An Annotation Management System for Relational Databases

Deepavali Bhagwat Laura Chiticariu Wang-Chiew Tan* Gaurav Vijayvargiya

University of California, Santa Cruz
Email: {dbhagwat, laura, wctan, gaurav}@cs.ucsc.edu

Abstract

We present an annotation management system for relational databases. In this system, every piece of data in a relation is assumed to have zero or more annotations associated with it and annotations are propagated along, from the source to the output, as data is being transformed through a query. Such an annotation management system is important for understanding the provenance and quality of data, especially in applications that deal with integration of scientific and biological data.

We present an extension, pSQL, of a fragment of SQL that has three different types of annotation propagation schemes, each useful for different purposes. The *default* scheme propagates annotations according to where data is copied from. The *default-all* scheme propagates annotations according to where data is copied from among *all* equivalent formulations of a given query. The *custom* scheme allows a user to specify how annotations should propagate. We present a storage scheme for the annotations and describe algorithms for translating a pSQL query under each propagation scheme into one or more SQL queries that would correctly retrieve the relevant annotations according to the specified propagation scheme. For the default-all scheme, we also show how we generate *finitely* many queries that can simulate the annotation propagation behavior of the set of all equivalent queries, which is possibly infinite. The algorithms are implemented and the feasibility of the system is demonstrated by a set of experiments that we have conducted.

1 Introduction

For many scientific domains, new databases are often created to support the data analysis needs of domain-specific scientists. Some examples of such databases from biology include UniProt [2] and SWISS-PROT [3]. Data that is collected from other sources is often cleansed and reformatted before it is compiled into the new database. Very often, the newly created database will also contain new analysis or results that are derived by scientists. By associating old and new data together in the new database, an integrated perspective is provided to scientists and this is critical for further analysis and scientific discovery. With the proliferation of many such inter-dependent databases[1], it is natural to ask what is the provenance of a piece of data (i.e., where that piece of data is copied or created from) in a database. Understanding the provenance of data is important towards understanding the quality of data which may help, for example, a scientist to decide on the amount of trust to place on a piece of information that she encounters in a database. We use the term *annotations* to mean information about data such as provenance, comments, or other types of metadata.

We describe an annotation management system for relational databases where annotations may be attached to a piece of data and are transparently carried along as data is being transformed. One immediate application is to use annotations to systematically trace the provenance and flow of data: if we attach to every piece of source data an annotation that describes its address (i.e., origins), then the annotations of a piece of data in the output of a transformation describe its provenance. Even if the data had undergone several transformation steps, we can easily determine the origins (or the flow of data for that matter) through the transformation steps by examining the annotations. Another use of annotations is to describe information about data that would otherwise have not been kept in a database. For example, an error report or remarks about a piece of data may be attached and propagated along to other databases, thus notifying other users of the error or additional information. The quality or security level of a piece of data can also be described in annotations. Since annotations are propagated along as a query is executed, the annotations on the result of a query can be aggregated to determine the quality or degree of sensitivity of the resulting output. This idea of using annotations to describe the security level of various data items or to specify fine-grained access control policies is not new and can be found in various forms in existing literature [11, 13, 19].

We describe three propagation schemes for propagating

*Supported in part by an NSF CAREER Award IIS-0347065.

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

[1] See [10] for a catalog of biology databases.

annotations that are motivated by different needs. They correspond to the default, default-all, and custom propagation schemes. The *default* scheme uses provenance as the basis for propagating annotations. If an output piece of data d' is copied from an input piece of data d, then the annotations associated with d are propagated to d'. A piece of output data d' is *copied from* an input piece of data d if d' is created from d according to the syntax and evaluation of the query. Although this natural definition corresponds intuitively to how people reason about provenance, the way annotations are propagated is dependent on the way a query is written. As shown in [24], two equivalent queries may propagate annotations differently. While this behavior may seem disturbing at first, in many applications including those described above, such an automatic provenance-based annotation propagation scheme which allows one to trace where data is copied from or copied to based on a given query is still very desirable. Indeed, similar ideas were proposed before in [18, 26]. An alternative method of propagating annotations, called the *default-all* scheme, is to propagate annotations according to where data is copied from in *all* equivalent formulations of the given query since one may be interested in obtaining all relevant annotations of a piece of data in the output regardless of how a query may have been written. Unlike the default scheme, two equivalent queries will always propagate annotations in the same way under this scheme. In some cases, a user may only be interested in annotations provided by a certain trusted data source. Hence we also have a third propagation scheme, the *custom* propagation scheme, where the user is free to specify how annotations should be propagated.

Summary of results We have implemented all three propagation schemes in our annotation management system by extending a fragment of SQL. We call this extension pSQL. A pSQL query is essentially an SQL query extended with a PROPAGATE clause that would propagate annotations according to one of the schemes described above as data is transformed. In our implementation, we assume that there is an "additional column" that stores the annotations for every attribute of every relation. A translation algorithm translates a given pSQL query into one or more SPJ queries against these underlying relations and these SPJ queries will retrieve the relevant annotations according to the specified propagation scheme. In the default-all scheme, we are required to propagate annotations according to every possible equivalent reformulations of a given query. At first sight, the default-all scheme seems impossible to implement as there are infinitely many equivalent reformulations of a given query. We show, however, that it is always possible to find a finite set of equivalent queries whose annotation propagation behavior is "representative" of all equivalent queries. Hence, by running every query in this finite set and taking the union of resulting tuples and annotations, we are able to obtain the annotated output of the given query under the default-all scheme. We have conducted experiments to evaluate the feasibility of such an annotation management system. Our experimental results indicate that the execution time of a query under any propagation scheme increases only slightly when the number of annotations in a database is doubled. Our results also show that for the queries we executed, the performance of a query under the default-all scheme can be at worst eight times slower than the performance of the same query under the default or no propagation scheme (i.e., SQL query). At best, it runs about twice as slow. For the default scheme, however, the execution times of pSQL queries are comparable to those of SQL queries. On the average, the pSQL queries with default scheme that we experimented with on a 100MB database took around 40% more time to execute than their corresponding SQL queries. For larger databases (500MB and 1GB), the pSQL queries with default scheme took only about 18% more time to execute than their corresponding SQL queries on the average.

Related Work The problem of computing data provenance is not new. Cui, Widom, and Wiener [9] first approached the problem of tracing the provenance of data that is the result of a query applied on a relational database. The solution proposed in [9] was to first generate a "reverse" query Q^r when asked to compute the provenance of an output tuple t in the result of a query Q applied on a database D (i.e., $Q(D)$). The result of applying Q^r on D consists of all combinations of source tuples in D such that each combination of source tuples and Q explain *why* t is in the output of $Q(D)$. The type of provenance studied by [9] is called *why-provenance* according to Buneman, Khanna, and Tan [6]. Additionally, we may also be interested in knowing *where* the values of a tuple t in the result of $Q(D)$ are copied from in D. The latter type of provenance is called *where-provenance* in [6] and it is this type of provenance that we use for determining where annotations are propagated from. In both works [6, 9], a "reverse" query is generated in order to answer provenance. While the reverse query approach works well in general, it requires a reverse query to be generated and evaluated every time the provenance of an output tuple is sought for. Hence if the provenance of a large number of output tuples is required, this may not be the optimal way to compute provenance.

The reverse query approach is what we call the *lazy* approach for computing provenance; a query is generated and executed to compute the provenance only when needed. In this paper, we propose to trade space for time and carry along the provenance of data as data is being transformed. Hence, in this approach, the provenance of data is *eagerly* computed and immediately available in the output. The idea of eagerly computing provenance by forwarding annotations along data transformations is also not new and has been proposed in various forms in existing literature [4, 18, 26]. In fact, our annotation propagation rules which propagate annotations based on where-provenance are similar to those proposed in [26]. In [26], however, only information about which source relations a value is copied from is propagated along. In contrast, our system is flexible in the amount of information that is carried along to the result (i.e., it could be the source relations, or the ex-

act location within the source locations, or a comment on the data).

Numerous annotation systems have been built to support and manage annotations on text and HTML documents [14, 17, 21, 23, 25]. Recently, annotation systems for genomic sequences [5, 12, 16] have also been built. Laliberte and Braverman [17] discussed how to use the HTTP protocol to design a scalable annotation system for HTML pages. Schickler, Mazer, and Brooks [23] discussed the use of a specialized proxy module that would merge annotations from an annotation store onto a Web page that is being retrieved before sending it to the client browser. Annotea [14, 25] is a W3C effort to support annotations on any Web document. Annotations are also stored on annotation servers and XPointer is used for pinpointing locations on a Web document. A specialized client browser that can understand, communicate, and merge annotations residing in the annotation servers with Web documents is used. Phelps and Wilensky [20, 21, 22] also discussed the use of annotations with certain desirable properties on multivalent documents [22] which support documents of different media types, such as images, postscript, or HTML. DAS or Biodas [5, 12] and the Human Genome Browser [16] are specialized annotation systems for genomic sequence data. In almost all of these systems, the design includes multiple distributed annotation servers for storing annotations and data is merged from various sources to display it graphically to an end user. The research of these systems has been focussed on the scalability of design, distributed support for annotations, or other added features.

We designed and implemented an annotation management system for relational databases where annotations can be made on relational data. This idea was first proposed in [7, 24]. Unlike Web pages, the rigid structure of relations makes it easy to describe the exact position where an annotation is attached. Web pages, however, are often retrieved in part or as a whole. Hence, the issue of what annotations to propagate along when a web page is retrieved is straightforward. In contrast, an annotated relation in our system may undergo a complex transformation as a result of executing a query. We are thus concerned with how annotations should propagate when such complex transformations occur. To the best of our knowledge, this is the first implementation of an annotation management system for relational databases that would allow a user to specify how annotations should propagate.

In Section 2, we describe pSQL and the three different propagation schemes. In Section 3, we describe the algorithm for generating a finite set of queries that can simulate the annotation propagation behavior of all equivalent queries of a given pSQL query. In Section 4, we describe the architecture of our system and a storage scheme for annotations as well as our translation algorithm that rewrites a pSQL query into an SQL query against the underlying storage scheme. In Section 5, we describe our experimental results and in Sections 6 and 7, we conclude with some possible future extensions to our system.

2 pSQL

In our subsequent discussions, we focus on a fragment of SQL that corresponds to conjunctive queries with union [1] (also known as the Select-Project-Join-Union fragment of SQL). We extend this fragment of SQL with a PROPAGATE clause to allow users to specify how annotations should propagate.

Definition 2.1 A *pSQL query* is a query of the form Q_1 UNION ... UNION Q_k, $k > 0$, where each Q_i, $i \in [1, k]$, is a *pSQL query fragment* of the form shown below:

SELECT DISTINCT	*selectlist*
FROM	*fromlist*
WHERE	*wherelist*
PROPAGATE	DEFAULT | DEFAULT-ALL | $r_1.A_1$ TO B_1, ..., $r_n.A_n$ TO B_n

The *fromlist* of a pSQL query fragment is of the form "$R_1\ r_1$, ..., $R_k\ r_k$" where r_i is a tuple variable of the corresponding relation R_i. The *selectlist* of a pSQL query fragment is of the form "$r_1.C_1$ AS D_1, ..., $r_m.C_m$ AS D_m" where r_i is a tuple variable defined in *fromlist*, C_i is an attribute of the relation that corresponds to r_i, and D_i is an attribute name of the output relation. The WHERE clause is optional and the *wherelist* is a conjunction of one or more equalities between attributes of relations or between attributes of relations and constants. The PROPAGATE clause can be defined with DEFAULT, DEFAULT-ALL, or a list of clauses of the form "$r.A$ TO B" definitions where $r.A$ denotes an attribute A of the tuple that is bound to r and B is an attribute among the D_js. ∎

The *SQL query that corresponds to a pSQL query Q* is the SQL query that results when all PROPAGATE clauses in Q have been removed. The meaning of a pSQL query is similar to that of its corresponding SQL query except that annotations are also propagated to each emitted tuple according to the specification given in the PROPAGATE clauses.

Example 2.1 Consider three databases SWISS-PROT (a protein database), PIR (another protein database), and Genbank (a gene database). Each of these databases is modeled as a relation. The schemas and an instance of each relation are shown in Figure 1 (ignore the rest of the relations in the figure). An annotation, shown in braces, is placed on every column of every tuple. Each annotation can be interpreted as the address of the value in the corresponding column of the tuple. An example of a pSQL query with the default propagation scheme is shown below.

Q_1 = SELECT DISTINCT s.ID AS ID, s.Desc AS Desc
FROM SWISS-PROT s
WHERE s.ID = "q229"
PROPAGATE DEFAULT

Intuitively, the default scheme specified in Q_1 propagates annotations of data according to where data is copied from. The result of Q_1 executed against the relation SWISS-PROT is shown in Figure 1. The annotation a_3 is attached to the value q229 in the output since q229 is copied from the ID attribute of the second tuple in SWISS-PROT. Likewise, a_4 in the output is propagated from the

SWISS-PROT

ID	Desc
z131 $\{a_1\}$	AB $\{a_2\}$
q229 $\{a_3\}$	CC $\{a_4\}$
q939 $\{a_5\}$	ED $\{a_6\}$

PIR

ID	Name
p332 $\{a_7\}$	AB $\{a_8\}$
p916 $\{a_9\}$	AB $\{a_{10}\}$

Genbank

ID	Desc
g231 $\{a_{11}\}$	AB $\{a_{12}\}$
g756 $\{a_{13}\}$	CC $\{a_{14}\}$

Result of Q_1:

ID	Desc
q229 $\{a_3\}$	CC $\{a_4\}$

Mapping_Table

entryid	swissprot	pir	genbank
1 $\{a_{15}\}$	z131 $\{a_{16}\}$	p332 $\{a_{17}\}$	g231 $\{a_{18}\}$
2 $\{a_{19}\}$	q229 $\{a_{20}\}$	p916 $\{a_{21}\}$	g756 $\{a_{22}\}$
3 $\{a_{23}\}$	q939 $\{a_{24}\}$	p677 $\{a_{25}\}$	g635 $\{a_{26}\}$

Result of Q_2:

ID	Name
p332 $\{a_7\}$	AB $\{a_8, a_{10}\}$
p916 $\{a_9\}$	AB $\{a_8, a_{10}\}$

Result of Q_3:

ID	Desc
g231 $\{a_{11}, a_{12}\}$	AB
g756 $\{a_{13}, a_{14}\}$	CC

Figure 1: Three protein databases, a mapping table, and three annotated outputs.

annotation of the Desc attribute of the second tuple in SWISS-PROT. ∎

While the default scheme is a natural scheme for propagating annotations, this scheme is not "robust" in that two equivalent queries that return the same output may not propagate the same annotations to the output.

Example 2.2 Consider two equivalent SQL queries Q' and Q'' (two queries are equivalent if they produce the same result on every database).

Q' = `SELECT DISTINCT` p.ID `AS` ID, p.Name `AS` Name
`FROM` PIR p, Mapping_Table m
`WHERE` p.ID = m.pir

Q'' = `SELECT DISTINCT` m.pir `AS` ID, p.Name `AS` Name
`FROM` PIR p, Mapping_Table m
`WHERE` p.ID = m.pir

The results of running Q' and Q'' under the default propagation scheme are shown below.

Result of Q':

ID	Name
p332 $\{a_7\}$	AB $\{a_8\}$
p916 $\{a_9\}$	AB $\{a_{10}\}$

Result of Q'':

ID	Name
p332 $\{a_{17}\}$	AB $\{a_8\}$
p916 $\{a_{21}\}$	AB $\{a_{10}\}$

For Q', the annotations for the ID column are from the PIR table while for Q'', the annotations for the ID column are from the Mapping_Table. ∎

While it is likely that a user will realise that Q' will generate a different annotated outcome from Q'' in general, the situation is not so straightforward for more complex queries. The above example motivates the need for a propagation scheme that is invariant under equivalent queries. One should be able to retrieve all relevant annotations about a piece of output data regardless of how the query is written, if desired. The default-all propagation scheme propagates annotations according to where data is copied from among all equivalent formulations of the given query. Hence the annotated outcome is the same for equivalent queries under this scheme. In case a user prefers to retrieve annotations from one source over another, the user is also free to specify how annotations should propagate in the custom scheme.

Example 2.3 The queries Q_2 and Q_3 are examples of pSQL queries with the default-all and custom propagation schemes respectively.

Q_2 = `SELECT DISTINCT` p.ID `AS` ID, p.Name `AS` Name
`FROM` PIR p
`PROPAGATE DEFAULT-ALL`

Q_3 = `SELECT DISTINCT` g.ID `AS` ID, g.Desc `AS` Desc
`FROM` Genbank g
`PROPAGATE` g.ID `TO` ID, g.Desc `TO` ID

The results of Q_2 and Q_3 are shown in Figure 1. The query Q_2 retrieves all tuples from the PIR table under the default-all propagation scheme. The annotations we get in the result are the combined annotations of results from all equivalent queries. In the custom scheme of Q_3, annotations are propagated according to the given user specification (i.e., g.ID `TO` ID, g.Desc `TO` ID). A clause "g.ID `TO` ID" states that the annotations associated with the value of the ID attribute of the tuple that is currently bound to g should propagate to the ID attribute of the output tuple. Similarly, the annotations associated to the value of the Desc attribute of the tuple that is currently bound to g should propagate to the ID attribute of the output tuple. ∎

Some Terminology A *cell* (or *location*) is a triple (r, t, i) which denotes the ith column of the tuple t in relation r. We sometimes use the attribute name at position i instead of the position i. We also write a cell simply as a pair (t, i) in the context where the relation r is clear. Each cell contains a value of some type. We use $v(c)$ to denote the value at cell c ($v(c)$ is called *a piece of data*). Let $\mathcal{L}$ denote the set of all strings. Each cell c in a database is *associated with* a set of annotations $\{a_1, ..., a_k\}$ where each a_i, $i \in [1, k]$, is an element in $\mathcal{L}$. We also say each a_i, $i \in [1, k]$, is an annotation attached to c. We use the notation $\mathcal{A}(c)$ to denote the set of all annotations attached to cell c.

Containment vs. Annotation-Containment. Two pSQL queries Q and Q' are equivalent, denoted as $Q = Q'$, if for every database D, $Q(D) = Q'(D)$. The query Q is contained in Q', denoted as $Q \subseteq Q'$, for every database D, $Q(D) \subseteq Q'(D)$. Two pSQL queries Q and Q' are *annotation-equivalent*, denoted as $Q =_a Q'$, if Q and Q' produce the same annotated output on all databases. More precisely, this means that for every database D, $Q(D)$ is equal to $Q'(D)$ and the set of annotations $\mathcal{A}((Q(D), t, i))$ is identical to $\mathcal{A}((Q'(D), t, i))$ for every output location (t, i) in $Q(D)$. A pSQL query Q is *annotation-contained* in Q', denoted as $Q \subseteq_a Q'$, if for every database D, $Q(D) \subseteq Q'(D)$ and for every output location (t, i) in $Q(D)$, $\mathcal{A}((Q(D), t, i)) \subseteq \mathcal{A}((Q'(D), t, i))$.

Example 2.4 Figure 1 shows several examples of annotated relations. The value z131 in SWISS-PROT is the value at cell (SWISS-PROT, (z131, AB), ID) which denotes the ID column of tuple (z131, AB) in the SWISS-PROT relation. Note that the attribute names in the tuple (z131, AB) have been omitted. The annotation $\{a_1\}$ is the set of annotations associated with this cell. Hence, $\mathcal{A}$((SWISS-PROT, (z131, AB), ID)) is $\{a_1\}$. In the result of Q_2, $\mathcal{A}$(((p332, AB), Name)) is $\{a_8, a_{10}\}$. ∎

2.1 The Custom Propagation Scheme

We allow the user the flexibility to specify custom propagation schemes using a PROPAGATE clause of the form "$r_1.A_1$ TO B_1, ..., $r_n.A_n$ TO B_n". The semantics of a pSQL query fragment Q with custom propagation scheme is as follows. For every binding μ of tuple variables to tuples in the respective relations according to the *fromlist* of Q such that the conditions in the *wherelist* are satisfied, emit an output tuple t according to the *selectlist*. For every clause "$r_i.A_i$ TO B_i" specified in the PROPAGATE clause, we add the set of annotations at the location (r_i, A_i) to the set of annotations (initially empty) at the output location (t, B_i). Finally, duplicate output tuples are merged. Suppose $t_1, ..., t_k$ are the emitted tuples and $s_1, ..., s_m$ are the tuples that result when duplicate output tuples have been merged. Then, for every output location (s, B), $\mathcal{A}((s, B))$ = $\bigcup_{t_j=s, j\in[1,k]} \mathcal{A}((t_j, B))$. The query Q_3 of Example 2.3 is an example of a pSQL query fragment with a custom propagation scheme; every tuple in Genbank is emitted in such a way that the set of annotations that is associated with the ID column of an output tuple is the union of annotations associated with the ID column and Desc column of the corresponding tuple in Genbank.

2.2 The Default Propagation Scheme

If PROPAGATE DEFAULT is used in a pSQL query fragment, the set of annotations of a piece of output data consists of all the annotations associated with where that piece of data is copied from in the source.

The semantics of a pSQL query fragment Q with the default propagation scheme is as follows. For every binding of tuple variables to tuples in the respective relations according to the *fromlist* of Q such that the conditions in the *wherelist* are satisfied, emit an output tuple t according to the *selectlist* as well as the corresponding sets of annotations for every cell in t. Since every value of an output cell c' in t is generated from some value of an input cell c according to the current bindings, the set of annotations attached to c is also attached to c'. Finally, duplicate output tuples are merged together. Suppose $t_1, ..., t_k$ are the emitted tuples and $s_1, ..., s_m$ are the tuples that result when duplicate output tuples have been merged. Then, for every output location (s, B), $\mathcal{A}((s, B))$ = $\bigcup_{t_j=s, j\in[1,k]} \mathcal{A}((t_j, B))$.

Example 2.5 Suppose we have the following pSQL query where each fragment uses the default propagation scheme.

```
SELECT      Desc AS Desc
FROM        SWISS-PROT
PROPAGATE   DEFAULT
UNION
SELECT      Desc AS Desc
FROM        Genbank
PROPAGATE   DEFAULT
```

Result:

Desc
AB $\{a_2, a_{12}\}$
CC $\{a_4, a_{14}\}$
ED $\{a_6\}$

The first subquery emits an output tuple "AB" with annotations $\{a_2\}$ and the second subquery emits the same output tuple "AB" but with annotations $\{a_{12}\}$. The merged result of these two tuples is a single output tuple "AB" with annotations $\{a_2, a_{12}\}$. This explains the first output tuple in the result. A similar reasoning applies to the rest of the output tuples. ∎

It is easy to see that a pSQL query fragment with default propagation scheme can be translated into a pSQL query fragment with custom propagation scheme. For example, the query Q_1 of Example 2.1 can be rewritten into a pSQL query with custom scheme where the propagate clause is replaced by "PROPAGATE s.ID TO ID, s.Desc TO Desc" since the ID value and Desc value of an output tuple are copied from s.ID and s.Desc, respectively.

2.3 The Default-All Propagation Scheme

A pSQL query with the default propagation scheme is, essentially, an SQL query with annotations propagated based on where a value is retrieved according to the syntax of the query. We have already seen an example of two pSQL queries under the default propagation scheme (Example 2.2) which are equivalent but not *annotation-equivalent*.

This motivates us to define a third propagation scheme, called the default-all scheme, where the annotation propagation behavior of a pSQL query is invariant to the syntax of the query. A pSQL query Q with default-all propagation scheme propagates annotations according to the default propagation behavior of all equivalent formulations of Q. The resulting tuples that are generated by all equivalent queries of Q according to the default scheme are then merged together. Despite the fact that there are infinitely many equivalent formulations of Q, we describe a method that would compute the desired result by examining only a *finite* number of pSQL queries. We call such a finite set of queries a *query-basis of* Q.

Definition 2.2 Let Q denote a pSQL query with default-all propagation scheme. Let $S(Q)$ denote the SQL query that corresponds to Q and let $\mathcal{E}(S(Q))$ denote the set of all pSQL queries Q' under the default propagation scheme such that $S(Q')$ is equivalent to $S(Q)$. A *query basis of* Q, denoted as $\mathcal{B}(Q)$, is a finite set of pSQL queries with default propagation scheme such that $\bigcup_{q\in\mathcal{B}(Q)} q =_a \bigcup_{q\in\mathcal{E}(S(Q))} q$.

We describe next an algorithm that finds a query basis for a pSQL query with default-all propagation scheme. The size of the query basis that the algorithm returns is always polynomial in the size of Q.

3 Generating a Query Basis

The algorithm for computing a query basis for a pSQL query with default-all propagation scheme proceeds by first generating a *representative* query of Q, called Q_0. Intuitively, a *representative* query of Q is a query that is equivalent to Q and for every attribute A that is equal or transitively equal to an attribute B in the *selectlist* of Q, the annotations of A are propagated to B. From Q_0, a finite number of *auxiliary* queries are also generated and these queries, together with Q_0, form a query basis of Q. Each auxiliary query is equivalent to Q but may propagate additional annotations to the output that are not propagated by Q_0. Intuitively, only a finite number of auxiliary queries are needed because only one auxiliary query needs to be generated for each attribute of a relation that "contributes annotations" to the output. In the rest of the discussion, we restrict our language to be pSQL query fragments. We present an algorithm for generating a query basis of a pSQL query fragment with default-all propagation scheme. The algorithm can be extended to handle pSQL queries in general and the details are omitted.

Algorithm Generate-Query-Basis
Input: A pSQL query fragment Q with default-all propagation scheme.
Output: A query basis of Q, $\mathcal{B}(Q)$.

Let Q be a pSQL query fragment of the form shown in Definition 2.1 with `PROPAGATE DEFAULT-ALL` clause.

1. *Generate Q_0, the representative query of Q.*
   Generate a query Q_0 that is identical to Q except that the propagation scheme of Q is replaced with the following propagation scheme:

   For every attribute "$r.A$ AS C" in the *selectlist*, add "$r.A$ TO C" in the PROPAGATE clause.

   For every attribute "$r.A$ AS C" in the *selectlist* and every attribute $s.B$ that is equal to $r.A$ or transitively equal to $r.A$ according to the *wherelist*: add "$s.B$ TO C" in the PROPAGATE clause.

   (The effect is that all attributes that are equal to an attribute C in the *selectlist* have their annotations propagated to C.)

2. *Generate auxiliary queries of Q_0.*
   Initialize $\mathcal{B}(Q)$ to the empty set. Add Q_0 to $\mathcal{B}(Q)$. For every attribute "$r.A$ AS C" in the *selectlist* of Q_0 and every "$s.B$ TO C" in the PROPAGATE clause of Q_0, do the following:

   Create a query Q' that is identical to Q_0. Suppose s is a tuple variable of relation S according to the *fromlist* of Q_0. Add "$S\ s'$" to the *fromlist* of Q' where s' is a tuple variable that does not occur in Q'. Add "$s'.B = s.B$" to the *wherelist* of Q' and "$s'.B$ TO C" to the PROPAGATE clause of Q'. Add Q' to $\mathcal{B}(Q)$.

   (The auxiliary query Q' is equivalent to Q but may carry additional annotations to the output.)

3. *Return $\mathcal{B}(Q)$.*

Example 3.1 Consider the three databases, SWISS-PROT, PIR, and Genbank along with a Mapping_table that contains the correspondences between identifiers of genes and proteins in the three databases in Figure 1. Such mapping tables commonly occur in integrating many sources with overlapping information [15]. Suppose we have the following query Q that integrates information from SWISS-PROT and PIR.

```
SELECT DISTINCT t.swissprot AS ID,
       p.Name AS Name, s.Desc AS Desc
FROM Mapping_Table t, SWISS-PROT s, PIR p
WHERE t.swissprot = s.ID AND t.pir = p.ID
PROPAGATE DEFAULT-ALL
```

After Step 1 of the above algorithm, we obtain the following representative query Q_0:

```
SELECT DISTINCT t.swissprot AS ID,
       p.Name AS Name, s.Desc AS Desc
FROM Mapping_Table t, SWISS-PROT s, PIR p
WHERE t.swissprot = s.ID AND t.pir = p.ID
PROPAGATE t.swissprot TO ID, s.ID TO ID,
       p.Name TO Name, s.Desc TO Desc
```

Note that the annotations of t.swissprot and s.ID will propagate to the output ID column according to Q_0. The second step of the algorithm generates four auxiliary queries. The first query is shown below and the rest are shown in Figure 2.

Q_1 =
```
SELECT DISTINCT t.swissprot AS ID,
       p.Name AS Name, s.Desc AS Desc
FROM Mapping_Table t, SWISS-PROT s, PIR p, Mapping_Table t'
WHERE t.swissprot = s.ID AND t.pir = p.ID,
      t'.swissprot = t.swissprot
PROPAGATE t.swissprot TO ID, s.ID TO ID,
       p.Name TO Name, s.Desc TO Desc,
       t'.swissprot TO ID
```

The query Q_1 is different from Q_0 only in the additional highlighted terms shown in Q_1. There is an extra relation, condition, and propagation in the FROM, WHERE, and PROPAGATE clauses respectively. It is easy to verify that the SQL queries of Q_0 and Q_1 are equivalent. There is a homomorphism h from the tuple variables of Q_1 to those of Q_0 such that h maps the *fromlist* of Q_1 to a subset of the *fromlist* of Q_0 and the conditions in the *wherelist* of Q_0 imply the conditions in the *wherelist* of Q_1 under h. Furthermore, h maps the *selectlist* of Q_1 to the *selectlist* of Q_0. There is also a homomorphism in the reverse direction. Similarly, Q_2, Q_3, and Q_4 of Figure 2 are each equivalent to Q_0. ∎

Intuitively, the representative query Q_0 propagates annotations according to where data is copied from and also where data could have been equivalently copied from. The reason why Q_0 is generated becomes clearer if we represent Q using conjunctive query-like notation

$$A(\mathbf{x}) : -S_1(\mathbf{y_1}), ..., S_n(\mathbf{y_n}), \textit{equalities}.$$

where $\mathbf{x}, \mathbf{y_i}, i \in [1, n]$, denote vectors of variables and every variable in $\mathbf{x}$ occurs in $\mathbf{y_i}$ for some $i \in [1, n]$ and *equalities* is a list of zero of more $y = y'$ clauses where y is a variable that occurs amongst $\mathbf{y_i}$s and y' is a constant. The variables in $\mathbf{x}$ are called *distinguished variables*. Each subgoal corresponds to a relation in the *fromlist* of Q. The

Q_2 =
```
SELECT DISTINCT
        t.swissprot AS ID,
        p.Name AS Name,
        s.Desc AS Desc
FROM Mapping_Table t, SWISS-PROT s,
     PIR p, SWISS-PROT s'
WHERE t.swissprot = s.ID AND
        t.pir = p.ID AND s'.ID = s.ID
PROPAGATE t.swissprot TO ID,
        s.ID TO ID, p.Name TO Name,
        s.Desc TO Desc, s'.ID TO ID
```

Q_3 =
```
SELECT DISTINCT
        t.swissprot AS ID,
        p.Name AS Name,
        s.Desc AS Desc
FROM Mapping_Table t, SWISS-PROT s,
     PIR p, SWISS-PROT s'
WHERE t.swissprot = s.ID AND
        t.pir = p.ID, s'.Desc = s.Desc
PROPAGATE t.swissprot TO ID,
        s.ID TO ID, p.Name TO Name,
        s.Desc TO Desc, s'.Desc TO Desc
```

Q_4 =
```
SELECT DISTINCT
        t.swissprot AS ID,
        p.Name AS Name,
        s.Desc AS Desc
FROM Mapping_Table t, SWISS-PROT s,
     PIR p, PIR p'
WHERE t.swissprot = s.ID AND
        t.pir = p.ID, p'.Name = p.Name
PROPAGATE t.swissprot TO ID,
        s.ID TO ID, p.Name TO Name,
        s.Desc TO Desc, p'.Name TO Name
```

Figure 2: Some of the auxiliary queries generated by Step 2 of Generate-Query-Basis on Example 3.1.

equalities between attributes in the *wherelist* of Q are represented by using the same variable in the respective positions of relations in the conjunctive query-like representation of Q. An equality between an attribute and constant is written out as *equalities*. The head of the query $A(\mathbf{x})$ represents the *selectlist* of Q. We use $C(Q)$ to denote the conjunctive query-like representation of the SQL query that corresponds to Q. For example, $C(Q)$ of Example 3.1 can be written as

$A_0(x, y, z)$:- Mapping_Table(w, x, u, v), SWISS-PROT(x, z), PIR(u, y).

Similar to the semantics of pSQL queries with the default propagation scheme, annotations are propagated according to where data is copied from for such queries [24] by tracing the occurrence of distinguished variables in the query. For example, by tracing the occurrence of the variable x in the query A_0, we can conclude that the annotations in the first column of an output tuple t is obtained from the annotations of the second column of a tuple in Mapping_Table and the first column of a tuple in SWISS-PROT that created t. A similar argument applies to the variables y and z in A_0. Hence, the representative query Q_0 of Example 3.1 is annotation-equivalent to A_0.

Proposition 3.1 *The representative query Q_0 that is generated by Generate-Query-Basis(Q) is annotation-equivalent to $C(Q_0)$.*

In Step 2 the algorithm generates one query for every position in the body where a distinguished variable occurs in A_0. For example, the following four auxiliary queries, in conjunctive query notation, are generated based on A_0. They are annotation-equivalent to the pSQL query fragments $Q_1, ..., Q_4$ shown in Example 3.1 and Figure 2, respectively.

$A_1(x, y, z)$:- Mapping_Table(w, x, u, v), SWISS-PROT(x, z), PIR(u, y), **Mapping_Table(**w_1, x, w_2, w_3**)**.

$A_2(x, y, z)$:- Mapping_Table(w, x, u, v), SWISS-PROT(x, z), PIR(u, y), **SWISS-PROT(**x, w_1**)**.

$A_3(x, y, z)$:- Mapping_Table(w, x, u, v), SWISS-PROT(x, z), PIR(u, y), **SWISS-PROT(**w_1, z**)**.

$A_4(x, y, z)$:- Mapping_Table(w, x, u, v), SWISS-PROT(x, z), PIR(u, y), **PIR(**w_1, y**)**.

Proposition 3.2 *For every query Q' in the result of Generate-Query-Basis(Q) (denoted as $\mathcal{B}(Q)$), $C(Q')$ is annotation-contained in $\bigcup_{q \in \mathcal{B}(Q)} q$.*

Each auxiliary query carries annotations to the output that may have been missed by the representative query of Q. We shall show next that the set of pSQL query fragments in $\mathcal{B}(Q)$ generated by the algorithm is a query basis for Q. We first prove the following lemma.

Lemma 3.1 *Let $\mathcal{B}(Q)$ denote the result of Generate-Query-Basis(Q) where Q is a pSQL query fragment and let Q' denote a pSQL query fragment under the default propagation scheme. If Q' is equivalent to Q, then Q' is annotation-contained in $\bigcup_{q \in \mathcal{B}(Q)} q$.*

Proof. We know from Proposition 3.1 that the representative query Q_0 that is generated at Step 1 of the algorithm is annotation-equivalent to the conjunctive query representation of the SQL query that corresponds to Q, $C(Q)$. We can also easily verify that $Q' \subseteq_a C(Q')$. Since $C(Q)$ and $C(Q')$ are equivalent queries, the minimal queries of $C(Q)$ and $C(Q')$ are identical up to variable renaming. For convenience, we shall assume that the minimal queries are identical in the form shown below. We also assume that there are no equalities between variables and constants, for convenience.

$C(Q)$: $H(\mathbf{x})$:- *minpart, rest1*.
$C(Q')$: $H(\mathbf{x})$:- *minpart, rest2*.

The subgoals denoted by *minpart* are the subgoals in the minimal query of $C(Q)$ or $C(Q')$ and *rest1* and *rest2* denote the rest of the subgoals in $C(Q)$ and $C(Q')$, respectively. Our proof makes use of an earlier result in [24] extended for unions of conjunctive queries. Given a conjunctive query Q, we use the notation $Q[0]$ to denote the head of Q, the notation $Q[i]$, $i > 0$, to denote the ith subgoal of Q, and var($Q[i]$) to denote the list of variables of the ith subgoal of Q.

Fact 1 ([24]) Given two unions of conjunctive queries $Q = \bigcup_{i=1}^{m} Q_i$ and $Q' = \bigcup_{j=1}^{n} Q'_j$, $Q \subseteq_a Q'$ if and only if for every Q_r, $r \in [1, m]$, and every variable x that occurs at both the ith position of var($Q_r[0]$) and the jth position of var($Q_r[p]$) for some p, there exists a homomorphism h from Q'_s (for some $s \in [1, n]$) to Q_r such that

1. h maps the body of Q'_s into the body of Q_r and the head of Q'_s to the head of Q_r, and
2. the variable that occurs at the jth position of the qth subgoal of Q'_s (i.e., var($Q'_s[q]$)[j]) is identical to the

variable at the ith position of the head of Q'_s (i.e., var($Q'_s[0]$)[i]), where $Q'_s[q]$ is a pre-image of $Q_r[p]$ under h. That is, for some subgoal q, var($Q'_s[q]$)[j] = var($Q'_s[0]$)[i] and $h(Q'_s[q]) = Q_r[p]$.

We shall show that for every distinguished variable x at the ith position in the head of $C(Q')$ and its occurrence at the jth position of the pth subgoal $S(\mathbf{u})$ (i.e., the jth variable of $\mathbf{u}$ is x) in the body of $C(Q')$, there is a generated query Q_g in $\mathcal{B}(Q)$ and a homomorphism $h : C(Q_g) \rightarrow C(Q')$ that satisfies the conditions (1) and (2) stated in the fact. Then by the above fact, we have $C(Q') \subseteq_a C(Q_g)$. We know that $C(Q_g) \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} q$ from Proposition 3.2. Therefore $C(Q') \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} q$. Since $Q' \subseteq_a C(Q')$ and $C(Q') \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} q$, we have $Q' \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} q$.

Let x be a distinguished variable at the ith position in the head of $C(Q')$ and suppose x occurs at the jth position of the pth subgoal $S(\mathbf{u})$ of $C(Q')$. If $S(\mathbf{u})$ is among the subgoals in the *minpart* of $C(Q')$, then it must also be among the subgoals in the *minpart* of $C(Q)$. Hence the algorithm Generate-Query-Basis would have generated one or more queries whose combined effect is the query $C(Q_g)$, shown below,

$H(\mathbf{x})$:- *minpart*, *rest1*, $S(\mathbf{w_1}, x, \mathbf{w_2})$.

(The variable x occurs at the jth position in the subgoal $S(\mathbf{w_1}, x, \mathbf{w_2})$ and $\mathbf{w_1}$ and $\mathbf{w_2}$ are vectors of distinct variables that do not occur in $C(Q)$.) This corresponds to Step 2 of the algorithm where a new relation S is added to the FROM clause and clauses of the form "B TO A" are added to the PROPAGATE clause to simulate the effect of x propagating annotations to the output. We assume that x occurs under the attribute A in the output and B is the attribute name of x in S in the named perspective. If x occurs under another attribute D in the output of $C(Q_g)$, there will be another query generated by Step 2 of the algorithm that propagates the annotations of B to D. Hence, there is possible more than one pSQL query whose combined annotation propagation effect equals that of $C(Q_g)$.

It is easy to see that there is a homomorphism from $C(Q_g)$ to $C(Q')$ with the desired properties required by the fact shown above. The homomorphism is obtained by extending the homomorphism $h' : C(Q) \rightarrow C(Q')$ which we know exists since $C(Q) = C(Q')$. The homomorphism h' is extended to h'' by mapping the ith variable in $\mathbf{w_1}$ to the corresponding ith variable in $\mathbf{u}$ and the ith variable in $\mathbf{w_2}$ to the $(j+i)$th variable in $\mathbf{u}$ (this is possible since $\mathbf{w_1}$ and $\mathbf{w_2}$ are distinct variables). Clearly, h'' satisfies the conditions required by the above fact. If $S(\mathbf{u})$ are among the subgoals in *rest2* of $C(Q')$, we first claim that a subgoal $S(\mathbf{u'})$, where the jth variable of u' is x, must also occur among subgoals in the *minpart* of Q'. With this, a similar argument presented before shows that there must be a homomorphism from a query $C(Q_g)$ to $C(Q')$ with the desired conditions required by the above fact and hence, $C(Q') \subseteq_a C(Q_g)$. Since the annotation propagation behavior of $C(Q_g)$ is equal to the combined annotation propagation effect of one or more queries in $\bigcup_{q \in \mathcal{B}(Q)} q$, we have $C(Q_g) \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} q$.

Figure 3: Architecture of our system.

We show next that if $S(\mathbf{u})$ are among the subgoals in *rest2* of $C(Q')$, there must exist such a subgoal $S(\mathbf{u'})$ among the *minpart* of $C(Q')$. Since there is a homomorphism g from $C(Q')$ to the minimal query of $C(Q')$ and $g(x) = x$ (since x is a distinguished variable), this implies that there must be a subgoal $S(...x...)$ among the subgoals in the *minpart* of $C(Q')$ such that x occurs at the jth position of this subgoal. We therefore conclude that $S(\mathbf{u'})$ exists. ∎

Theorem 3.2 *Let Q be a pSQL query fragment with default-all propagation scheme. The algorithm Generate-Query-Basis(Q) returns a query basis of Q.*

Proof. Let $\mathcal{E}(Q)$ denote the set of pSQL query fragments q under the default propagation scheme such that the SQL query that corresponds to q is equivalent to that of Q (i.e., $S(q) = S(Q)$). Let $\mathcal{B}(Q)$ denote the result of running the algorithm Generate-Query-Basis on Q. By Lemma 3.1, $\bigcup_{q \in \mathcal{E}(Q)} q \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} q$. Since $\mathcal{B}(Q) \subseteq \mathcal{E}(Q)$, we immediately have $\bigcup_{q \in \mathcal{B}(Q)} q \subseteq_a \bigcup_{q \in \mathcal{E}(Q)} q$ and hence the result. ∎

Proposition 3.3 *Given a pSQL query fragment Q with the default-all propagation scheme, the number of queries returned by Generate-Query-Basis(Q) is polynomial in the size of Q. Furthermore, each query in Generate-Query-Basis(Q) is polynomial in the size of Q.*

An optmization Observe that the auxiliary pSQL queries overlap significantly in the PROPAGATE clauses (e.g., see Figure 2); they differ only in the last (highlighted) propagation. In fact, we show that the non-highlighted propagations in the auxiliary queries are unnecessary (the details are omitted). Intuitively, they are unnecessary because these propagations are identical to the propagations of the representative query Q_0. Hence, in our *optimized* implementation of Generate-Query-Basis, these non-highlighted propagations are not generated in the auxiliary queries. We refer to our original implementation of algorithm Generate-Query-Basis as the *unoptimized* implementation.

4 System Architecture

The architecture of our Annotation Management System is illustrated in Figure 3. We have two main modules: the translator module and the postprocessor module. The translator module takes as input a pSQL query and returns as output an SQL query (i.e., a union of SPJ queries) which is sent to the RDBMS. The SQL query is then executed by the RDBMS. The tuples that are returned by the RDBMS are sorted in a certain order and sent to the postprocessor module which merges annotations of identical cells of duplicate tuples together in one pass through the returned tuples.

4.1 A Naive Storage Scheme

At present, we store our annotations using a naive storage scheme: we assume that every attribute A of a relation scheme R has an extra column A_a that will be used to store annotations. We denote this new relation with extra columns as R'. For example, a relation $R(A, B)$ will be represented as $R'(A, A_a, B, B_a)$ in the naive storage scheme. Given a tuple t in a relation of R, if $\{a_1, ..., a_k\}$ are the annotations associated with the location (t, A), then there will be k tuples $t_1, ..., t_k$ in R' such that $t_i.A_a = a_i$ for $i \in [1, k]$ and $t_i.R = t$, $i \in [1, k]$ where $t_i.R$ denotes the projection of t_i on the attributes of R. For convenience, we sometimes use the relation name R to refer to R'. As an example, the two instances of R shown below are both valid representations of the tuple (a $\{a_1, a_2\}$, b $\{b_1\}$).

A	A_a	B	B_a
a	a_1	b	b_1
a	a_2	b	–

A	A_a	B	B_a
a	a_1	b	–
a	a_2	b	–
a	a_2	b	b_1

Observe that a query returns the same result regardless of the underlying storage instance used. In the case where every cell has a distinct annotation that denotes its address, then one could define R as a view definition of R using the internal row identifier used in many database systems such as Oracle and Postgres.

4.2 The Translator

The translator module takes as input a pSQL query Q and translates Q to an SQL query Q' against the naive storage scheme. A pSQL query with default or default-all propagation scheme is first reformulated into one with a custom propagation scheme. A pSQL query with the custom propagation scheme is reformulated into an SQL query (i.e., a union of SPJ queries). The algorithm for reformulating a pSQL query fragment with default propagation scheme into a pSQL fragment with custom propagation scheme is described briefly at the end of Section 2.2. The algorithm for reformulating a pSQL query fragment with default-all propagation scheme into a pSQL query fragment with custom propagation scheme is described by the Generate-Query-Basis algorithm in Section 2.3. We describe next the algorithm for reformulating a pSQL query with custom propagation scheme into an SQL query.

Algorithm Custom-pSQL-To-SQL

Input: A pSQL query fragment Q with custom propagation scheme.

Output: An SQL query Q_s written against the naive schema.

Let Q be a pSQL query fragment of the form shown in Definition 2.1 with a *custom-propagatelist*.

1. *Generate intermediate SQL queries.* Each intermediate SQL query retrieves annotations (as much as possible) from the naive schema according to the given query Q.

   Let Q_0 be a query that is identical to Q except that it does not have the PROPAGATE clause of Q.

   For each output attribute C of Q, create an empty bin for C. Denote this bin as bin(C). For each propagate clause "$s.B$ TO C" in the *custom-propagatelist* of Q, add "$s.B_a$ AS C_a" to bin(C).

   Let $\mathcal{Q}$ be the empty set of SQL queries. Repeat until all bins are empty:

   Let Q' be a query that is identical to Q_0. For each output attribute C of Q, if bin(C) is nonempty, remove a clause "$s.B_a$ AS C_a" from bin(C) and add it to the *selectlist* of Q'. If bin(C) is empty, we add "NULL AS C_a" to the *selectlist* of Q'. Add Q' to $\mathcal{Q}$.

2. *Generate a wrapper SQL query Q_s for $\mathcal{Q}$.*

   SELECT DISTINCT *
   FROM (Q_1 UNION $\cdots$ UNION Q_n)
   ORDER BY *orderbylist*

   where $\mathcal{Q} = \{Q_1, ..., Q_n\}$ and *orderbylist* is the list of all output attributes in the *selectlist* of Q. The *orderbylist* is required so that the Postprocessor can merge annotations of identical tuples together with one pass over the result of Q_s.

3. *Return Q_s.*

Example 4.1 Consider the SWISS-PROT relation of Figure 1 and assume that there is an extra attribute Size. Suppose we have the following pSQL query Q with custom propagation scheme written against SWISS-PROT:

```
SELECT     s.ID AS ID, s.Desc AS Desc, s.Size AS Size,
FROM       SWISS-PROT s
PROPAGATE  s.ID TO Desc, s.Desc TO Desc,
           s.Size TO Size,
```

Observe that every tuple in SWISS-PROT will be emitted in such a way that the set of annotations associated with the Desc column of a tuple in the output is the union of annotations associated with both ID and Desc of the corresponding tuple in SWISS-PROT. Furthermore, the annotations associated with the Size column of a tuple are the same annotations associated with the Size column of the corresponding tuple in SWISS-PROT and the column ID of every tuple in the output does not carry any annotations.

In step 1 of algorithm Custom-pSQL-To-SQL, the following two intermediate SQL queries are generated since bin(ID) is empty, bin(Desc) = { s.ID$_a$ AS Desc$_a$, s.Desc$_a$ AS Desc$_a$} and bin(Size) = { s.Size$_a$ AS Size$_a$ }.

```
Q1 =  SELECT   s.ID AS ID, NULL AS ID_a
                s.Desc AS Desc, s.ID_a AS Desc_a,
                s.Size AS Size, s.Size_a AS Size_a,
      FROM     SWISS-PROT s
Q2 =  SELECT   s.ID AS ID, NULL AS ID_a
                s.Desc AS Desc, s.Desc_a AS Desc_a,
                s.Size AS Size, NULL AS Size_a,
      FROM     SWISS-PROT s
```

In step 2, the algorithm generates the following wrapper SQL query:

```
Qs =  SELECT DISTINCT  *
      FROM             (Q1 UNION Q2)
      ORDER BY         ID, Desc, Size
```

Observe that Q_1 and Q_2 are unioned and the result is sorted according to the attributes in the *selectlist* of Q. The tuples are sorted according to the *selectlist* of Q so that the Postprocessor can merge annotations associated with identical cells in the output of Q in one pass over the result

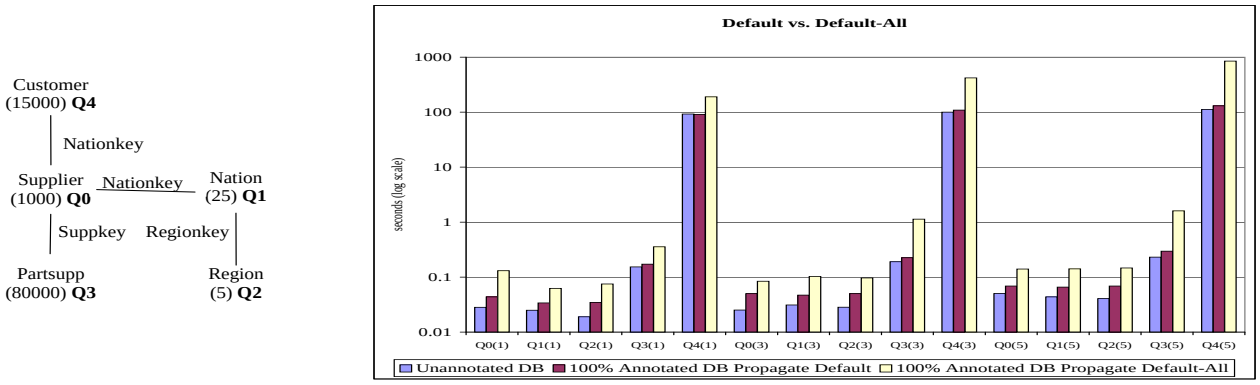

Figure 4: Queries used in our experiments and comparison in performance for 100MB, 100% annotated TPCH database.

of Q_s. Observe also that the number of SQL queries in $\mathcal{Q}$ is equal to the maximum bin size. ∎

4.3 The Postprocessor

The Postprocessor scans the set of tuples returned by the RDBMS and unions together the annotations from duplicate tuples for proper display. This operation is done in linear time in the number and size of tuples retrieved, provided that the set of emitted tuples is already sorted. For example, if the postprocessor receives the first table of Section 4.1 as input, it returns { (a $\{a_1, a_2\}$, b $\{b_1\}$) }.

5 Experimental Evaluation

We conducted several experiments to evaluate the feasibility of our annotation management system. Our main goal is to compare the performance of queries under different propagation schemes (default, default-all, or no propagation scheme (i.e., SQL queries)) and to compare the performance of queries when the number of annotations in a database is varied.

Setup We have implemented our system on top of Oracle 9i Enterprise Edition. For our experiments we used 100MB TPCH database (and subsequently 500MB and 1GB TPCH databases), which we call unannotated database. We have also modified TPCH schema to conform to our naive storage scheme by adding an additional attribute for every attribute of every relation in the TPCH schema. We have created three different instances of the modified TPCH database schema corresponding to 30%, 60% and 100% annotated databases. A 30% annotated database means 30% of the total number of cells in every relation instance of the database will contain one annotation. We ran queries of increasing join sizes to determine how well our system scales for this type of queries. (We did not use TPCH queries in our experiments because they include aggregates and nested queries.) The queries $Q_0, ..., Q_4$ denote queries with zero to four joins, respectively, and are shown on the left of Figure 4. For example, Q_2 denotes the query $\text{Supplier} \bowtie \text{Nation} \bowtie \text{Region}$ with two joins, on the attributes Nationkey and Regionkey respectively. The cardinality of each relation is shown in brackets. Our experiments are conducted on a Pentium 4, 2.8GHz machine with 1GB RAM.

Experiments We first measure the performance of our system for queries under the default and default-all propagation scheme on the 100% annotated database. We have implemented and tested both optimized as well as unoptimized versions of our Generate-Query-Basis algorithm. For space reasons we present only our results obtained with the optimized version, as we observed that it consistently and significantly outperforms the unoptimized version. We executed queries $Q_i(1), Q_i(3), Q_i(5), i \in [0, 4]$, which denote queries with i joins and one, three, and five output attributes, respectively. We also executed the SQL query that corresponds to each of these queries on the unannotated database. The results are shown in Figure 4.

Figure 4 illustrates the execution time (the total time taken by the translator, RDBMS, and postprocessor to emit all tuples in the result) of each query for the default and default-all propagation schemes for the 100MB and 100% annotated TPCH database. As expected, the execution time of each query under the default scheme (respectively, default-all scheme) increases slightly as more output attributes are emitted (see, for instance, $Q_0(1)$, $Q_0(3)$, and $Q_0(5)$). The increase in time is due to longer execution time taken by Oracle as well as additional overhead incurred in postprocessing, as more attributes of different tuples need to be compared. Additionally, for the default-all scheme, the number of SPJ queries that are sent to Oracle increases (2, 4, and 6 SPJ queries, respectively) as the number of output attributes increases. Table 1 provides the exact execution times of each query for 100% annotated database and the number of SPJ queries that are generated for the default all-scheme. We note that in the worst case, a query such as $Q_4(5)$ may run about 8 times slower than both the query with default scheme and the actual SQL query. This is not unexpected, however, as there are 6 SPJ queries, each with four joins, that are generated and sent to Oracle for $Q_4(5)$, instead of 1. In the best case (see $Q_4(1)$), a query with default-all scheme runs about twice as slow than the same query with default scheme. We note however that for the default scheme, the execution times of pSQL queries are comparable to those of SQL queries. On

Query	Unannotated	30% Def	30% Def-All	60% Def	60% Def-All	100% Def	100% Def-All	#pSQL	#SPJ
$Q_0(1)$	0.0282	0.0374	0.1316	0.0408	0.125	0.0438	0.1308	2	2
$Q_1(1)$	0.025	0.0344	0.0658	0.034	0.072	0.034	0.0624	2	2
$Q_2(1)$	0.019	0.0312	0.0722	0.0342	0.0748	0.0346	0.075	2	2
$Q_3(1)$	0.1532	0.1752	0.3622	0.1688	0.3594	0.1718	0.356	2	2
$Q_4(1)$	92.4604	92.2198	190.7312	91.7214	190.826	91.2248	190.3552	2	2
$Q_0(3)$	0.0252	0.0468	0.0848	0.0468	0.084	0.05	0.084	4	4
$Q_1(3)$	0.0312	0.0502	0.0968	0.0374	0.0968	0.047	0.103	4	4
$Q_2(3)$	0.0284	0.0502	0.1002	0.0562	0.0998	0.05	0.0968	4	4
$Q_3(3)$	0.191	0.219	1.1186	0.2216	1.1188	0.225	1.1314	4	4
$Q_4(3)$	100.0106	113.4292	422.6232	108.2372	424.6066	109.012	419.5722	4	4
$Q_0(5)$	0.0502	0.069	0.1372	0.072	0.1438	0.069	0.1404	6	6
$Q_1(5)$	0.0438	0.0654	0.138	0.0718	0.1312	0.0658	0.1412	6	6
$Q_2(5)$	0.0406	0.0662	0.1498	0.0658	0.1468	0.0688	0.1466	6	6
$Q_3(5)$	0.231	0.287	1.6128	0.2908	1.6096	0.2968	1.6064	6	6
$Q_4(5)$	111.8918	131.3138	858.8238	130.5282	836.5362	130.6594	850.6284	6	6

Table 1: The execution times of each query for each database and propagation scheme. The columns "#pSQL" and "#SPJ" denote the size of the query basis and number of SPJ queries that are generated, respectively, for the default-all scheme.

the average, the pSQL queries with default scheme that we experimented with took around 40% more time to execute than their corresponding SQL queries, and at best the execution time of a pSQL query with default scheme is the same as the execution time of its corresponding SQL query (e.g., $Q_4(1)$). For larger databases (500MB and 1GB), the pSQL queries with default scheme took only about 18% more time to execute than their corresponding SQL queries on the average (these results are not shown).

Subsequently, we also conducted the same experiments on 30% and 60% 100MB annotated databases. The results are tabulated in Table 1. We observe that the execution time of each query increases only slightly across different databases. For example, the execution time of each query for both default and default-all scheme increases marginally when the number of annotations in the database is doubled from 30% annotations to 60%. We also remark that for the default-all scheme there is no increase in the number of pSQL and SPJ queries that are generated when the number of joins increases because the attributes that are selected do not participate in the joins. The number of pSQL and SPJ queries that are generated increases when the number of output attributes increases and they increase linearly. The execution times of $Q_1(j)$, $j \in [1, 3, 5]$, decreases slightly when compared with $Q_0(j)$ because a join on a small relation has been made.

We also ran the same set of experiments (results are not shown) on 500Mb and 1GB TPCH databases with 30%, 60% and 100% annotations and we observed the same trend as in Figure 4. All our results indicate that the time required to translate the queries is insignificant when compared to the execution time of the queries and the postprocessing time of the queries is proportional to the number and size of emitted tuples. Also, the execution times of default queries are comparable to the performance of SQL queries since only one SPJ is generated.

6 Discussion

So far, our pSQL queries do not allow aggregates and bag semantics (i.e., the `DISTINCT` keyword must be present). We discuss briefly next how we might extend pSQL to handle aggregates and bag queries as well.

Aggregates For the default propagation scheme, if a pSQL query contains aggregates such as count, sum, and average, we assume the semantics that no annotations are associated with the result of these aggregates, since these aggregate values are not copied from any source values. However, for aggregates such as min(a) and max(a), where a is an attribute name, our semantics is that the annotations associated with the location of the resulting min (or max) value are the union of all annotations of the corresponding a-values whose value equals to the min (or max) value. It remains to investigate whether the default-all propagation scheme for pSQL queries with aggregates can be achieved.

Bag semantics It is known from [8] that two conjunctive queries are equivalent under bag semantics if and only if they are isomorphic. This result of [8] implies that to propagate annotations for a pSQL query under the default-all propagation scheme and bag semantics, it suffices to generate only the representative query of that pSQL query in Algorithm Generate-Query-Basis. To handle bag queries, however, the naive storage scheme can no longer be used since the multiplicity of a tuple in this storage scheme depends on the number of annotations that are associated with that tuple. An alternative storage scheme that does not modify the original relation is needed (e.g., store every annotation and its location in a separate relation). To propagate annotations under the default-all propagation scheme and bag semantics for unions of conjunctive queries, however, it remains to first provide a characterization of bag equivalence for unions of conjunctive queries.

7 Conclusion and Future Work

We have described an implementation of an annotation management system where different propagation schemes can be used. Insofar, our system only supports annotations on attributes of tuples. We would like to extend our system to handle annotations on tuples or relations and, in general, to handle annotations on hierarchical data, such as XML. In our current system, annotations are propagated based on where-provenance. In addition, we would like

to extend our system to propagate annotations based on why-provenance, which will provide reasons to why a tuple is in the output. The default-all propagation scheme returns the union of all annotations of an output location returned by all equivalent queries. Conceivably, there could be a complementary propagation scheme that returns the set of all annotations in an output location if it occurs in the same output location in the results of all equivalent queries. It remains to be investigated whether a query basis can be generated for such propagation scheme. The performance of our annotation management system on other storage schemes also needs to be investigated. It would also be interesting to investigate opportunities for optimizations on the generated SQL queries.

Acknowledgements We thank Xinyu Hua for her help during the initial implementation of this system and Ariel Fuxman for helpful suggestions.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley Publishing Co, 1995.

[2] R. Apweiler, A. Bairoch, C. Wu, W. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. Martin, D. Natale, C. O'Donovan, N. Redaschi, and L. Yeh. Uniprot: the universal protein knowledgebase. *Nucleic Acids Research*, 32:D115–D119, 2004.

[3] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research*, 28:45–48, 2000.

[4] P. Bernstein and T. Bergstraesser. Meta-Data Support for Data Transformations Using Microsoft Repository. *IEEE Data Engineering Bulletin*, 22(1):9–14, 1999.

[5] biodas.org. http://biodas.org.

[6] P. Buneman, S. Khanna, and W. Tan. Why and Where: A Characterization of Data Provenance. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 316–330, London, United Kingdom, 2001.

[7] P. Buneman, S. Khanna, and W. Tan. On Propagation of Deletions and Annotations Through Views. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 150–158, Wisconsin, Madison, 2002.

[8] S. Chaudhuri and M. Y. Vardi. Optimization of *real* conjunctive queries. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 59–70, Washington, DC, 1993.

[9] Y. Cui, J. Widom, and J. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.

[10] DBCAT, The Public Catalog of Databases. http://www.infobiogen.fr/services/dbcat/, cited 5 June 2000.

[11] D. E. Denning, T. F. Lunt, R. R. Schell, W. R. Shockley, and M. Heckman. The SeaView Security Model. In *IEEE Symposium on Security and Privacy*, pages 218–233, Washington, DC, 1988.

[12] R. Dowell. A Distributed Annotation System. Technical report, Department of Computer Science, Washington University in St. Louis, 2001.

[13] S. Jajodia and R. S. Sandhu. Polyinstantiation integrity in multilevel relations. In *IEEE Symposium on Security and Privacy*, pages 104–115, Oakland, California, 1990.

[14] J. Kahan, M. Koivunen, E. Prud'Hommeaux, and R. Swick. Annotea: An open rdf infrastructure for shared web annotations. In *Proceedings of the International World Wide Web Conference(WWW10)*, pages 623–632, Hong Kong, China, 2001.

[15] A. Kementseitsidis, M. Arenas, and R. J. Miller. Mapping Data in Peer-to-Peer Systems: Semantics and Algorithmic Issues. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 325–336, San Diego, CA, 2003.

[16] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The Human Genome Browser at UCSC. *Genome Research*, 12(5):996–1006, 2002.

[17] D. LaLiberte and A. Braverman. A Protocol for Scalable Group and Public Annotations. In *Proceedings of the International World Wide Web Conference(WWW3)*, Darmstadt, Germany, 1995.

[18] T. Lee, S. Bressan, and S. Madnick. Source Attribution for Querying Against Semi-structured Documents. In *Workshop on Web Information and Data Management (WIDM)*, Washington, DC, 1998.

[19] A. C. Myers and B. Liskov. A decentralized model for information control. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[20] T. A. Phelps and R. Wilensky. Multivalent Annotations. In *Proceedings of the First European Conference on Research and Advanced Technology for Digital Libraries*, pages 287–303, Pisa, Italy, 1997.

[21] T. A. Phelps and R. Wilensky. Multivalent documents. *Proceedings of the Communications of the Association for Computing Machinery (CACM)*, 43(6):82–90, 2000.

[22] T. A. Phelps and R. Wilensky. Robust intra-document locations. In *Proceedings of the International World Wide Web Conference(WWW9)*, pages 105–118, Amsterdam, Netherlands, 2000.

[23] M. A. Schickler, M. S. Mazer, and C. Brooks. Pan-Browser Support for Annotations and Other Meta-Information on the World Wide Web. In *Proceedings of the International World Wide Web Conference(WWW5)*, Paris, France, 1996.

[24] W. Tan. Containment of relational queries with annotation propagation. In *Proceedings of the International Workshop on Database and Programming Languages (DBPL)*, Potsdam, Germany, 2003.

[25] W3C. Annotea Project. http://www.w3.org/2001/Annotea.

[26] Y. R. Wang and S. E. Madnick. A Polygen Model for Heterogeneous Database Systems: The Source Tagging Perspective. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 519–538, Brisbane, Queensland, Australia, 1990.