on the Q-type attribute and the union of the evaluation of their Q-values is equal to the evaluation of q.

3.4 Joins Between Q-values

Joins can also be performed on Q-type columns of different relations, providing the ability to check for the existence of a common tuple in the result relations of the evaluation of their respective Q-values.

DEFINITION 3.7. *Let* t_1 *(*t_2*) denote a tuple of relation* R_1 *(respective* R_2*), with a* Q*-type attribute* Q_1 *(*Q_2*), and* $\mathcal{A}_1$ *(*$\mathcal{A}_2$*) be the set of attributes in the result relation of* $eval(t_1[Q_1])$ *(*$eval(t_2[Q_2])$*). If* $A_1, \ldots, A_n, A'_1, \ldots, A'_n$ *are attribute names, expression* $t_1.Q_1[A_1, \ldots, A_n] \doteq t_2.Q_2[A'_1, \ldots, A'_n]$ *evaluates to true if*

1. $\forall i = 1..n$ $A_i \in \mathcal{A}_1$ *and* $A'_i \in \mathcal{A}_2$ *, and*
2. $\exists t' \in eval(t_1[Q_1])$, $\exists t'' \in eval(t_2[Q_2])$*, such that* $t'[A_i] = t''[A'_i]$, $\forall i = 1..n$*.*

EXAMPLE 3.8. *Assume that one would like to know which data sources have contributed to tuples that have the same* `PhoneLine` *as tuples referenced by the* `NJDB` *source. This can be expressed as:*

```
select  p2.*  from  Provenance p1, p2
where   p1.Source='NJDB' and
        p1.Rf1[PhoneLine] ≐ p2.Rf1[PhoneLine]
```

The answer to this query will include both tuples of the `Provenance` *table. In particular, the second tuple will be included since q mentions* `PhoneLine` *in its* select *clause, and both* $eval(q1)$ *and* $eval(q2)$ *reference common* `Customers` *tuples.*

3.5 Discussions

Using Catalog Tables: One of the issues that we have not explicitly discussed yet is how one can know which are the right tables to join. How can one know that the information stored in a specific table is metadata information of another, and how to know that a join on a specific Q-type attribute makes sense. This question is no different from the question of finding the right join paths in a relational schema. Recall that the relational model does not disallow joins that are based on any type-compatible attributes, even if they do not necessarily make semantic sense. It is up to the user to identify the right semantics either with external knowledge or by looking at schema constraints like key/foreign key relationships. In a similar fashion, it is assumed that the catalog tables of a database management system record the Q-type attributes that exist in the database. By querying it, one can find what parts of what tables are referenced by Q-values to form meaningful joins.

Syntactic Correctness: Since the query expressions in Q-values may have to be executed at run time, a fundamental requirement one wants to guarantee is their syntactic correctness. Syntactic correctness can be checked at the time of the value insertion or modification through a function that plays a role similar to the check constraints defined on relational tables. The function can also check for possible recursions.

A related issue is what happens when schemas are altered, because certain queries of the existing Q-values may become inconsistent. Mechanisms similar to those used for views when the underlying schema is altered [20, 25] can be used to deal with these cases.

Dynamic vs Static Associations: Q-values naturally support *dynamic* associations. As such, no special care needs to be taken when the data is updated. Consider, for instance, a data tuple t that satisfies the conditions of the query expression of a Q-value q in a metadata tuple t_m. Naturally, this means that t_m is associated to t. Now, assume that t is modified. If its modified values continue to satisfy the conditions of q, t will remain associated to t_m, but if not, it will not. If another tuple t' that was not initially associated to t_m is modified and its new values are such that they satisfy the conditions of q, then t' will become automatically associated to t_m. This helps having metadata with generic references to data, independently of the current database instance. In Figure 1, for instance, the first tuple of table `Provenance` is meant to be associated to any tuple of customers with location 'NJ', that exists, or may exist in the future, in the database. This form of dynamic behavior is also found in views where the view query specifies certain conditions and at any point in time, its instance depends on the instance of the base tables.

An alternative semantics is the one in which a metadata entry is associated with certain data tuples and only with them. This means that if at a later time new data tuples are added, even if they satisfy the conditions of the Q-value query, they will not be considered associated to the metadata tuple. To support this kind of *static* semantics, the system needs to "remember" the data with which a metadata entry was initially associated. In our system dynamic semantics is the default, but static semantics is also supported.

4. IMPLEMENTING QUERY-TYPES

This section describes how Q-types can be implemented in a database management system in order to provide the ability to efficiently and effectively manage data, metadata and their associations in a unified way. A highlight of this approach is that it builds on existing relational database technology, that makes it possible to build on top of modern commercial database management systems.

First, we present a pure rewriting-based strategy. Then, we describe how index structures on the Q-type attributes, referred to as Q-indexes, can be used to speed up query evaluation.

4.1 Storage

Our system consists of two main components. One is a set of auxiliary tables that are used to facilitate query answering involving the Q-values. These tables can be considered part of the catalog schema, thus they do not appear as part of the database schema presented to the user. Query answering is performed by the second component which is a query preprocessor. Its role is to identify the parts of the query that refer to Q-type columns and rewrite them to expressions that use the auxiliary tables. The outcome of the preprocessor is a query in standard SQL that can be executed by the database management system. The result of the query is a relation as expected. Q-type column values are presented as strings by default, but additional functions can be defined to transform them to other forms.

A first step in supporting queries as values is the introduction of a

QTypeValues

Qid	TblName	ColName	RId
q1	Provenance	Rf1	1101
q2	Provenance	Rf1	1011
...	...	...	

AttrTbl

Qid	AttrName
q1	Name
q1	Type
q1	PhoneLine
q2	Loc
q2	PhoneLine
q2	CircuitID
...	...

Figure 3: Auxiliary tables for Q-values

new user-defined type called Q. The type is defined as an extension of the string atomic type, in order to store the query expressions.

A critical task for the query processing system is to be able to identify and use in a declarative fashion the attributes of the relations $eval$(q), i.e., the relations generated by the evaluation of the query expressions of the Q-values. In current DBMSs, the attribute names of a table can be turned into values of a column (so that the relational operators can be applied on them) by using the UNPIVOT operator [7]. Unfortunately, UNPIVOT works only on relations that the DBMS is aware of, i.e., the database tables. Since the attributes in the select clause of the Q-values are not recorded in the catalog tables, UNPIVOT cannot be used for them. To overcome this limitation, two auxiliary tables are introduced, whose role is to record that information: QTypeValues and AttrTbl.

The QTypeValues table associates a unique identifier for each Q-value, which is identified using a combination of the (metadata) table name, the column name, and record identifier of the tuple in which the Q-value appears. For each Q-value that exists in the database, and for each attribute name of its virtual relation, there is one tuple in table AttrTbl, which contains the unique identifier of the Q-type value (from the QTypeValues table), and records an attribute name of its virtual relation. Figure 3 illustrates part of the contents of the auxiliary tables QTypeValues and AttrTbl, that record the information for the two tuples of table Provenance as described in Figure 1.

4.2 Query Evaluation: Alternatives

Having information about the $eval$(q) relation attributes for the query expression q of every Q-value recorded, user queries involving (the newly introduced) conditions on Q-type columns can be evaluated. There are different strategies that can be followed, each one with its own advantages and disadvantages. Assume that a user query has a condition of the form $S.a[A_1, ..., A_n] \doteq [v_1, ..., v_n]$. One approach is to ignore the condition initially and evaluate the rest of the query as usual. Then for every variable binding that is found to be satisfactory, find the Q-value to which expression $S.a$ evaluates, the constant values to which expressions $v_1, ..., v_n$ evaluate, and test whether the conditions in Definition 3.2 are satisfied. The drawback of this approach is that the conditions of Definition 3.2 will have to be checked for every variable binding satisfying the specifications of the remaining part of the user query. Checking these conditions means evaluating the query expressions of the Q-values each time. Thus, this approach is preferable in the case where the variable bindings found to satisfy the remaining part of the user query are highly selective compared to the bindings and the number of Q-type values that satisfy condition $S.a[A_1, ..., A_n] \doteq [v_1, ..., v_n]$.

An alternative approach is to start from condition $S.a[A_1, ..., A_n] \doteq [v_1, ..., v_n]$ by first finding all the Q-values in column $S.a$ whose query expression select clause specifies *each* attribute name in the set $A_1, ..., A_n$. For each Q-value that passes this first test, all the variable bindings $v_1, ..., v_n$ that satisfy condition $S.a[A_1, ..., A_n] \doteq [v_1, ..., v_n]$ are computed, and for each one of them, it is checked whether the rest of the conditions specified in the user query are also satisfied. This approach is preferable if the number of Q-values that pass the first test and the variable bindings that satisfy condition $S.a[A_1, ..., A_n] \doteq [v_1, ..., v_n]$ is much smaller than those bindings that satisfy the other conditions that exist in the user query.

EXAMPLE 4.1. *Consider the query*
select p.Source from Customers c, Provenance p
where p.Rf1[Name]$\doteq$[c.Name] and c.Name *LIKE 'A%'*

introduced in Example 3.4. The first approach suggests to ignore the condition on p.Rf1*, find all the customers with a name starting with A and make all the possible pairs with the* Provenance *tuples. For each such pair, take the* Q*-value in column* Rf1 *of* Provenance*. Test whether the evaluation of that query would have contained a column* Name*. This is done either by analyzing its* select *clause, or by performing a lookup on table* AttrTbl *where this information has been recorded. If the* Q*-value fails to pass the test, the pair of* Customers *and* Provenance *tuple is rejected. If not, the query expression is evaluated, and it is checked whether in the result relation there is a tuple with a value* v *in column* Name *equal to the value in the* Name *attribute of the* Customers *tuple (this is the value to which expression* c.Name *evaluates). If this test is also passed the value of attribute* Source *is reported to the user. The drawback here is that the* Q*-value expressions will have to be evaluated multiple times.*

The second approach suggests to evaluate each query that appears in column Rf1 *first, and check whether it has an attribute* Name *in its result set. For those that do, select the values that appear in that attribute and build a set of names. Then, the remainder of the query can be evaluated where instead of condition* p.Rf1[Name]$\doteq$[c.Name]*, it is now required that the value to which expression* c.Name *evaluates exists in the set of names that was constructed. The drawback here is that the set of names may be really large, but only a few satisfy the condition about starting with A, which means that many query expression evaluations could have been avoided.*

Clearly there is no best approach to follow always. Each time it depends on the specific query that is executed. The decision on what to follow cannot be made without some data statistics like those kept by the database management system. Unfortunately, these statistics are not always available to external applications. For that, we will try to rewrite the query in such a way that the database engine will be able to take the right decisions based on its data statistics information.

The main issue in this process is that the expressions of the Q-values are not part of the query provided by the user or the application. They are "hidden" as values in the database. Thus, the query engine cannot use them when deciding the evaluation and optimization strategy. To overcome this issue, we introduce a preprocessing step that has two main goals.

The first goal is to check the satisfaction of the first condition in Definition 3.2. As explained, such a check could not be performed by the query engine since the attributes are not explicitly mentioned in the user query but are encoded in the select clause of the Q-value expressions. This preprocessing step can be seen as an UNPIVOT operation followed by a relational selection on the attribute names. The difference is that instead of being applied on the materialized tables of the database, it is applied on the virtual relations specified by the query expressions in the Q-values.

The second goal is to expose these query expressions of the Q-type values to the database optimization engine, by making them part of the query that the user or the application has posed. That

way, the query optimizer will be able to balance all the factors and, based on the information that the DBMS has about data distributions and value cardinalities, it will take the most promising decision for the task at hand.

Once this preprocessing step is done, the $\doteq$ condition can be removed from the modified user query expression. A detailed description of how this is achieved is described next.

4.3 Query Rewriting: Using Union

Consider a user query of the form

Q_u: select exp_1^u,..., exp_g^u from R_1^u, ..., R_h^u, T
where $cond_1^u$ and ... and $cond_f^u$ and
$T.a[A_1, ..., A_n] \doteq [e_1, ..., e_n]$.

Step 1: The first step is to identify which of all the Q-values in attribute $T.a$ have attributes named $A_1, ..., A_n$ in the relation obtained by the evaluation of their query expression. This is achieved through the following query on the auxiliary tables AttrTbl and QTypeValues:

select tv.RId
from QTypeValues tv, AttrTbl at_1, AttrTbl at_2, ..., AttrTbl at_n
where tv.TblName='T' and tv.ColName='a'
and tv.Qid=at_1.Qid and at_1.AttrName='A_1'
and tv.Qid=at_2.Qid and at_2.AttrName='A_2'
...
and tv.Qid=at_n.Qid and at_n.AttrName='A_n'

Step 2: From all the Q-values identified in Step 1, only those whose query evaluation has a tuple with value e_j in attribute A_j, $\forall j = 1..n$, have to be kept.

Let q_i be a Q-value identified through the first step.
A new query Q_i is constructed as follows:

Q_i: select *
from (q_i) AS R
where $R.A_1$=e_1 and ... and $R.A_n$=e_n

This query answers the question of whether $eval(q_i)$ has a tuple with a value e_j in attribute A_j, where j ranges between 1 and n. If the result is an empty set, Q-value q_i does not satisfy condition $T.a[A_1, ..., A_n] \doteq [e_1, ..., e_n]$. Since we are not interested in the actual results of query Q_i but only in finding whether it returns an empty set or not, the select clause can be rewritten to select TOP 1 '1'. The constant value '1' is a random constant and the 'TOP 1' clause instructs the query processor to return only the first tuple in the result set with some potential saving in execution time.

Step 3: Once the set of queries Q_i have been constructed for the Q-type values returned during Step 1, condition $T.a[A_1, ..., A_n] \doteq [e_1, ..., e_n]$ in the user query can be replaced by a condition that tests whether the Q-value of attribute $T.a$ is one of those computed during Step 1, and if so, that its respective query Q_i returns a non-empty result set. To do that, we exploit the Union feature of SQL queries. After the replacement of condition $T.a[A_1, ..., A_n] \doteq [e_1, ..., e_n]$, query Q_u becomes:

select exp_1^u,..., exp_g^u from R_1^u, ..., R_h^u, T, (q_1) AS R
where $cond_1^u$ and ... and $cond_f^u$ and
T.RId=rid_1 and ($R.A_1$=e_1 and ... and $R.A_n$=e_n)
Union
select exp_1^u,..., exp_g^u from R_1^u, ..., R_h^u, T, (q_2) AS R
where $cond_1^u$ and ... and $cond_f^u$ and
T.RId=rid_2 and ($R.A_1$=e_1 and ... and $R.A_n$=e_n)
Union
...
Union
select exp_1^u,..., exp_g^u from R_1^u, ..., R_h^u, T, (q_i) AS R
where $cond_1^u$ and ... and $cond_f^u$ and
T.RId=rid_i and ($R.A_1$=e_1 and ... and $R.A_n$=e_n)

To avoid the nested queries, the above expression can be rewritten by embedding the from and where clause of the q_i query in the respective clauses of the union query component they appear in. For instance, if query q_i is:

select e'_p AS A_p, e'_o AS A_o, ..., e'_k AS A_k, e'_r AS A_r
from S_1, ..., S_s
where $cond_1$ and ... and $cond_t$

the last component of the union expression can become:

select exp_1^u,..., exp_g^u from R_1^u, ..., R_h^u, T, S_1, ..., S_s
where $cond_1^u$ and ... and $cond_f^u$ and
T.RId=rid_i and (e'_1=e_1 and ... and e'_n=e_n) and
$cond_1$ and ... and $cond_t$

All the operators in the rewritten query are standard SQL operators, thus, it can now be sent to the database management system for execution. Furthermore, since the query expressions q_i of the Q-values are explicitly mentioned in the query, the optimizer will be able to take them into consideration, and come up with the best evaluation strategy.

EXAMPLE 4.2. *The user query*

select p.Source from Customers c, Provenance p
where p.Rf1[PhoneLine]$\doteq$[c.PhoneLine]
and c.Name *LIKE 'A%'*

will get the form:

select p.Source
from Customers c, Provenance p, Customers $c2$
where c.Name *LIKE 'A%'* and p.RId=$r1$ and
$c2$.Loc=*'NJ'* and $c2$.PhoneLine=c.PhoneLine
Union
select p.Source
from Customers c, Provenance p, Customers $c2$
where c.Name *LIKE 'A%'* and p.RId=$r2$ and
$c2$.Type=*'business'* and $c2$.PhoneLine=c.PhoneLine
Union
....

The ... symbol in the query denotes additional cases that may exist due to other Q*-values of attribute* Rf1 *in table* Provenance *that may qualify but do not appear in the portion of the relation illustrated in Figure 1. Note also that due to Step 1, the above queries are guaranteed to be union compatible.*

An alternative approach is to define virtual views based on the expressions of every Q-value that exists in the database. The rewriting can then refer to the virtual relation described by the Q-value through the respective view. We have tried that approach but we found it having a very poor performance, which may be due to the way the query optimizer was handling the virtual views, so we did not consider this approach further.

4.4 Query-Indexes

Query evaluation based on the query rewriting approach described in the previous section is expected to be efficient when the rewritten query has only a few disjuncts, i.e., when the number of Q-type values in attribute $T.a$ that are identified in Step 1 is small. If the metadata table contains a large number of Q-values, all of which have attributes named $A_1, \ldots, A_n$, then each of these Q-values would need to be evaluated, even if $[e_1, \ldots, e_n]$ were a tuple of constants. To further prune out "irrelevant" Q-values in attribute $T.a$, the values present in the relations obtained by evaluating the queries would need to be used.

In this section, we present such an approach based on maintaining indexes on the Q-values, which we refer to as Q-indexes, for this purpose. Our Q-indexes can be easily realized using relational tables and B-tree index structures available in commercial relational database management systems. Such an index can complement other approaches that use queries as values [23] since it can help them perform their functionality more efficiently.

4.4.1 Indexing Alternatives

Clearly, the best possible index from the perspective of minimizing the query execution time would be an index that given any n-tuple of attributes $[A_1, \ldots, A_n]$ and any n-tuple of values $[v_1, \ldots, v_n]$ would precisely identify the Q-values in attribute $T.a$ that satisfy the conditions of Definition 3.2. We implemented such an index and found that it is unlikely to be feasible in practice, since it would need to essentially index the "union" of the relations obtained by evaluating all the queries in attribute $T.a$.

A more space-efficient alternative is to build multiple single attribute Q-indexes that given any attribute A_i and any value v_i would precisely identify the Q-values in attribute $T.a$ that satisfy the conditions of Definition 3.2. Such an approach has both advantages and disadvantages.

The key disadvantage is that given an n-tuple of attributes $[A_1, \ldots, A_n]$ and an n-tuple of values $[v_1, \ldots, v_n]$, one cannot precisely determine the desired Q-values, using the single attribute Q-indexes. Intuitively, the reason is that even if each v_i appeared in attribute A_i of a query q's relation R_q, they may not all be present in the *same* tuple of R_q.

However, the single attribute Q-indexes can be used as an *effective filter*, since they may have false positives when $n > 1$, but they do not have false negatives. For n=1 there are neither false positives nor false negatives.

The key advantages of maintaining multiple single attribute Q-indexes instead of the covering index on all attributes are the significantly lower space cost, and the consequently lower cost of constructing and maintaining such indexes. Intuitively, the reason is that even if a value v_j appears in attribute A_i in multiple tuples of a query q's relation R_q, the pair (v_j, q) need be indexed only once in the single attribute Q-index of A_i.

4.4.2 Realizing Single Attribute Query-Indexes

A simple and elegant way of implementing a single attribute Q-index on attribute A_i, in a commercial database system, is to (i) materialize a 3-ary relational table $(val, qid, count)$, where the meaning of a tuple (v, q, c) in this table is that value v appears in attribute A_i of c different tuples of query q's relation R_q, (ii) make the pair $<val, qid>$ a key for that table, and (iii) create a B-tree index on val. The *count* field is present merely to efficiently maintain this index under insertions, deletions and modifications to the base tables used in the definitions of the metadata queries.

EXAMPLE 4.3. *The table* ITName *and* ITType *in Figure 4 illustrate a fraction of a realization of a single-attribute index for the attributes* Name *and* Type*, respectively, of the database instance in Figure 1. The interesting observation is that since $<val, qid>$ is a key, it is likely to have a unique index. If a* Q*-type join involves only one attribute (case n=1 mentioned in section 4.4.1), an intelligent query optimizer can implement the join using the index without having to access the data table.*

4.4.3 Updating Single Attribute Query-Indexes

For the Q-index to be useful, it would need to be efficiently updatable as data and metadata entries are inserted, deleted and updated in the database. We next describe how this can be achieved.

Suppose that the various queries in the Q-type attribute $T.a$ are select-project queries over single tables, and their where clauses are conjunctions of conditions of the form $R.A_k \leq v_k$ and $R.A_m \geq v_m$, where v_k and v_m are constants. Then, a condition of the form $R.A_k \leq v_k$ present in the where clause of query q_m can be represented in a relational table having the

Customers

Name	Type	...
AFLAC	business	...
J. Lu	residence	...
H. Ford	residence	...
AMEX	business	...
NJC	business	...
BCT	business	...
...	...	...

ITName

v	q	c
AFLAC	q1	1
H. Ford	q1	1
NJC	q1	1
BCT	q1	1

ITType

v	q	c
business	q1	3
residential	q1	1
...	...	...

Provenance

Rf1	Source	...
q1	NJDB	...
q2	3State	...
...	...	...

LEQ = GEQ

R	A	t	q
Customers	Loc	NJ	q1
Customers	Type	business	q2
...	...	...	...

Figure 4: RDBMS Q-index realization.

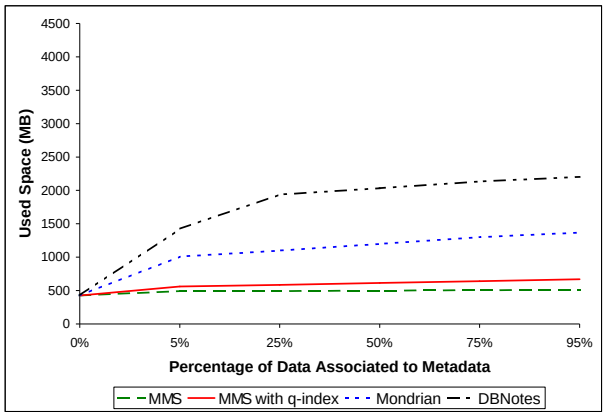

Figure 5: Size increase as a function of the percentage of data associated to Metadata.

schema $LEQ(relation, attribute, value, queryid)$. Similarly, conditions of the form $R.A_m \geq v_m$ can be represented in a relational table GEQ with the same schema. When a new tuple $(t_1, \ldots, t_n)$ is inserted in relation $R(A_1, \ldots, A_n)$, one can query the relation LEQ for tuples of the form $\{(R, A_i, t, q) | t < t_i, 1 \leq i \leq n\}$ and the relation GEQ for tuples of the form $\{(R, A_i, t, q) | t > t_i, 1 \leq i \leq n\}$. Such tuples can be efficiently identified using standard relational indexes such as B-trees on the concatenation of the first three attributes of LEQ (or GEQ). A query q present in an identified tuple is clearly *not* affected by the insertion, and can be eliminated. Any query that is not eliminated has all the conditions in its where clause satisfied by the newly inserted tuple, thereby identifying the tuples that need to be inserted/modified in the Q-index.

If a new tuple is inserted in a table with a Q-type column, the query expression of the Q-value for that column is evaluated. The values of each attribute in the result relation are used to update the respective tables of the Q-index. This is done by checking whether there is an entry with the pair $(value, queryid)$ already in the index table. If yes, its *count* value is increased by 1. Otherwise, a new entry $<value, queryid, 1>$ is inserted.

Note that using single attribute indexes does not mean that the Q-value queries have only one attribute. They may involve multiple attributes, but each index is on one attribute only.

The experimental section that follows will demonstrate the significant benefits obtained by using Q-indexes.

5. EXPERIMENTAL EVALUATION

Our proposed framework has been implemented in a system called MMS (Metadata Management System) [22] on top of a commercial relational database management system. A number of experiments were conducted with three metrics in mind: the space usage, the update cost and the query execution time. The results of the experiments indicated that: **(i)** Due to the intensional way in which the metadata is associated with the data in MMS, a lot of repetition is avoided and the space usage of the metadata was very low. **(ii)** In the absence of any Q-index, updates to both metadata and data entries are very cheap in MMS. Maintaining the Q-index under updates adds additional cost, but is still cheaper than existing metadata management systems for updates on metadata, while being more expensive for updates on the data. **(iii)** Query execution, in the absence of any Q-index, is expensive in MMS, since it involves the analysis and evaluation of each Q-type query value. However, the experiments indicate that the proposed Q-index can not only significantly reduce the query execution time, but in many cases offer much better performance than that offered by other existing metadata management systems. This is analogous to the benefits provided by the use of indexes for SQL query evaluation.

The experiments were performed on a dataset obtained from a real enterprise application having data about customers, their provisioned hardware, their billing information and the service orders that had been placed for them. On that data, we associated multi-attribute metadata information.

For comparison purposes, we chose two recent annotation management systems: DBNotes [4] and MONDRIAN [12], because annotations are a generic and commonly used kind of metadata. In terms of semantics, in these systems the association between the data and the annotation is explicit, and they cannot accommodate future tuples. For fairness, we have ignored that factor in our experiments in order to make sure that the expressive power of all the systems is the same.

In DBNotes, every relational table column is associated with a second column in the same table that is used to hold the annotation. If a value in a tuple has more than one annotation associated to it, then the tuple is recorded multiple times, once for every annotation. Since our metadata information consists of multiple fields, to simulate this behavior in DBNotes, we had to associate more than one column to each attribute. Mondrian follows a similar, but more compact, approach. For each relation there is one extra column that keeps the annotation, and also for each attribute there is a shadow column of type bit that can get values 0 or 1, specifying whether the annotation text refers to the respective attribute or not. Again, due to the structured nature of our metadata, to simulate the behavior in Mondrian, instead of one annotation column per table we had to have more. In our own approach, for each data table, we had another table in which the metadata was recorded (we will refer to these as "metadata" tables, as opposed to the "data" tables that contain the original data, although from the perspective of the MMS system any table can serve as a "data" or a "metadata" table). The metadata table had a Q-type column that was used to associate each metadata tuple with a set of data tuples in a data table. Note that in the extreme case where every tuple has a separate (and only one) metadata annotation, all three approaches are comparable.

For the experiments, we considered data tables of 5 million tuples and metadata tables of up to 500,000 tuples. Indexes were used in both Mondrian and DBNotes, as well as MMS.

5.1 Space Usage

First, we used all the data entries from the data tables and from the 500,000 metadata entries we randomly selected a fraction of them so that only a portion of the data was associated to some metadata. This was performed for portions of 5%, 25%, 50%, 75%, and 95%. For each one the size of the database was measured. The size was measured both with and without the Q-index. The same information was also encoded in separate databases using the Mondrian and the DBNotes schemes. The result sizes are all illustrated in Figure 5. The chart indicates that even when 95% of the data tuples has been associated with metadata, the MMS approach results in a minor increase in database size. On the other hand, Mondrian and DBNotes indicate a substantial growth in size as the percentage of the data bearing metadata gets larger. This is mainly because of the fact that in Mondrian and DBNotes the metadata information has to be repeated for every tuple it is associated to. For the case of DBNotes in particular, it has to be repeated for every column it is associated to. In MMS, on the other hand, the annotation text is stored only once in the metadata table, no matter how many tuples are associated to it, and most importantly, no additional information has to be stored in the data table itself. Furthermore, the use of the Q-index for the Q-type column in the metadata table results in only a minor increase in the total database size, a cost that is offset by the huge benefit the Q-index offers in terms of query execution time, as we will see later.

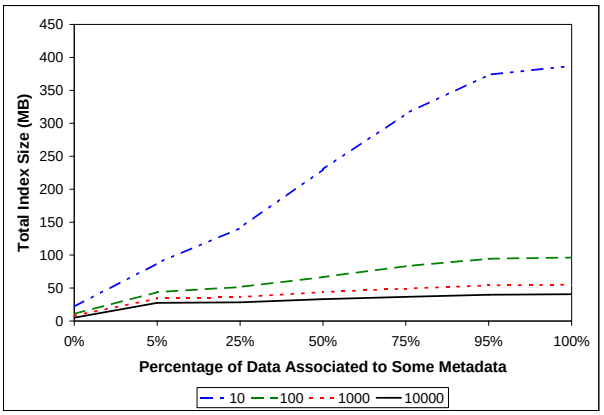

Figure 6: Q-Index size for different footprints.

The same experiment was repeated with double the number of metadata entries that are associated to each data tuple. The results were similar to those in in Figure 5, but with increased differences between the approaches. In particular, the size increase for Mondrian and DBNotes was substantial since each data entry had to be stored as many times as the number of metadata entries it was associated to, while with MMS only the metadata entries were doubled along with the respective Q-index entries.

To measure how the size of the Q-index structures are affected by the characteristics of the data, we kept the size of the data fixed and we chose different footprint values. The footprint is the number of data tuples a metadata entry is associated to. This is actually the number of tuples in the virtual relation specified by the Q-value. For each footprint value we generated metadata in a way that a specific portion of the data was associated to metadata. Figure 6 indicates the change in size for the different cases. When the footprint is larger, fewer metadata tuples are needed to cover a specific portion of the data, thus, the Q-index size is smaller. The Q-index size for the case of 0% is non-zero because the measurement includes the space allocated for the Q-index tables by the DBMS.

5.2 Update Cost

Here we investigate the cost of updating the various structures in the presence of data associated with metadata. The same dataset as

Footprint	MMS	MMS Q-index	Mondrian	DBNotes
10	0.002	0.092	1.350	0.650
100	0.002	0.075	1.340	0.654
1000	0.002	0.122	1.617	2.241
10000	0.002	0.600	1.789	2.555
100000	0.002	3.514	12.049	5.945

Table 2: Aver. metadata insertion time (in sec)

before was considered and a series of 2500 new metadata entries were inserted one after the other. The average time of performing one such insertion was measured for each of the four different approaches that we are testing. The results are illustrated in Table 2 for multiple such experiments with different footprints. Since in MMS the metadata are stored in a separate table from the data, and the metadata entries are much fewer than the data, insertions were extremely fast. For the Q-indexed version of MMS, some extra time is needed to update the Q-index structures but it is still faster than Mondrian and DBNotes.

Deletions were also tested on metadata. For Mondrian, DBNotes and the non-Q-indexed version of MMS the performance is similar to that of insertions. The Q-indexed version, on the other hand, indicates a 30% speedup over the one required for insertion.

Insertions and Deletions were also tested for data entries. Table 3 indicates the average time to delete a data tuple from a data table. For Mondrian, DBNotes and the non-Q-indexed MMS this time is affected by the size of the data table. Since MMS does not store the metadata along with the data, as Mondrian and DBNotes do, deletion time is better. The Q-indexed version of MMS, as expected, requires extra time in order to update the Q-index structures. The average time to execute a data insertion are analogous to those of data deletion for the same reasons.

5.3 Query Execution Time

The next set of experiments was performed in order to measure the query execution time. The important factor studied was how fast the join between data and metadata values can be performed using the Q-values. As before, a database was considered with data tables of 5 million tuples and metadata tables of various sizes up to 500,000 tuples. The kind of queries that were performed were those forming a join between the data and the metadata table based on the Q-values. The queries were run on databases with different Q-value footprints. For larger footprints the metadata table entries were naturally fewer (we kept only as many metadata entries as required to ensure that every data entry had some metadata tuple associated to it). The queries were performed on both the Q-indexed and the non-Q-indexed version of MMS, and their equivalent queries were also run on the DBNotes and Mondrian databases. The results are illustrated in Figure 7. The Q-indexed MMS version always significantly outperforms the other approaches. Furthermore, as the footprint value gets larger the execution time also gets smaller (since there are fewer metadata entries). The non-Q-indexed MMS (not shown in Figure 7) had much larger execution times because for each distinct Q-value, its query had to be evaluated. Such an evaluation took an average of 3 secs per Q-value, but since there were many Q-values the overall time was that many times larger.

One of the unique features of MMS is that metadata can be queried independently of the data it is associated to. This allows metadata of the same kind, say for instance, user comments, to be all stored together in the same table independent of the data for which the comment was made. This permits a very compact storage schema and allows queries of the form "What comments have been made so far mentioning circuit IDs?" to be efficiently answered. Answering these kinds of queries in the schemas provided

Metadata Entries	MMS	MMS with Q-index	Mondrian	DBNotes
500000	0.002	20.697	0.069	0.111
50000	0.002	2.345	0.114	0.105
5000	0.002	0.869	0.109	0.100
500	0.002	0.192	0.160	0.064
50	0.002	0.076	0.111	0.087

Table 3: Average data deletion time (in sec).

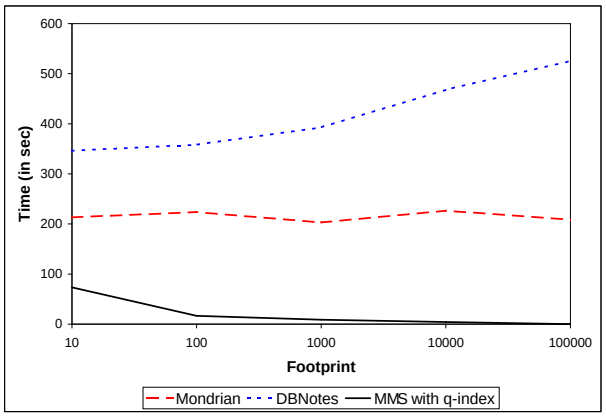

Figure 7: Average query execution time.

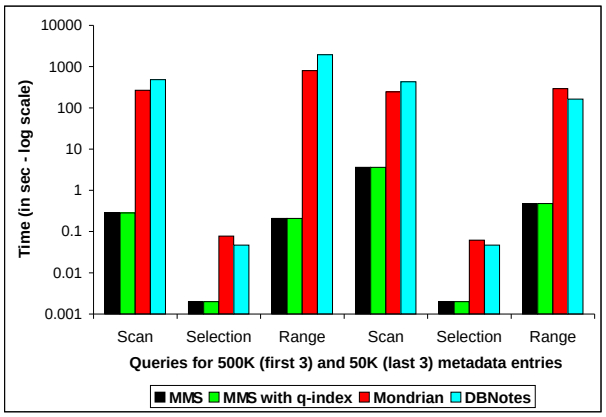

Figure 8: Avg. metadata-query execution.

by Mondrian or DBNotes would first require knowledge of the data schema, but even if the schema is known, the performance will be poor since data tables are usually huge. Figure 8 illustrates the average execution time of such queries in all the four approaches evaluated here.

Another interesting feature of MMS is that it allows metadata to be defined easily over other existing metadata. This requires queries with multiple joins based on Q-types, which introduces the need for the final experiment. In particular, we needed to investigate how query execution time is affected by the number of Q-type based joins that exist in the query. We tried the Q-indexed approach for databases with different footprints and for different numbers of joins. We made sure that for the same database, the queries, even though they had different numbers of joins, all returned results of the same size (viz., 1). That way, the observed differences in time were exclusively due to the number of joins. The results are illustrated in Table 4. Note that the query executions between the

	Footprint		
Joins	**1000**	**10000**	**100000**
1	0.35	1.182	0.121
2	0.21	1.767	0.431
3	0.28	0.771	0.343
4	0.26	1.923	1.072

Table 4: Avg. Q-type join execution (in sec).

different databases are not to be compared since they have different results. The graph intends to provide some intuition of the query executions within the same database. The conclusion is that the number of Q-type based joins in a query is not a factor that dramatically affects the query evaluation time.

6. CONCLUSION

There is a clear need to associate a variety of metadata with the underlying data, to understand, maintain, query, integrate and evolve databases. In this paper, we presented a simple, elegant approach to uniformly model and query data and arbitrary metadata. The key intuitions are that: (1) the relational model augmented with queries as data values is a natural way to uniformly model data, arbitrary metadata and their association, and (2) relational queries with a join mechanism augmented to permit matching of relations specified by Q-values, instead of only atomic values, is an elegant way to uniformly query across data and metadata.

Our MMS system implements this approach, providing a mechanism for recording metadata into a database without having to alter existing tables. User queries are efficiently evaluated using Q-indexes. We experimentally evaluated the performance of the MMS system, in comparison with previous proposals (DBNotes and MONDRIAN) for metadata management, and showed significant benefits both in terms of space usage and query execution times. Our results validate the generality and practicality of our uniform approach to metadata management.

7. REFERENCES

[1] Y. An, A. Borgida, and J. Mylopoulos. Refining Semantic Mappings from Relational Tables to Ontologies. In *SWDB*, pages 84–90, 2004.

[2] P. A. Bernstein. Repositories and object oriented databases. *SIGMOD Record*, 27(1):88–96, 1998.

[3] E. Bertino, S. Castano, and E. Ferrari. On specifying security policies for web documents with an XML-based language. In *SACMAT*, pages 57–65, 2001.

[4] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. In *VLDB*, pages 900–911, 2004.

[5] P. Buneman, S. Khanna, and W. Tan. On Propagation and Deletion of Annotations Through Views. In *PODS*, 2002.

[6] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *ICDE*, pages 4–19, 1998.

[7] C. Cunningham, G. Graefe, and C. A. Galindo-Legaria. PIVOT and UNPIVOT: Optimization and Execution Strategies in an R DBMS. In *VLDB*, 2004.

[8] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning*. Wiley Publishers, 2003.

[9] J. V. den Bussche, S. Vansummeren, and G. Vossen. Meta-SQL: Towards practical meta-querying. *Information Systems*, 30(4):317–332, 2005.

[10] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[11] D. Gawlick, D. Lenkov, A. Yalamanchi, and L. Chernobrod. Applications for Expression Data in Relational Database System. In *ICDE*, pages 609–620, 2004.

[12] F. Geerts, A. Kementsietsidis, and D. Milano. MONDRIAN: Annotating and querying databases through colors and blocks. In *ICDE*, 2006.

[13] M. Jarke, R. Gallersdorfer, M. A. Jeusfeld, and M. Staudt. ConceptBase - A Deductive Object Base for Meta Data Management. *J. Intell. Inf. Syst.*, 4(2):167–192, 1995.

[14] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. SchemaSQL: An extension to SQL for multidatabase interoperability. *ACM TODS*, 26(4):476–519, 2001.

[15] D. Maier and L. M. L. Delcambre. Superimposed Information for the Internet. In *WebDB*, pages 1–9, 1999.

[16] G. Mihaila, L. Raschid, and M.-E. Vidal. Querying "Quality of Data" Metadata. In *In Third IEEE META-DATA Conference, Bethesda, Maryland*, apr 1999.

[17] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing Knowledge About Information Systems. *ACM TODS*, 8(4):325–362, 1990.

[18] F. Neven, J. V. Bussche, D. V. Gucht, and G. Vossen. Typed Query Languages for Databases Containing Queries. In *PODS*, pages 189–196, 1998.

[19] R. Rose and J. Frew. Lineage Retrieval for Scientific Data Processing: A Survey. *ACM Comp. Surveys*, 37(1):1–28, 2005.

[20] E. A. Rundensteiner, A. Koeller, X. Zhang, A. van Wyk, Y. Li, A. J. Lee, and A. Nica. Evolvable View Environment (EVE): Non-Equivalent View Maintenanc e under Schema Changes. In *SIGMOD*, pages 553–555, 1999.

[21] M. Siegel and S. E. Madnick. A Metadata Approach to Resolving Semantic Conflicts. In *VLDB*, pages 133–145, 1991.

[22] D. Srivastava and Y. Velegrakis. MMS: Using Queries As Data Values for Metadata Management (Deomonstration paper). In *ICDE*, Apr. 2007.

[23] M. Stonebraker, J. Anton, and E. N. Hanson. Extending a Database System with Procedures. *ACM TODS*, 12(3):350–376, 1987.

[24] Y. Velegrakis, R. J. Miller, and J. Mylopoulos. Representing and Querying Data Transformations. In *ICDE*, pages 81–92, 2005.

[25] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping Adaptation under Evolving Schemas. In *VLDB*, pages 584–595, 2003.

[26] W3C. Resource description framework. http://www.w3.org/RDF.

[27] R. Wang, M. P. Reddy, and H. B. Kon. Toward Quality Data: an Attribute-based Approach. *Decision Support Systems Journal*, 13(3-4):349–372, 1995.

[28] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *CIDR*, pages 262–276, 2005.

[29] C. M. Wyss and E. L. Robertson. Relational languages for Metadata Integration. *ACM TODS*, 30(2):624–660, 2005.