The VLDB Journal (2005) 14(4): 373–396
DOI 10.1007/s00778-005-0156-6

REGULAR PAPER

**Deepavali Bhagwat · Laura Chiticariu ·
Wang-Chiew Tan · Gaurav Vijayvargiya**

An annotation management system for relational databases

Received: 30 November 2004 / Revised version: 12 April 2005 / Published online: 25 October 2005
© Springer-Verlag 2005

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 could be used for understanding the provenance (aka lineage) of data, who has seen or edited a piece of data or the quality of data, which are useful functionalities for 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.

Keywords Data provenance · Lineage · Annotation propagation · Metadata

D. Bhagwat · L. Chiticariu (✉) · W.-C. Tan · G. Vijayvargiya
Department of Computer Science, University of California, Santa Cruz, 1156 High Street, Santa Cruz, CA 95064, USA
E-mail: laura@cs.ucsc.edu

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 [1] and SWISS-PROT [2]. Data that is collected from other sources is often cleansed and reformatted before it is compiled into a new database. Furthermore, it is common for such newly created databases to 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. Very often, there is information about data that is not kept in the database but one would like to propagate this information along as data is being moved around. Examples include information about the perceived accuracy or reliability of experimental results by domain experts, or information about who has seen or edited a piece of data. In fact, our initial motivation for the design of a system that can propagate additional information around is to propagate the provenance of data items along as data is being copied. With the proliferation of many such interdependent databases (see [3] for a catalog of biology databases), 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 describe an annotation management system for relational databases where every column of every tuple in every relation can be annotated with zero or more annotations. We use the term *annotation* to mean information about data such as provenance, comments, or other types of metadata. The annotations are automatically propagated along as data is being transformed through a query. In its default behavior, our system propagates annotations based on where data is copied from. As a consequence, if every column of every tuple in a database is annotated with

its address, the provenance of data is propagated along as data is being transformed. Hence, one immediate application is to use these annotations to systematically trace the provenance and flow of data. 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 [4–6].

We describe three propagation schemes for propagating 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 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 [7], two equivalent queries may propagate annotations differently. For instance, consider the relations $R(A,B)$ and $S(B,C)$. The following two equivalent queries compute the join of R and S on the B attribute.

Q_1:		Q_2:	
SELECT	$r.B$	SELECT	$s.B$
FROM	$R\ r, S\ s$	FROM	$R\ r, S\ s$
WHERE	$r.B = s.B$	WHERE	$r.B = s.B$

Intuitively, it is easy to see that Q_1 propagates the annotations from the B attribute of R, while Q_2 propagates annotations from the B attribute of S. 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 [8, 9]. We also describe an alternative method of propagating annotations, called the *default-all* scheme, which propagates 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 provide a third propagation scheme, called 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 for every attribute of every relation, there is an additional column that stores the annotations for that attribute. 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 (on the average, the default and default-all queries we experimented with took about 0.71%, and respectively 0.1% more time to execute on the 100 MB database when the number of annotations was doubled from 30 to 60%). 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 100 MB database took around 40% more time to execute than their corresponding SQL queries. For larger databases (500 MB and 1 GB), the pSQL queries with default scheme took only about 15%, and respectively 24% more time to execute than their corresponding SQL queries on the average. However, our empirical results indicate that the performance of pSQL queries starts to degrade on databases annotated more than 100%. This suggests that

perhaps our scheme for storing annotations is not the best suited in such scenarios. We plan to investigate the trade-offs between different other annotation storage schemes in the future.

Related Work The problem of computing data provenance is not new. Cui, Widom, and Wiener [10] 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 [10] 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 [10] is called *why-provenance* according to Buneman, Khanna, and Tan [11]. 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 [11] and it is this type of provenance that we use for determining where annotations are propagated from. In both works [10, 11], 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 [8, 9, 12]. In fact, our annotation propagation rules which propagate annotations based on where-provenance are similar to those proposed in [9]. In [9], 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 exact location within the source locations, or a comment on the data). An annotation is also an example of superimposed information (data "placed" over existing information), as described in [13].

Numerous annotation systems have been built to support and manage annotations on text and HTML documents [14–18]. Recently, annotation systems for genomic sequences [19–21] have also been built. Laliberte and Braverman [15] discussed how to use the HTTP protocol to design a scalable annotation system for HTML pages. Schickler, Mazer, and Brooks [17] 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, 18] 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 [16, 22, 23] also discussed the use of annotations with certain desirable properties on multivalent documents [23] which support documents of different media types, such as images, postscript, or HTML. DAS or Biodas [19, 20] and the Human Genome Browser [21] 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 focused 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 Sect. 2, we describe pSQL and the three different propagation schemes. In Sect. 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 Sect. 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 Sect. 5, we describe our experimental results and in Sects. 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 [25] (also known as the Select-Project-Join-Union fragment of SQL). We extend this fragment of SQL with a `PROPAGATE`

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 $\{a_{14}\}$

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_1:

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

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

Fig. 1 Three protein databases, a mapping table and the result of three pSQL queries

clause to allow users to specify how annotations should propagate.

Definition 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 |
                  r1.A1 TO B1, ..., rn.An TO Bn
```

The fromlist *of a pSQL query fragment is of the form "*$R_1\, r_1, \ldots, R_k\, r_k$*" 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_j*s.* □

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.

We note at this point that other set operators (such as intersection or set difference) and aggregate functions are not allowed to appear in a pSQL query. In Sect. 6.2, we describe how we extend pSQL queries to a larger fragment of SQL where some aggregates are also allowed.

Example 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 at the top of Fig. 1. 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 Fig. 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 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 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 realize 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 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 at the bottom of Fig. 1. The query Q_2 retrieves all tuples from the PIR table under the default-all propagation scheme. Since the following query is equivalent to Q_2,

`SELECT DISTINCT` p.ID `AS` ID, p.Name `AS` Name
`FROM` PIR p, PIR q
`WHERE` p.Name = q.Name

annotations of proteins with the same name are combined together. As a consequence, the protein with name AB has both annotations a_8 and a_{10}. Intuitively, the annotations we get in the result of a default-all pSQL query fragment Q are the combined annotations of results from all equivalent queries of Q. 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. Let $\mathcal{L}$ denote the set of all strings. Each cell c in a database is *associated with* a set of annotations $\{a_1, \ldots, a_k\}$ 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 $\mathcal{A}(r, t, i)$ to denote the set of all annotations attached to the cell (r, t, i). Similarly, $\mathcal{A}(t, i)$ denotes the set of all annotations attached to the cell (t, i) in the context where the relation r is clear.

Example 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 $\{a_8, a_{10}\}$. □

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, we have $Q(D) \subseteq Q'(D)$ and for every output location (t, i) in $Q(D)$, it is the case that $\mathcal{A}(Q(D), t, i) \subseteq \mathcal{A}(Q'(D), t, i)$.

Example 5 The queries Q' and Q'' in Example 2 are equivalent. However, they are not annotation-equivalent since different annotations are associated with the results. Consider the following query Q:

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

The query Q_3 of Example 3 is annotation-contained in Q since they are equivalent and the annotations associated

with each cell in the result of Q_3 is contained in the set of annotations associated with the corresponding cell in the result of Q. Intuitively, Q_3 is annotation-contained in Q because they are equivalent and the `ID` attribute in the *selectlist* of both queries receive the same annotations from g.ID and g.Desc. Furthermore, the Desc attribute in the *selectlist* of Q receives annotations from g.ID. □

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` $B_1, \ldots, r_n.A_n$ `TO` B_n". The queries Q_3 of Example 3 and Q of Example 5 are examples of pSQL queries with custom propagation scheme. 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 $t_1, \ldots, t_k$ are the emitted tuples and $s_1, \ldots, s_m$ are the tuples that result when duplicate output tuples have been merged. Then, for every output location (s, B), we have $\mathcal{A}(s, B) = \bigcup_{t_j=s, j\in[1,k]} \mathcal{A}(t_j, B)$.

Example 6 To illustrate the effect of removing duplicate output tuples and merging annotations of duplicate tuples, consider the query below:

```
SELECT DISTINCT   Name AS Name
FROM              PIR
PROPAGATE         DEFAULT
```

The result of executing the above query will merge the annotations a_8 and a_{10} of the Name values of the first and second tuple in PIR. Hence, the final output is a single tuple (AB) with annotations $\{a_8, a_{10}\}$.

As another example, the query Q_3 of Example 3 has a custom propagation scheme where annotations on both `ID` and `Desc` columns of each tuple propagate to the `ID` column of the output tuple. As a consequence, the `ID` column of every output tuple is the union of annotations associated with the `ID` and `Desc` columns of the corresponding tuple in Genbank. □

Observe that the result of a pSQL fragment evaluated over a database would not contain any duplicate tuples, since we assume set semantics. We refer the reader to Sect. 6.2 for a discussion on extending pSQL to handle bag semantics as well.

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 the locations 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. Suppose $t_1, \ldots, t_k$ are the emitted tuples and $s_1, \ldots, s_m$ are the tuples that result when duplicate output tuples have been merged. That is, for every output location (s, B), we have $\mathcal{A}(s, B) = \bigcup_{t_j=s, j\in[1,k]} \mathcal{A}(t_j, B)$.

Example 7 Suppose we have the following pSQL query where each fragment uses the default propagation scheme.

```
SELECT DISTINCT   Desc AS Desc
FROM              SWISS-PROT
PROPAGATE         DEFAULT
UNION
SELECT DISTINCT   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 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) 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 *Let Q denote a pSQL query with default-all propagation scheme. Let* SQL(Q) *denote the SQL query that corresponds to Q and let $\mathcal{E}$(SQL(Q)) denote the set of all pSQL queries Q' under the default propagation scheme such that* SQL(Q') *is equivalent to* SQL(Q). *A* query basis of Q, *denoted as $\mathcal{B}(Q)$, is a finite set of pSQL queries such that*

$$\bigcup_{q\in\mathcal{B}(Q)} q =_a \bigcup_{q\in\mathcal{E}(\mathrm{SQL}(Q))} q$$

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. (The size of a query basis is the sum of sizes of each pSQL query fragment in the query basis. The size of each pSQL query fragment is the sum of the number of attributes in the *selectlist*, the number of relations in the *fromlist* and the number of attributes appearing in the *wherelist*.)

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. (This is Step 1 of the algorithm Generate-Query-Basis below.) Intuitively, a *representative* query of Q is a query that is equivalent to Q and for every attribute B that is equal or transitively equal to an attribute A in the *selectlist* of Q, the annotations of B are propagated to A. More precisely, if A is among the *selectlist* and we have $A = B$ and $D = B$ in the *wherelist* of Q, then the *propagatelist* will contain the propagate clauses "A TO A", "B TO A" and "D TO A".

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. (This is Step 2 of the algorithm.) Each auxiliary query is equivalent to Q but may propagate additional annotations to the output that are not propagated by Q_0. In other words, every output value may contain additional annotations from attributes of other relations which contain identical values. 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. In other words, a query basis of Q, denoted as $\mathcal{B}(Q)$, is a finite set of pSQL query fragments such that $\bigcup_{q\in\mathcal{B}(Q)} q =_a \bigcup_{q\in\mathcal{E}(SQL(Q))} q$, where $\mathcal{E}(SQL(Q))$ denotes the set of pSQL query fragments Q' such that $SQL(Q')$ is equivalent to $SQL(Q)$.

We present next 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 (i.e., union of pSQL query fragments) in general and the details are described in the Appendix.

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 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" to 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" to 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 D" in the PROPAGATE clause of Q_0 where C=D, do the following:
   Create a query Q' that is identical to Q_0. Assume that s is a tuple variable for relation S. 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'. (The auxiliary query Q' is equivalent to Q but may carry additional annotations to the output.)
3. *Return $\mathcal{B}(Q)$.*

Example 8 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 Fig. 1. Such mapping tables commonly occur in integrating many sources with overlapping information [26]. 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

Q_2 =	Q_3 =	Q_4 =
SELECT DISTINCT t.swissprot AS ID, p.Name AS Name, s.Desc AS Desc	SELECT DISTINCT t.swissprot AS ID, p.Name AS Name, s.Desc AS Desc	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'	FROM Mapping_Table t, SWISS-PROT s, PIR p, **SWISS-PROT** s'	FROM Mapping_Table t, SWISS-PROT s, PIR p, **PIR** p'
WHERE t.swissprot = s.ID AND t.pir = p.ID AND s'**.ID =** s**.ID**	WHERE t.swissprot = s.ID AND t.pir = p.ID AND s'**.Desc =** s**.Desc**	WHERE t.swissprot = s.ID AND t.pir = p.ID AND p'**.Name =** p**.Name**
PROPAGATE t.swissprot TO ID, s.ID TO ID, p.Name TO Name, s.Desc TO Desc, s'**.ID** TO **ID**	PROPAGATE t.swissprot TO ID, s.ID TO ID, p.Name TO Name, s.Desc TO Desc, s'**.Desc** TO **Desc**	PROPAGATE t.swissprot TO ID, s.ID TO ID, p.Name TO Name, s.Desc TO Desc, p'**.Name** TO **Name**

Fig. 2 Some of the auxiliary queries generated by Step 2 of Generate-Query-Basis on Example 8

second step of the algorithm generates four auxiliary queries. The first query is shown below and the rest are shown in Fig. 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 AND
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 differs 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 Fig. 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 in conjunctive query-like notation, which we will continue to use throughout the rest of the discussion, for ease of exposition. In conjunctive query-like notation, a query Q is represented as

$$H(\bar{x}) : -S_1(\bar{y_1}), \ldots, S_n(\bar{y_n}), \textit{equalities}.$$

where $\bar{x}$, $\bar{y_i}$, $i \in [1, n]$, denote vectors of variables and every variable in $\bar{x}$ occurs in $\bar{y_i}$ for some $i \in [1, n]$ and *equalities* is a list of zero of more y=c clauses where y is a variable that occurs amongst $\bar{y_i}$s and c is a constant. The variables in $\bar{x}$ are called *distinguished variables*. Each subgoal corresponds to a relation in the *fromlist* of Q. The 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 $H(\bar{x})$ 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, $C(Q_0)$ of Example 8 can be written as

$$\mathrm{H}_0(x, y, z) \text{ :-Mapping_Table}(w, x, u, v), \text{SWISS-PROT}(x, z), \text{PIR}(u, y).$$

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 [7] by tracing the occurrence of distinguished variables in the query. For example, by tracing the occurrence of the variable x in the query H_0, we can conclude that the annotations in the first column of an output tuple t are 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 H_0. Hence, the representative query Q_0 of Example 8 is annotation-equivalent to $C(Q_0)$.

We next focus on showing that given a query Q, the algorithm Generate-Query-Basis(Q) correctly generates $\mathcal{B}(Q)$, the query basis of Q. We first formally show that the representative query Q_0 generated by the algorithm is annotation-equivalent to its conjunctive query-like representation, $C(Q_0)$ (Proposition 1). Using this result, we further show that the conjunctive query-like representation of each query generated by our Generate-Query-Basis algorithm is annotation-contained in $\bigcup_{q\in\mathcal{B}(Q)} q$, the union of all queries in $\mathcal{B}(Q)$ (Proposition 2). Moreover, Lemma 1 shows that every query that is equivalent to Q is annotation-contained in $\bigcup_{q\in\mathcal{B}(Q)} q$. Finally, we prove our main result (Theorem 1) which states that the algorithm Generate-Query-Basis correctly generates a query basis $\mathcal{B}(Q)$ for the input query Q.

Proposition 1 *The representative query Q_0 that is generated by Generate-Query-Basis(Q) is annotation-equivalent to its conjunctive query-like representation, $C(Q_0)$.*

Proof Obviously, the query Q_0 is equivalent to $C(Q_0)$ since there is a subgoal S in $C(Q_0)$ for every relation S in the

fromlist of Q_0 and vice versa, there is an equality condition e in $C(Q_0)$ for every equality condition e the *wherelist* of Q_0 and vice versa and the head of $C(Q_0)$ produces the same attributes as the *selectlist* of Q_0. We show next that Q_0 and $C(Q_0)$ are annotation-equivalent by showing that for every database D and every output location (t, i) of $Q_0(D)$, the set of annotations $\mathcal{A}(Q_0(D), t, i)$ is equal to $\mathcal{A}(C(Q_0)(D), t, i)$. We show that if a location l' in the source D *corresponds to* a location l in the output of $C(Q_0)(D)$, then the annotations at l' are part of the annotations at l according to Q_0 and D. The converse is also true.

According to the semantics of conjunctive queries with annotation propagation stated in [7] (Appendix A.1), the set of annotations associated with an output location l is the union of the sets of annotations associated to each source location l' that *corresponds* to l. A location (s, i) in D *corresponds to* (t, j) in $C(Q_0)(D)$ where $C(Q_0)$ is of the form "$H(\bar{x}) : -S_1(\bar{y_1}), \ldots, S_n(\bar{y_n}), equalities$" if the following holds:

– for some $k \in [1, n]$, $\bar{y_k}[i]$=$\bar{x}[j]$, and there exists a valuation φ from $C(Q_0)$ into D such that $H(\varphi(\bar{x})) = t$, $S_k(\varphi(\bar{y_k}))$=s and the *equalities* are satisfied.

Suppose there is such a valuation φ for $C(Q_0)$ as stated above. Then there is a valuation φ' for Q_0 that produces t. The valuation φ' is such that $\varphi'(r)$=$S(\varphi(\bar{y}))$ where r is a tuple variable in Q_0 and $S(\bar{y})$ is the corresponding subgoal in $C(Q_0)$ which represents the relation that r ranges over. So $\varphi'(u)$=s for the tuple variable u in Q_0 which ranges over the relation S_k and the output tuple is t under φ' according to Q_0. Since $\bar{y_k}[i]$=$\bar{x}[j]$ in $C(Q_0)$, it must be that the attribute at position i of S_k (call it A) is equal to the attribute at position j in the *selectlist* of Q_0 (call it B) or transitively equal to B. Hence, there must be a clause "`PROPAGATE` $u.A$ `TO` B" in the propagate clause of Q_0. Therefore, under the valuation φ', the annotations at (s, i) are part of the annotations at (t, j) according to Q_0 and D.

For the converse, suppose there is a valuation φ for Q_0 and D such that the annotations at (s, i) are part of the annotations at (t, j) according to Q_0 and D with φ. So $\varphi(u)$=s for some tuple variable u in Q_0 and the output tuple is t under φ. Clearly, there must also be a valuation φ' for $C(Q_0)$ and D that produces t. The valuation φ' is such that $S(\varphi'(\bar{y}))$=$\varphi(r)$ where r is a tuple variable in Q_0, $S(\bar{y})$ is a subgoal in $C(Q_0)$ and S is the relation that r ranges over. So there exists a subgoal $S_k(\bar{y_k})$ in $C(Q_0)$ for some $k \in [1, n]$ such that $S_k(\varphi'(\bar{y_k}))$=s=$\varphi(u)$. Let the ith attribute of s be A and the jth attribute of the output tuple t be B. Hence, there must be a "`PROPAGATE` $u.A$ `TO` B" clause in Q_0 and "$v.C$ `AS` B" is in the *selectlist* for some tuple variable v and attribute C. According to Generate-Query-Basis algorithm, this means that either $u.A$ is equal to $v.C$ or transitively equal to $v.C$. Hence, in $C(Q_0)$, it must be that $\bar{y_k}[i]$=$\bar{x}[j]$. So we have $\bar{y_k}[i]$=$\bar{x}[j]$, $H(\varphi'(\bar{x}))$=t, $S_k(\varphi'(\bar{y_k}))$=s and the equalities are satisfied under φ'. Hence, (s, i) corresponds to (t, j) according to $C(Q_0)$ and D with valuation φ'. □

In Step 2, the algorithm generates one query for every position in the body where a distinguished variable occurs in H_0. For example, the following four auxiliary queries, in conjunctive query notation, are generated based on H_0. They are annotation-equivalent to the pSQL query fragments $Q_1, \ldots, Q_4$ shown in Example 8 and Fig. 2, respectively.

$H_1(x, y, z)$:- Mapping_Table(w, x, u, v), SWISS-PROT(x, z),
PIR(u, y), **Mapping_Table**(w_1, x, w_2, w_3).
$H_2(x, y, z)$:- Mapping_Table(w, x, u, v), SWISS-PROT(x, z),
PIR(u, y), **SWISS-PROT**(x, w_1).
$H_3(x, y, z)$:- Mapping_Table(w, x, u, v), SWISS-PROT(x, z),
PIR(u, y), **SWISS-PROT**(w_1, z).
$H_4(x, y, z)$:- Mapping_Table(w, x, u, v), SWISS-PROT(x, z),
PIR(u, y), **PIR**(w_1, y).

Proposition 2 *For every query $Q' \in \mathcal{B}(Q)$ where $\mathcal{B}(Q)$ is the result of Generate-Query-Basis(Q), $C(Q')$ is annotation-contained in $\bigcup_{q \in \mathcal{B}(Q)} q$.*

Proof First, $C(Q_0)$ is annotation-contained in $\bigcup_{q \in \mathcal{B}(Q)} q$ since $Q_0 \in \mathcal{B}(Q)$ and $C(Q_0)$ is annotation-equivalent to Q_0 according to Proposition 1.

Let Q' denote a query in $\mathcal{B}(Q)$ and Q' is not Q_0. That is, Q' is one of the auxiliary queries. Let $C(Q')$ be of the form "$H(\bar{x}) : -S_1(\bar{y_1}), \ldots, S_n(\bar{y_n})$, equalities". Given any database D, let (s, i) be a location in D which corresponds to a location (t, j) in $C(Q')(D)$ on a valuation φ. So $S_k(\varphi(\bar{y_k}))$ $= s$ for some $k \in [1, n]$ and $H(\varphi(\bar{x})) = t$ and $\bar{y_k}[i] = \bar{x}[j]$. There is also a valuation φ' for Q' and D which produces t. The valuation φ' is such that $\varphi'(r) = S(\varphi(\bar{y}))$ where r is a tuple variable in Q' and $S(\bar{y})$ is the corresponding subgoal in $C(Q')$ which represents the relation that r ranges over in Q'. So $\varphi'(r_1)$=s for the tuple variable r_1 in Q' which ranges over the relation S_k and the output tuple is t under φ' according to Q'. We show next that for every annotation propagated by Q', there is a query in $\mathcal{B}(Q)$ that would propagate the annotation in the same way.

Suppose $S_k(\bar{y_k})$ is a subgoal among the subgoals of $C(Q_0)$ where Q_0 is the representative query generated by Step 1 of the algorithm Generate-Query-Basis. (Recall that $C(Q')$ differs from $C(Q_0)$ in that it has an additional subgoal added by Step 2 of the algorithm.) Since $\bar{y_k}[i]$=$\bar{x}[j]$ and $S_k(\bar{y_k})$ is a subgoal among the subgoals of $C(Q_0)$, it must be that the attribute at position i of S_k (call it B) is equal to the attribute at position j in the *selectlist* of Q' (call it A) or transitively equal to A. Hence, there must be a clause "`PROPAGATE` $r_1.B$ `TO` A" in the propagate clause of Q_0 (and hence Q'). Therefore, under the valuation φ', the annotations at (s, i) are part of the annotations at (t, j) according to Q' and D.

Suppose $S_k(\bar{y_k})$ is not a subgoal among the subgoals of $C(Q_0)$. That is, $S_k(\bar{y_k})$ is the subgoal that corresponds to the extra relation in the *fromlist*, added by Step 2 of algorithm Generate-Query-Basis. Let the attribute at the ith position of S_k be B. By Step 2 of the algorithm, it must be that the condition "$r_1.B$=$r_2.B$" is the added condition in the *wherelist* for some tuple variable r_2 that ranges over a second S_k relation in Q' and "$r_1.B$ `TO` C" is the added propagate clause of Q' for some output attribute C in the *selectlist*. Let the attribute

at the jth position of the output be A. If C is the same as A, then the annotations at (s, i) are part of the annotations at (t, j) according to Q' and D under the valuation φ'. Suppose C is not equal to A. Since "$r_1.B$=$r_2.B$" and $\bar{y_k}[i]$=$\bar{x}[j]$ in $C(Q')$, it must be that $r_2.B$ is equal or transitively equal to A. (Therefore, Q_0 must contain "$r_2.B$ TO A" in the propagate clause.) Hence, there must be a query q in $\mathcal{B}(Q)$ which is identical to Q' except that "$r_1.B$ TO A" is in the propagate clause instead of "$r_1.B$ TO C". Therefore, under the valuation φ', the annotations at (s, i) are part of the annotations at (t, j) according to q and D. □

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 1 *Let $\mathcal{B}(Q)$ denote the result produced by the algorithm 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 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, $C(Q)$ (Proposition 1). 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. (A minimal query is a query in which no subquery, one that has less subgoals or joins, is equivalent to it.)

$C(Q)$: $H(\bar{x})$:- *minpart*, *rest1*.
$C(Q')$: $H(\bar{x})$:- *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 [7] 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 ([27], Appendix A.3) 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 where $r \in [1, m]$, every variable x, every i, and every p such that x that occurs at both the ith position of var($Q_r[0]$) and the jth position of var($Q_r[p]$), 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 next 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 $S(\bar{u})$ (i.e., the jth variable of $\bar{u}$ is x) in the body of $C(Q')$, there is a generated query Q_g in $\mathcal{B}(Q)$ and a homomorphism $h : C(Q_g) \to C(Q')$ that satisfies the conditions (1) and (2) stated in the fact. Then by the above fact, we have $C(Q') \subseteq_a C(Q_g)$. By Proposition 2, we know that $C(Q_g) \subseteq_a \bigcup_{q\in\mathcal{B}(Q)} q$. 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 $Q' \subseteq_a \bigcup_{q\in\mathcal{B}(Q)} q$, which was to be shown.

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 $S(\bar{u})$ of $C(Q')$.

Case 1 If $S(\bar{u})$ 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,

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

The variable x occurs at the jth position in the subgoal $S(\bar{w_1}, x, \bar{w_2})$ and $\bar{w_1}$ and $\bar{w_2}$ 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. (Note that a clause "B TO A" is also 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 possibly 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 $h' : C(Q) \to C(Q')$ which we know exists since $C(Q)$ and $C(Q')$ are equivalent. The homomorphism h' is extended to h'' by mapping the ith variable in $\bar{w_1}$ to the corresponding ith variable in $\bar{u}$ and the ith variable in $\bar{w_2}$ to the $(j + i)$th variable in $\bar{u}$ (this is possible since $\bar{w_1}$ and $\bar{w_2}$ are distinct variables). Clearly, h'' satisfies the conditions required by the above fact.

Case 2 If $S(\bar{u})$ are among the subgoals in *rest2* of $C(Q')$, we first claim that a subgoal $S(\bar{u'})$, 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, which was to be shown.

We show next that if $S(\bar{u})$ are among the subgoals in *rest2* of $C(Q')$, there must exist such a subgoal $S(\bar{u'})$ among the *minpart* of $C(Q')$ where the jth variable of $\bar{u'}$ is x. 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 $S(\ldots x \ldots)$ among the subgoals in the *minpart* of $C(Q')$ such that x occurs at the jth position of this subgoal. We therefore conclude that $S(\bar{u'})$ exists. □

Theorem 1 *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., $SQL(q)$=$SQL(Q)$). Let $\mathcal{B}(Q)$ denote the result of running the algorithm Generate-Query-Basis on Q. By Lemma 1, we have that $\bigcup_{q\in\mathcal{E}(Q)} q \subseteq_a \bigcup_{q\in\mathcal{B}(Q)} q$. Since $\mathcal{B}(Q) \subseteq \mathcal{E}(Q)$ (the representative query and auxiliary queries are each equivalent to Q), we immediately have $\bigcup_{q\in\mathcal{B}(Q)} q \subseteq_a \bigcup_{q\in\mathcal{E}(Q)} q$ and hence the result. □

The next proposition shows that the size of a query basis is polynomial in the size of Q. The size of a query basis is the sum of sizes of each pSQL query fragment in the query basis. The size of each pSQL query fragment is the sum of the number of attributes in the *selectlist*, the number of relations in the *fromlist* and the number of attributes appearing in the *wherelist*. This result shows that the result of executing a query basis is polynomial in the size of the database (data complexity).

Proposition 3 *Given a pSQL query fragment Q with 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.*

Proof Let s, f, and w denote the number of clauses in the *selectlist*, number of relations in the *fromlist*, and number of equalities in the *wherelist* of Q, respectively. The size of Q consists in the sum of the number of attributes in the *selectlist*, the number of relations in the *fromlist* and the number of attributes appearing in the *wherelist*, that is $|Q|$ is at most $s+f+2*w$. One representative query Q_0 is generated by the algorithm. The size of the propagate list of Q_0 is at most $s+s*2*w$. (In the worst case, every attribute in the *wherelist* propagates to every attribute in the *selectlist*.) The number of auxiliary queries generated is therefore at most $s*(s+s*2*w)$ which is $|selectlist|*|propagatelist|$. Hence, the total number of queries in Generate-Query-Basis(Q) is at most $1+s*(s+s*2*w)$.

The size of the *selectlist*, *fromlist*, *wherelist*, and *propagatelist* of Q_0 is s, f, w, and at most $s+s*2*w$, respectively. The size of each auxiliary query is thus at most $s+(f+1)+(w+2)+(s+s*2*w+1)$ since one additional relation, one condition, and one propagate clause is added to Q_0. □

An optimization Observe that the auxiliary pSQL queries overlap significantly in the PROPAGATE clauses (e.g., see Fig. 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 Fig. 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 relational database management system (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 sorted 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 $\{a_1, \ldots, a_k\}$ are the annotations associated with the location (t, A), then there will be k tuples $t_1, \ldots, t_k$ in R' such that $t_i.A_a$=a_i for $i \in [1, k]$ and the projection of t_i on the attributes of R equals t, for $i \in [1, k]$. 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

Fig. 3 Architecture of our system

(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.

Propagating Provenance To use our system to automatically propagate provenance along, we first associate each cell with a distinct annotation to denote its address. In what is shown below, R' is defined as a view of an original relation R using internal row identifiers:

```
CREATE VIEW   R' AS
SELECT        A AS A, rowid||'#A' AS A_a,
              B AS B, rowid||'#B' AS B_a
FROM          R
```

For the above view definition, rowid is an internal row identifier used in many database systems such as Oracle and Postgres. We refer the interested reader to [28] for a detailed explanation of how one can automatically trace the provenance and flow of data using this naive storage scheme.

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 Sect. 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 Sect. 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 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 ··· 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 9 Consider the SWISS-PROT relation of Fig. 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$_a$ }.

```
Q_1 =  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
Q_2 =  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:

```
Q_s =  SELECT DISTINCT   *
       FROM              (Q_1 UNION Q_2)
       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 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 Sect. 4.1 as input, it returns { (a $\{a_1, a_2\}$, b $\{b_1\}$) }.

Example 10 Suppose the following tuples are returned by the database system, sorted according to the attributes A and B.

A	A_a	B	B_a
a	a_1	b	a_2
a	a_3	b	–
a	–	c	a_2

The result returned by the postprocessor is {(a $\{a_1, a_3\}$, b $\{a_2\}$), (a {}, c $\{a_2\}$)}.

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.

5.1 Methodology

Our system is implemented with Java v1.4.2 on top of Oracle 9*i* Enterprise Edition Release 9.2.0.1.0. We conducted the experiments on a Pentium 4, 2.8 GHz machine with 1 GB RAM.

Datasets The databases used to perform the experiments are from the TPC Benchmark H (TPCH) Standard Specification Revision 2.1.0 [29]. For our experiments we used TPCH data of various sizes and we call these databases the *unannotated databases*. In order to create annotated datasets, we modified the TPCH schema to conform to our naive storage scheme by adding an additional attribute for every attribute of every relation in the TPCH schema. For each unannotated database, 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 of the database will contain an annotation. We experimented with three datasets of sizes 100 MB, 500 MB, and 1 GB. In each dataset we have the unannotated database and the three annotated databases (30, 60, and 100%).

Workload We ran queries of increasing join sizes and with varying number of output attributes to determine how well our system scales for these types of queries. As mentioned in Sect. 3, the number of joins and output attributes of a query are in fact particularly important in our Generate-Query-Basis algorithm. We did not use TPCH queries in our experiments because they include aggregates and nested queries.

The queries $Q_0, \ldots, Q_4$ which denote queries with zero to four joins, respectively, are shown in Fig. 4(a). For example, Q_2 denotes the query Supplier $\bowtie$ Nation $\bowtie$ Region with two joins, on the attributes `Nationkey` and `Regionkey`, respectively. The cardinality of each relation in the 100 MB dataset is shown in brackets. (For the 500 MB and 1 GB datasets, the cardinalities of relations Nation and Region are the same, while the cardinalities of relations Customer, Supplier and Partsupp are 5, and respectively, 10 times larger.) Our workload consists of queries $Q_i(1), Q_i(3), Q_i(5), i \in [0, 4]$, which denote the queries with i joins and one, three, and five output attributes, respectively.

Techniques We executed the workload queries under both the default and the default-all schemes on the annotated databases. We also executed the SQL query that corresponds to each of these queries on the unannotated databases, in order to be able to measure the overhead that the propagation of annotations introduces in the overall running time of the queries. All the experiments were performed on warm buffer and the buffer size was set to 256 MB.

We have implemented and tested both optimized as well as unoptimized versions of our Generate-Query-Basis algorithm. In what follows we present only our results obtained with the optimized version, as we observed that it consistently and significantly outperforms the unoptimized version.

5.2 Experimental results

Experiment 1 The goal of this experiment is to compare the performance of pSQL queries under different propagation schemes (default, default-all, or no propagation scheme). We measured the performance of our system for queries under the default and default-all propagation scheme on the 100% annotated database in each of our three datasets. We executed the workload queries $Q_i(1), Q_i(3), Q_i(5), i \in [0, 4]$ on the 100% annotated databases. We also executed the SQL query that corresponds to each of these queries on the unannotated databases. The results we obtained with the 100 MB, 500 MB, and 1 GB datasets are shown in Fig. 4.

Figure 4(b) 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 scheme on the 100% annotated

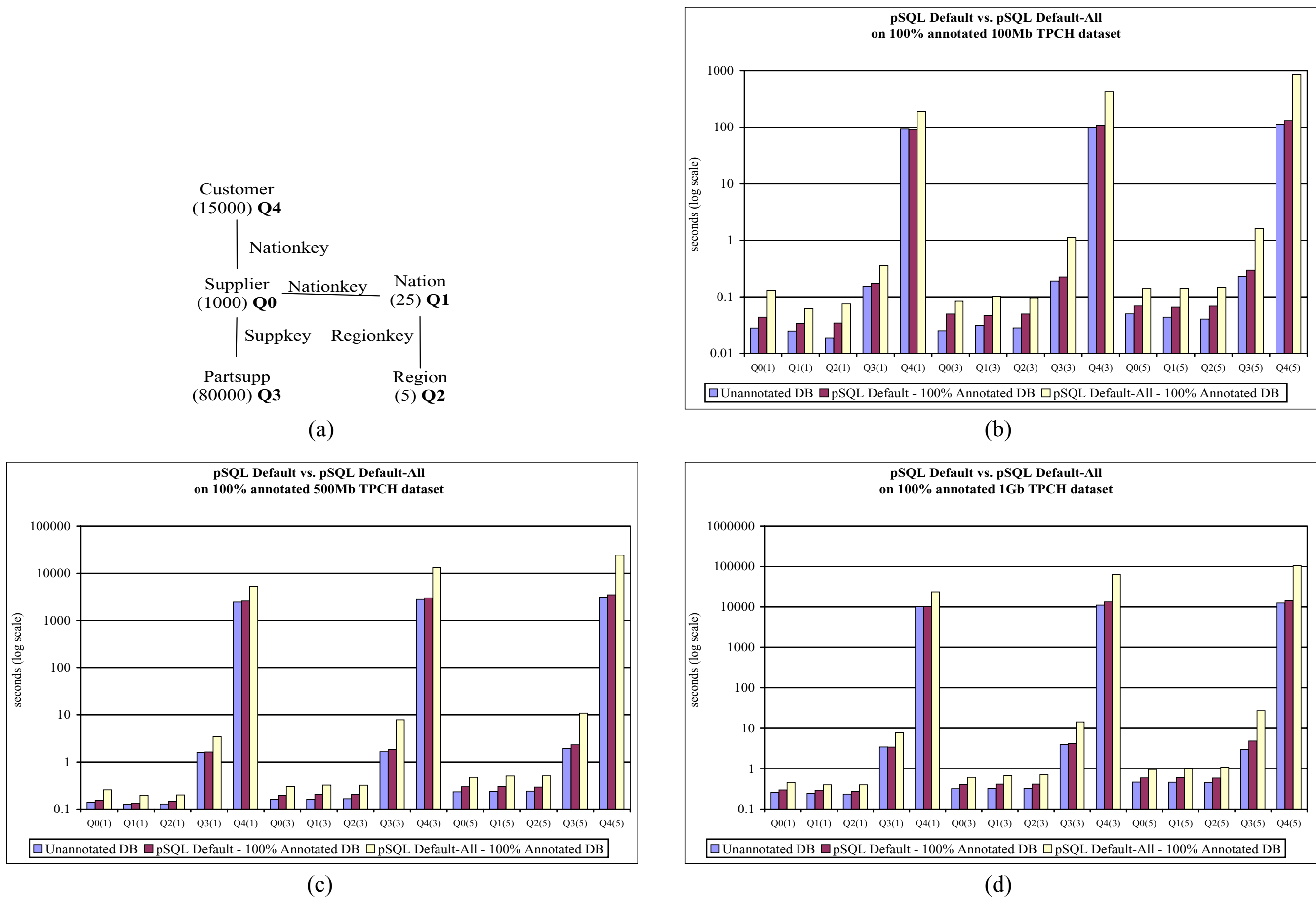

Fig. 4 (a) Queries used in our experiments and (b)–(d) comparison in performance for default and default-all schemes on the 100 MB, 500 MB, and respectively 1 GB dataset

database in the 100 MB dataset. As expected, the execution time of each query under the default scheme (respectively, the 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 eight 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 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 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, since the attributes that are selected do not participate in the joins. (The performance of default-all pSQL queries where attributes that participate in the joins are selected as well is evaluated in Experiment 3.) 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 observed the same trends for larger datasets as well. Figure 4(c) and 4(d) illustrate the execution time of each query for the default and default-all propagation scheme on the 100% annotated databases in the 500 MB, and respectively, 1 GB datasets. However, we observed that the overhead of propagating annotations under the default scheme is smaller on larger datasets. On the average, the pSQL queries with default scheme took only about 15 and 24% more time to execute than their corresponding SQL queries

Table 1 The execution time of each query for each database in the 100 MB dataset and each 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.

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

on the 500 MB, and respectively, 1 GB dataset. This is explained by the fact that on larger datasets, the postprocessing time tends to become less significant when compared to the actual time taken by the database engine to execute the queries.

Experiment 2 In this experiment we evaluate the influence of the number of annotations in a database on the execution time of pSQL queries under default or default-all schemes. We executed the workload queries $Q_i(1)$, $Q_i(3)$, $Q_i(5)$, $i \in [0, 4]$ under both default and default-all schemes on the 30, 60 and 100% annotated databases. The results we obtained with the 100 MB, 500 MB, and 1 GB datasets are illustrated in Fig. 5 (the results obtained with the 100 MB dataset are also tabulated in Table 1).

We observed that the execution time of each query increases only slightly across databases annotated in various degrees and this fact is not unexpected. As the number of annotations in the database increases, we expect an increase in the postprocessing time, as more annotations need to be compared and unioned together. Table 2 shows the average percentage increases incurred in the total execution times of the default and default-all queries we experimented with. On the 100 MB dataset, for example, the total execution time for default queries increases on the average 0.71% when the number of annotations in the database is doubled from 30% annotations to 60% annotations and 1.85% when the number of annotations is varied from 60% annotations to 100% annotations. We also remark that an increase in the number of annotations in the database induces smaller increases in the total execution times of default-all queries when compared to default queries. This is intuitive because for default-all queries, the postprocessing time is less significant when compared to the actual time taken by the engine to execute the queries.

Experiment 3 In this experiment we evaluate the effect of selecting attributes that participate in join conditions on the performance of default-all pSQL queries. For this purpose, we measured the execution time of queries $Q_i(1+j)$, $Q_i(3+j)$, $Q_i(5+j)$, $i \in [1, 3]$, $j \in [1, i]$ under the default-all propagation scheme on the annotated databases in the 100 MB dataset. These queries are identical to our original workload queries $Q_i(1)$, $Q_i(3)$, $Q_i(5)$, $i \in [1, 3]$, except that their *selectlist* additionally contains j attributes selected among the attributes that appear in some join condition in the *wherelist*. For example, consider the query $Q_1(1)$ which computes the join of tables Supplier and Nation on the `Nationkey` attribute. The query $Q_1(1+1)$ is identical to $Q_1(1)$, except that the attribute `Nationkey` (which does not appear in the *selectlist* of $Q_1(1)$) appears in the *selectlist* of $Q_1(1+1)$. The execution times of these queries are shown in Fig. 6 (they are also tabulated in Table 3). The execution times of queries $Q_i(1)$, $Q_i(3)$, $Q_i(5)$, $i \in [1, 3]$ are shown as well, for comparison purposes. Table 3 also illustrates the number of tuples retrieved by each query before the postprocessing step. The number of output tuples retrieved by each query after postprocessing is 1,000.

As expected, the execution time of the queries under the default-all propagation scheme increases as more attributes that participate in the joins are selected. On the 30% annotated database for example, the query $Q_2(1+1)$ runs three times slower when compared to $Q_2(1)$ and the query $Q_2(1+2)$ takes 15% more time to run compared to $Q_2(1+1)$. This is expected, since more pSQL queries are generated by our Generate-Query-Basis algorithm (hence more SPJ queries are executed) as the number of selected attributes involved in join conditions increases. As shown in Table 3, there are 5, and respectively, 7 SPJ queries that are executed in order to retrieve the correct annotations under the default-all scheme for queries $Q_2(1+1)$ and $Q_2(1+2)$, while only 2 SPJ queries are executed in case of $Q_2(1)$. On the average, we observed that the queries we experimented with took about 5.9, 6.2, and 34.5 times more time

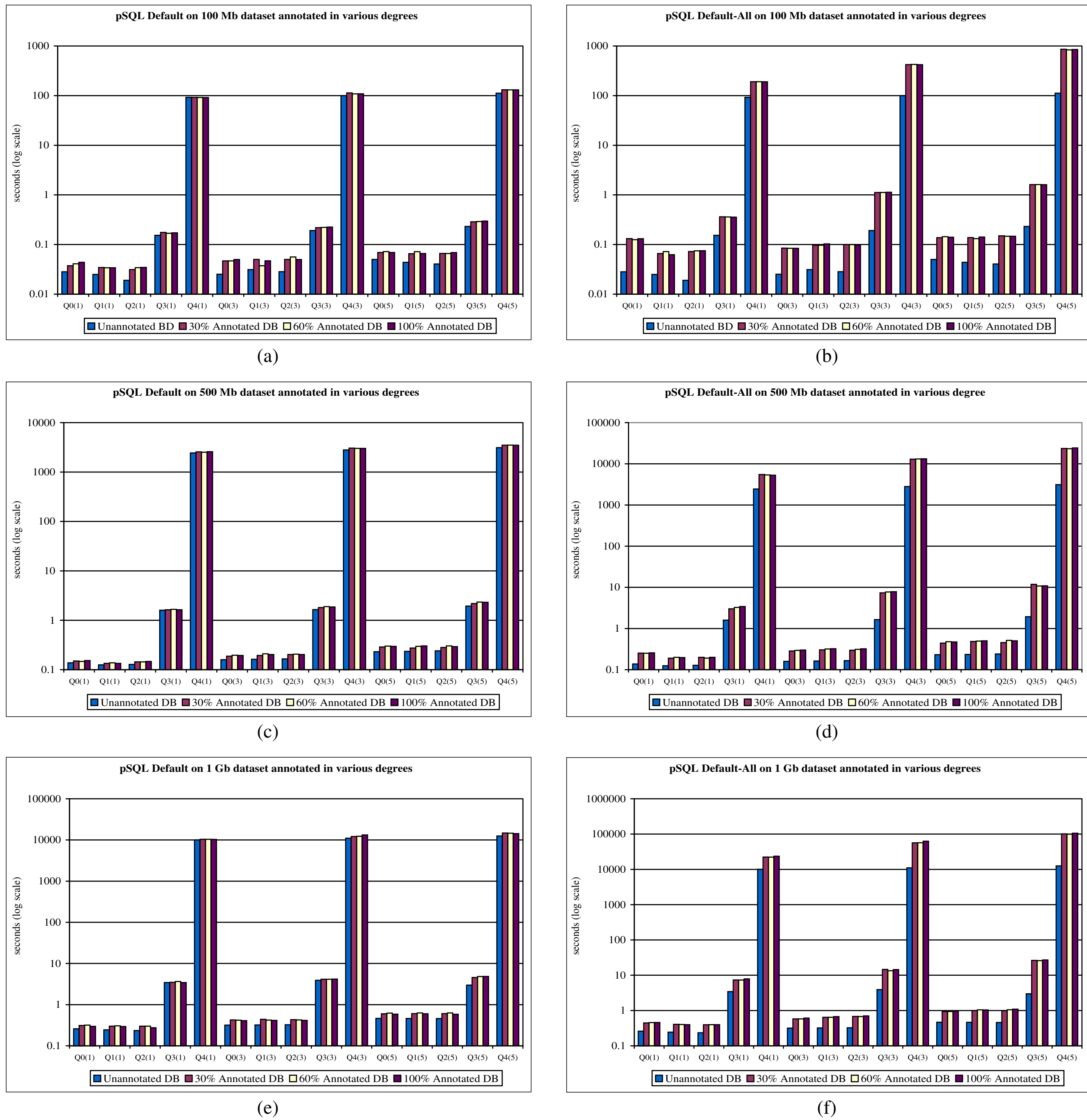

Fig. 5 Performance comparison for default and default-all pSQL queries on databases annotated in various degrees. (a), (b) Default and respectively, default-all queries on the 100 MB dataset, (c), (d) default, and respectively, default-all queries on the 500 MB dataset and (e), (f) default, and respectively, default-all queries on the 1 GB dataset

to execute on the 30, 60%, and respectively, 100% annotated databases when one join attribute was selected compared to the same queries with no join attributes appearing in their *selectlist*. When two join attributes where selected, the queries ($Q_i(1+2)$, $Q_i(3+2)$, $Q_i(5+2)$, $i \in [2,3]$) run on the average about 1.03, 1.84, and 1.24 times slower on the 30, 60%, and respectively, 100% annotated databases compared to the same queries where only one join attribute was selected (i.e., $Q_i(1+1)$, $Q_i(3+1)$, $Q_i(5+1)$, $i \in [2,3]$). Finally, the queries which select three join attributes ($Q_3(1+3)$, $Q_3(3+3)$, $Q_3(5+3)$) run about 4, 11, and respectively, 10 times slower on the 30, 60%, and respectively, 100% annotated databases compared to the same queries where only two join attributes are selected (i.e., $Q_3(1+2)$, $Q_3(3+2)$, $Q_3(5+2)$).

Observe that as the number of selected join attributes increases, not only that there are more SPJ queries that are executed, but the query engine and the postprocessor

Table 2 The average percentage increases incurred in the total execution times of the queries under both default and default-all schemes when the number of annotations in the database is varied from 30 to 60% annotations and from 60 to 100% annotations

	Default		Default-all	
Dataset	30–60% (%) increase	60–100% (%) increase	30–60% (%) increase	60–100% (%) increase
100 MB	0.71	1.85	0.10	0.02
500 MB	1.26	1.83	0.31	0.65
1 GB	1.22	1.36	0.44	0.99

Table 3 The execution time of each default-all query from Experiment 3 on the 30, 60, and 100% annotated databases in the 100 MB dataset. The columns "#pSQL" and "#SPJ" denote the size of the query basis, and respectively, the number of SPJ queries that are generated. The columns "#tuples" show the number of tuples retrieved by the queries before the postprocessing phase.

Query	Unannotated		30% Def-All		60% Def-All		100% Def-All		#pSQL	#SPJ
	#Tuples	Exec. time	#Tuples	Exec. time	#Tuples	Exec. time	#Tuples	Exec. time		
$Q_1(1)$	1,000	0.025	1,000	0.0658	1,000	0.072	1,000	0.0624	2	2
$Q_1(1+1)$	1,000	0.125	1,000	0.266	2,000	0.345	41,826	2.375	5	5
$Q_2(1)$	1,000	0.019	1,000	0.0722	1,000	0.0748	1,000	0.075	2	2
$Q_2(1+1)$	1,000	0.062	1,000	0.219	2,000	0.249	41,826	2.204	5	5
$Q_2(1+2)$	1,000	0.032	1,000	0.25	8,000	0.688	47,826	2.937	7	7
$Q_3(1)$	1,000	0.1532	1,000	0.3622	1,000	0.3594	1,000	0.356	2	2
$Q_3(1+1)$	1,000	0.188	1,000	8.515	2,000	8.546	41,826	10	5	5
$Q_3(1+2)$	1,000	0.187	1,000	6.547	8,000	7.359	47,826	13.718	7	7
$Q_3(1+3)$	1,000	0.204	22,679	23.516	109,692	37.782	206,826	118.078	9	9
$Q_1(3)$	1,000	0.0312	1,000	0.0968	1,000	0.0968	1,000	0.103	4	4
$Q_1(3+1)$	1,000	0.063	1,000	0.234	2,000	0.313	41,826	3.093	7	7
$Q_2(3)$	1,000	0.0284	1,000	0.1002	1,000	0.0998	1,000	0.0968	4	4
$Q_2(3+1)$	1,000	0.047	1,000	0.266	2,000	0.328	41,826	3.077	7	7
$Q_2(3+2)$	1,000	0.062	1,000	0.312	8,000	0.796	47,826	4	9	9
$Q_3(3)$	1,000	0.191	1,000	1.1186	1,000	1.1188	1,000	1.1314	4	4
$Q_3(3+1)$	1,000	0.187	1,000	6.937	2,000	6.859	41,826	15.485	7	7
$Q_3(3+2)$	1,000	0.219	1,000	8.375	8,000	9.438	47,826	19.249	9	9
$Q_3(3+3)$	1,000	0.219	22,679	30.015	109,692	115.202	206,826	334.532	11	11
$Q_1(5)$	1,000	0.0438	1,000	0.138	1,000	0.1312	1,002	0.1412	6	6
$Q_1(5+1)$	1,000	0.063	1,000	0.313	2,000	0.375	41,828	4.859	9	9
$Q_2(5)$	1,000	0.0406	1,000	0.1498	1,000	0.1468	1,002	0.1466	6	6
$Q_2(5+1)$	1,000	0.063	1,000	0.547	2,000	0.453	41,828	5.375	9	9
$Q_2(5+2)$	1,000	0.063	1,000	0.375	8,000	1.047	47,828	6.11	11	11
$Q_3(5)$	1,000	0.231	1,000	1.5128	1,000	1.6096	1,002	1.6064	6	6
$Q_3(5+1)$	1,000	0.249	1,000	9.047	2,000	9.173	41,828	110.422	9	9
$Q_3(5+2)$	1,000	0.265	1,000	10.953	8,000	12.297	47,828	117.984	11	11
$Q_3(5+3)$	1,000	0.266	22,679	50.563	109,692	197.828	206,828	547.313	13	13

module are given significantly more tuples to sort and respectively, merge. In the case of query $Q_1(1)$ for example, there are 1,000 tuples that have to be sorted and further postprocessed. However, in the case of query $Q_1(1+1)$ (which additionally selects one join attribute), there are 2,000, and respectively, 41,000 tuples that have to be sorted and postprocessed when this query is run on the 60%, and respectively, 100% annotated databases. This explains why $Q_1(1+1)$ runs about 5, and respectively, 38 times slower on the 60%, and respectively, 100% databases when compared to $Q_1(1)$. There is a simple explanation for the fact that as many as 41,000 tuples are retrieved (before postprocessing) when query $Q_1(1+1)$ is run on the 100% database. Recall that this query performs a join between the tables Supplier and Nation on the `Nationkey` attribute which is also selected in the output. There are 1,000 tuples in Supplier and 25 distinct values for the attribute `Nationkey`. Since in the 100% database each value has one distinct annotation, it follows that each distinct `Nationkey` value in the table Supplier has about 40 distinct annotations. According to our Generate-query-basis algorithm, a query that performs a self-join of Supplier on the `Nationkey` attribute will be executed in order to extract the 40 distinct annotations for each Supplier tuple (these annotations are

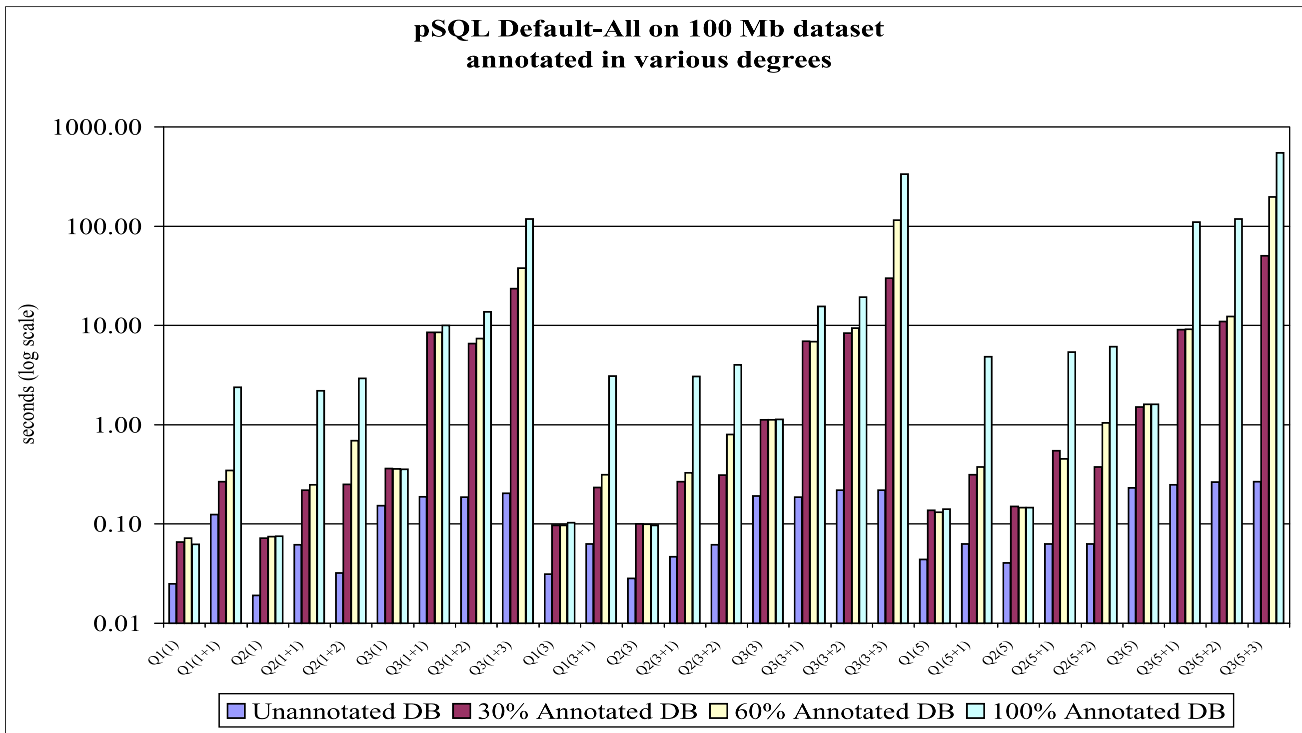

Fig. 6 Performance comparison for default-all queries on the 100 MB dataset when the number of join attributes selected in the output is varied

all needed, according to the semantics of pSQL queries with default-all propagation scheme). This query will clearly generate around 40,000 tuples. Although it seems very excessive to pull out all these 40 annotations for each tuple in Supplier, we note however that this situation arose precisely because each `Nationkey` value had a distinct annotation in the Supplier table. A scenario where we may have one annotation for each `Nationkey` value is when we are interested in tracing the provenance of data and each annotation represents an address. In this case, however, the default scheme for propagating annotations is more suitable. Our experimental results show that there may be significant overhead to the default-all scheme when annotations can be excessive.

Experiment 4 In this experiment we evaluate the performance of pSQL queries with default and default-all propagation schemes on databases annotated more than 100%. For this purpose, we have created three additional databases of size 100 MB with 130, 160%, and respectively, 200% annotations. In the 130% (respectively, 160%) annotated database, 30% (respectively, 60%) of the values have two annotations, while the rest of the values have only one annotation. In the 200% database, each value has two annotations.

We measured the performance of our system for the workload queries $Q_i(1), Q_i(3), Q_i(5)$, $i \in [0, 4]$ under the default and default-all propagation scheme on the 130, 160, and 200% annotated 100 MB databases. The results we obtained are tabulated in Table 4 (For comparison purposes, Table 4 also shows the execution time of the corresponding SQL queries on the unannotated 100 MB database.)

Figure 7 illustrates the execution time of each query for the default and default-all propagation scheme on the 100 MB database with 200% annotations. As expected, the execution time of each query under the default scheme (and respectively, default-all scheme) increases as more output attributes are emitted. As we previously explained (Experiment 1), this increase is due to longer execution time taken by Oracle, as well as additional overhead in postprocessing and an increase in the number of SPJ queries that are generated and executed (for the default-all scheme). On average, a pSQL default query $Q_i(1), Q_i(3), Q_i(5)$, $i \in [0, 1, 2, 4]$ took between three times (e.g., queries with 0 or 1 joins) and 25 times (e.g., queries with 4 joins) more time to execute compared to their corresponding SQL queries. This is an obvious consequence of our naive scheme, since each 200% annotated relation has a double number of tuples compared to the same relation with no annotations. This leads to longer postprocessing time as well as longer execution time taken by the query engine, as a double number of tuples have to be processed from each relation. (Also note that the more joins in the query, the longer the execution time taken by the query engine.) Under the default-all scheme, a query such as $Q_4(5)$ may run around 13 times slower when compared to the same query with default propagation scheme, in the worst case. This is expected, since there are 6 SPJ queries that are sent to the query engine, instead of one. In the best case (see $Q_1(1)$) a query with default-all scheme runs about twice as slow then the same query

Table 4 The execution time of each query for each propagation scheme and each database annotated more than 100% in the 100 MB dataset. 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.

Query	Unannotated	130% Def	130% Def-all	160% Def	160% Def-all	200% Def	200% Def-all	#pSQL	#SPJ
$Q_0(1)$	0.0282	0.0814	0.1846	0.084	0.1906	0.087	0.1844	2	2
$Q_1(1)$	0.025	0.0746	0.122	0.0656	0.1314	0.072	0.1472	2	2
$Q_2(1)$	0.019	0.069	0.1376	0.0782	0.1628	0.0908	0.2092	2	2
$Q_3(1)$	0.1532	1.933	5.9284	2.875	9.2096	6.8282	37.4134	2	2
$Q_4(1)$	92.4604	1644.0896	5029.2506	2648.0438	8392.7502	1682.842	5594.8612	2	2
$Q_0(3)$	0.0252	0.1034	0.4622	0.1062	0.481	0.0966	0.5084	4	4
$Q_1(3)$	0.0312	0.103	0.4968	0.1094	0.55	0.1062	0.5534	4	4
$Q_2(3)$	0.0284	0.1128	0.5344	0.1126	0.6254	0.1312	0.725	4	4
$Q_3(3)$	0.191	2.356	17.9904	3.4872	27.1808	7.569	135.0818	4	4
$Q_4(3)$	100.0106	1973.217	16386.8308	3152.887	27529.0176	2324.8586	20096.4258	4	4
$Q_0(5)$	0.0502	0.1438	0.953	0.1534	1.0408	0.1594	1.1088	6	6
$Q_1(5)$	0.0438	0.1438	1.019	0.147	1.147	0.1498	1.216	6	6
$Q_2(5)$	0.0406	0.147	1.1	0.156	1.443	0.175	1.5158	6	6
$Q_3(5)$	0.231	2.7968	34.1068	4.153	51.9968	8.2562	346.1802	6	6
$Q_4(5)$	111.8918	2347.7426	32228.192	3857.2314	53906.3124	3082.9914	42571.6014	6	6

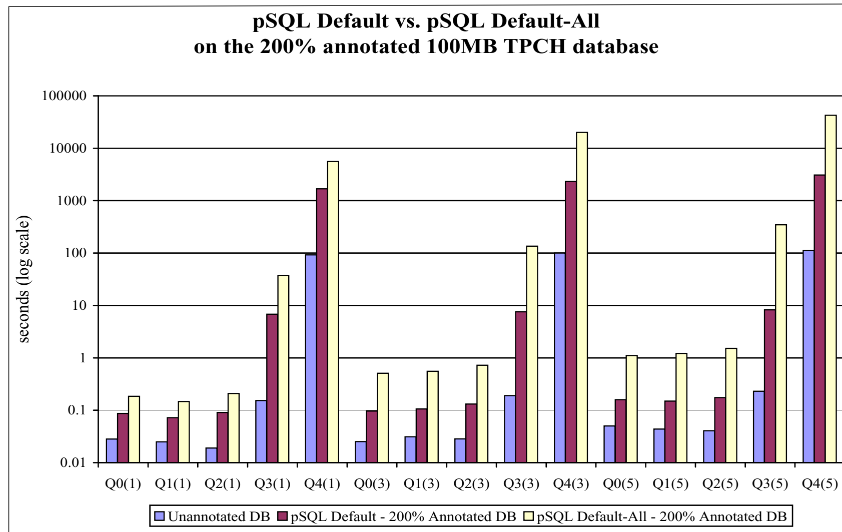

Fig. 7 Comparison in performance for default and default-all schemes on the 100 MB database annotated 200%

under the default scheme. Under both default and default-all schemes, the queries with 3 joins (i.e., $Q_3(1)$, $Q_3(3)$, and $Q_3(5)$) behaved unexpectedly. Under the default scheme, these queries ran about 39 times slower (on the average) compared to their corresponding SQL queries. The queries $Q_3(3)$ and $Q_3(5)$ took about 17, and respectively, 42 times longer to execute under the default-all scheme when compared to the default scheme. While investigating this issue we discovered that the anomaly arises because Oracle chose really poor execution plans for these particular queries.

Figure 8 shows the execution times of the queries with default and default-all schemes on the 100 MB databases with 100, 130, 160, and 200% annotations. On average, we observed that the queries with default scheme run six times slower when the number of annotations was increased from 100 to 130%. This is due to two factors. First, there are 30% more tuples in the 130% annotated database compared to the 100% annotated database. Second, Oracle chose a poorer plan for executing the queries on the 130% database, with a different join ordering as well as different join algorithms. The plan built for the 130% annotated database involved the nested loops algorithm, while hash joins were used in the plan constructed for the 100% annotated database. By tweaking the Oracle optimizer, we were able to detect that the plan built for the 100% annotated database (using hash joins only) performed much better on the 130% annotated database compared to the plan chosen by the optimizer

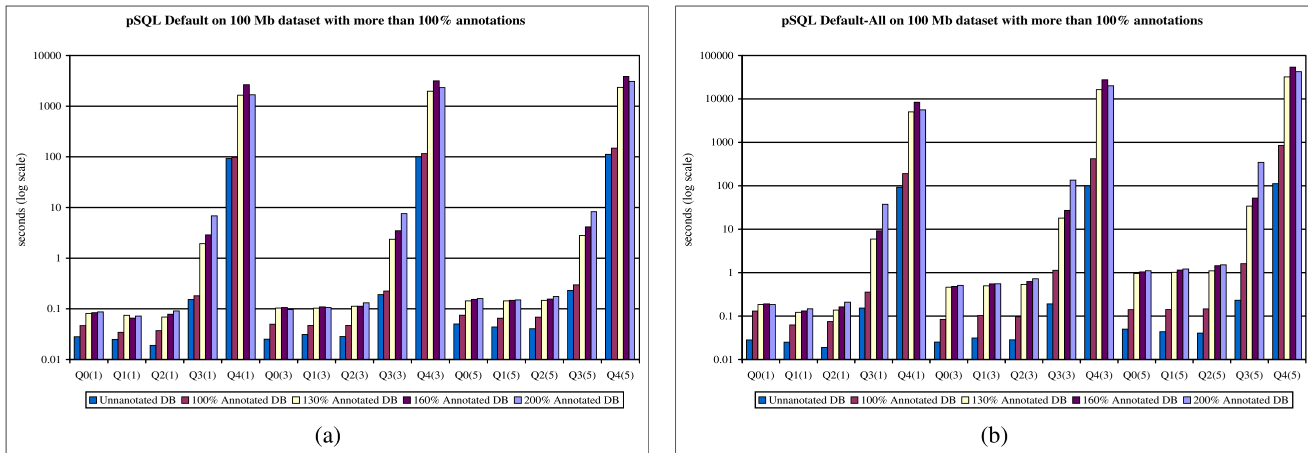

Fig. 8 Performance comparison for default (a) and default-all (b) pSQL queries on 100 MB databases with more than 100% annotations

(which involved nested loops). The queries with default scheme run on the average about 1.2 times slower when the number of annotations was increased from 130 to 160% annotations, as well as from 160 to 200% annotations. This increase in the execution time is mostly due to the fact that the number of tuples in the database increases with the number of annotations. In general, we observed that for each query, the plan the optimizer chose for the 160 to 200% annotated databases was the same as the plan chosen for the 130% annotated database. For the queries with 3 joins (i.e., $Q_3(1)$, $Q_3(3)$, and $Q_3(5)$), the optimizer chose a poorer plan on the 160% annotated database when compared to the plans generated for both the 130 and 200% annotated databases. This explains why these queries (under both default and default-all propagation schemes) run slower on the 160% annotated database than on both the 130 and 200% annotated databases. On the average, the queries with default-all scheme took about 13, 1.3, and respectively, 1.8 times longer to execute when the number of annotations was increased from 100 to 130% annotations, 130 to 160% annotations, and respectively, from 160 to 200% annotations.

Empirical Conclusions 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 post-processing time of the queries is proportional to the number and size of emitted tuples. Also, the execution times of default queries on databases annotated up to 100% are comparable to the performance of SQL queries since only one SPJ is generated and the number of annotations in a database does not have a major influence in the execution time of pSQL queries in this case. The execution time of each query for both default and default-all scheme increases marginally when the number of annotations in the database is, for example, doubled from 30% annotations to 60%. However, the performance of pSQL queries starts to degrade significantly on databases with more than 100% annotations. This indicates that our naive storage scheme is perhaps not the best suited in such conditions. As future work, we plan to investigate the trade-offs between the naive storage scheme and other possible storage schemes which we briefly discuss in Sect. 6.1.

6 Discussion

6.1 Other possible schemes for managing annotations

Besides our naive storage scheme, there are other possible schemes for storing and managing annotations. We briefly discuss two of them next.

Annotation-Relation Storage Scheme In this scheme, annotations of a relation R are stored in a separate relation RA, which we call the '*annotation-relation* of R'. The basic schema of RA has three attributes (id, attribute, annotation) where an id value uniquely identifies a tuple in R, a name value is an attribute name in the schema of R and an annotation value is an annotation of the location (id, name). An id can either be the primary key of relation R, in which case RA may have more than three attributes, or some unique identifier used in the database system (e.g., *rowid* in Oracle). For example, to store the tuples { $(a\ \{a_1, a_2\}, b, \{b_1\})$, (c, d) } of the relation $R(A, B)$ with A as the key of the relation, we would have an annotation-relation RA(id, attribute, annotation) with the following tuples: (a, A, a_1), (a, A, a_2), (a, B, b_1).

We have yet to investigate the trade-offs between the naive scheme and annotation-relation scheme. However, we expect that the annotation-relation scheme may require less storage space than the naive scheme in general. On the other hand, one needs to pay a performance penalty in using the annotation-relation scheme as a join between R and RA is required to retrieve the relevant annotations of a location in R.

Nested Sets Approach It is easy to observe that the multiplicity of a tuple in the naive storage scheme depends on the

number of annotations associated with that tuple. Instead, a more natural approach would have been to store annotations associated with each location as nested sets (i.e., the relation $R(A, B)$ would be stored as $R'(A, A_a, B, B_a)$, where A_a and B_a are of type nested set). Unfortunately, the nested set approach is not currently feasible, since not all commercial databases support nested sets and among those who do, none offers satisfactory support for the operations we need. As an example, in Oracle $10g$ the annotation union operation (i.e., the operation of merging duplicate tuples and their corresponding annotations together) is not direct and has to be performed in several steps.

6.2 Extensions

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 [30] that two conjunctive queries are equivalent under bag semantics if and only if they are isomorphic. This result of [30] 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 this extended framework we are interested in determining which annotations to propagate under different operators. We would also like to investigate whether our results for the default-all propagation scheme still hold.

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.

Appendix

Generating a query basis for pSQL queries

To generate a query basis for a pSQL query $Q = Q_1 \cup \ldots \cup Q_l$ where each Q_i, $i \in [1, l]$, is a pSQL query fragment with default-all propagation scheme, we modify Generate-Query-Basis algorithm described in Sect. 3 to the following algorithm, called Generate-Containment-Basis. Step 1 of Generate-Containment-Basis remains the same as in Generate-Query-Basis. The algorithm Generate-Containment-Basis differs from Generate-Query-Basis in Step 2, where for each pSQL query fragment Q_i ($1 \le i \le l$), the set of all queries that are *contained* in Q_i are generated and added to the auxiliary queries of Q_i. A consequence of this effect is that a query that is identical to Q_i but with an additional relation R that does not occur in Q_i is considered as a query contained in Q_i. Annotations from R may propagate to the output. In contrast, Step 2 of Generate-Query-Basis generates a set of auxiliary pSQL query *fragments* that are each equivalent to Q_i. Note that we are not restricting our language to be pSQL query fragments here (as opposed to algorithm Generate-Query-Basis). We are computing a query basis for the set of all pSQL queries that are each equivalent to a given pSQL query. We describe the algorithm next and then an example.

Algorithm Generate-Containment-Basis
Input: A pSQL query $Q = Q_1 \cup \ldots \cup Q_l$ with default-all propagation scheme.
Output: A query basis of Q, $\mathcal{B}(Q)$.

Let Q be a pSQL query of the form $Q_1 \cup \ldots \cup Q_l$ where each Q_i, $i \in [1, l]$, is a pSQL query fragment of the form shown in Definition 1 with `PROPAGATE DEFAULT-ALL` clause. For each Q_i, $i \in [1, l]$, we execute the following two steps.

1. *Generate Q_0^i, the representative query of Q_i.*
   Generate a query Q_0^i that is identical to Q_i except that the propagation scheme of Q_i is replaced with the following propagation scheme:
   For every attribute "$r.A$ AS C" in the *selectlist*, add "$r.A$ TO C" to 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" to 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^i.*
   Initialize $\mathcal{B}(Q_i)$ to the empty set. Add Q_0^i to $\mathcal{B}(Q_i)$.
   For every clause "$s.A$ AS B" in the *selectlist* of Q_0^i, for every relation R in the database and every attribute C in the relation schema of R:
   Create a query Q' that is identical to Q_0^i. Add "R r" to the *fromlist* of Q' where r is a tuple variable that does not occur in Q'. Add the condition "$r.C = s.A$" to the *wherelist* of Q' and the propagate clause "$r.C$ TO B" to the *propagatelist* of Q'. Add Q' to $\mathcal{B}(Q_i)$.
   (The query Q' is contained in Q_i but may propagate additional annotations. Furthermore, $\bigcup_{q\in\mathcal{B}(Q_i)} q$ is equivalent to Q_i.)

Return $\mathcal{B}(Q_1) \cup ... \cup \mathcal{B}(Q_m)$.

Example 11 Assume that the database consists of the following relations: Emp(name, dept), Dept(did, budget), Project(proj, mgr). Consider following query Q which picks out employees who belong to some department.

```
SELECT      DISTINCT e.name AS Name
FROM        Emp e, Dept d
WHERE       e.dept = d.did
PROPAGATE   DEFAULT-ALL
```

Two of the queries generated by Step 2 of the algorithm Generate-Containment-Basis on Q are shown below:

Q_1:

```
SELECT      DISTINCT e.name AS Name
FROM        Emp e, Dept d, Project p
WHERE       e.dept = d.did AND p.proj = e.name
PROPAGATE   e.name TO Name, p.proj TO Name
```

Q_2:

```
SELECT      DISTINCT e.name AS Name
FROM        Emp e, Dept d, Project p
WHERE       e.dept = d.did AND p.mgr = e.name
PROPAGATE   e.name TO Name, p.mgr TO Name
```

The highlighted parts of Q_1 and Q_2 denote the additional relation, condition and propagate clauses added to the representative query Q_0 by Step 2 of the algorithm. Observe that Q_1 and Q_2 are queries that are contained in Q (but $Q_0 \cup Q_1 \cup Q_2$ is equivalent to Q). Furthermore, Q_1 propagates annotations on projects to the result and Q_2 propagates annotations from the names of managers to the result. Arguably, Q_1 should not have been generated since the annotations for projects are irrelevant for names of employees. The query Q_2, however, propagates the annotations for a manager to an employee name and this is desired since the manager and the employee have the same name (and are therefore referring to the same entity).

Some observations from the above example follow. First, since our language now allows for union, the query basis for Q contains more queries than the query basis generated by Generate-Query-Basis for the same query Q. This is because in the case of Generate-Query-Basis, we consider only pSQL query fragments that are equivalent to Q. In contrast here, we consider all pSQL queries that are equivalent to Q. The above example also suggests that a more refined method of generating a query basis is needed. Namely, one should only generate an auxiliary query if it propagates relevant annotations. In the above example, Q_2 is desired but not Q_1. To generate only auxiliary queries that propagate relevant annotations, one would require the knowledge of semantically equivalent attributes in a database to be kept in the system. Queries are then generated by adding the extra relation and equating only semantically equivalent attributes. In what follows, we shall assume that both Q_1 and Q_2 are generated.

Observe that by equating "$r.C = s.A$" in Step 2, we are assuming that all attributes have the same type. The algorithm Generate-Containment-Basis takes as input $Q = Q_1 \cup ... \cup Q_l$ and generates as output a query basis $\mathcal{B}(Q)$ which is $\mathcal{B}(Q_1) \cup ... \cup \mathcal{B}(Q_l)$. The following proposition is similar to Proposition 2, adapted for queries generated by algorithm Generate-Containment-Basis.

Proposition 4 *For every query Q' in the result of Generate-Containment-Basis(Q) (denoted as $\mathcal{B}(Q)$), $C(Q')$ is annotation-contained in $\bigcup_{q\in\mathcal{B}(Q)} q$.*

Proof Let $Q = Q_1 \cup ... \cup Q_l$ and let Q_0^i denote the representative query generated by Step 1 of the algorithm for query Q_i. We have $C(Q_0^i)$ is annotation-contained in $\bigcup_{q\in\mathcal{B}(Q_i)} q$ since $Q_0^i \in \mathcal{B}(Q_i)$ and $C(Q_0^i)$ is annotation-equivalent to Q_0^i according to Proposition 1.

Let Q' denote a query in $\mathcal{B}(Q)$ and Q' is not Q_0^i for every $i \in [1, l]$. That is, Q' is one of the auxiliary queries, generated by Step 2 of the algorithm. Let $C(Q')$ be of the form "$H(\bar{x}) : -S_1(\bar{y_1}), \ldots, S_n(\bar{y_n})$, equalities". Given any database D, let (s, i) be a location in D which corresponds to a location (t, j) in $C(Q')(D)$ on a valuation φ. So $S_k(\varphi(\bar{y_k})) = s$ for some $k \in [1, n]$ and $H(\varphi(\bar{x})) = t$ and $\bar{y_k}[i] = \bar{x}[j]$. There is also a valuation φ' for Q' and D which produces t. The valuation φ' is such that $\varphi'(r) = S(\varphi(\bar{y}))$ where r is a tuple variable in Q' and $S(\bar{y})$ is the corresponding subgoal in $C(Q')$ which represents the relation that r ranges over in Q'. So $\varphi'(r_1) = s$ for some tuple variable r_1 in Q' and the output tuple is t under φ' according to Q'. We show next that for every annotation propagated by Q', there is a query in $\mathcal{B}(Q)$ that would propagate the annotation in the same way.

Suppose Q' is in $\mathcal{B}(Q_i)$ for some $i \in [1, l]$ and $S_k(\bar{y_k})$ is a subgoal among the subgoals of $C(Q_0^i)$ where Q_0^i is the representative query generated by Step 1 of the algorithm Generate-Containment-Basis. (Recall that $C(Q')$ differs from $C(Q_0^i)$ in that it has an additional subgoal added by Step 2 of the algorithm.) Since $\bar{y_k}[i] = \bar{x}[j]$ and $S_k(\bar{y_k})$ is a subgoal among the subgoals of $C(Q_0^i)$, it must be that the attribute at position i of S_k (call it B) is equal to the attribute at position j in the *selectlist* of Q' (call it A) or transitively equal to A. Hence, there must be a clause "PROPAGATE

$r_1.B$ TO A" in the propagate clause of Q_0^i (and hence Q'). Therefore under the valuation φ', the annotations at (s, i) are part of the annotations at (t, j) according to Q' and D.

Suppose $S_k(\bar{y_k})$ is not a subgoal among the subgoals of $C(Q_0^i)$. That is, $S_k(\bar{y_k})$ is the subgoal that corresponds to the extra relation in the *fromlist*, added by Step 2 of algorithm Generate-Containment-Basis. Let the attribute at the ith position of S_k be C. Since $\bar{y_k}[i] = \bar{x}[j]$ and by Step 2 of the algorithm, it must be that the condition "$r_1.C = r_2.A$" is the added condition in the *wherelist* for some tuple variable r_2 that ranges over a relation in Q'. The clause "$r_1.C$ TO B" is the added propagate clause of Q' for some output attribute B in the *selectlist*. (Hence, "$r_2.A$ AS B" is among the *selectlist* of Q'.) Let the attribute at the jth position of the output be F. If B is the same as F, then the annotations at (s, i) are part of the annotations at (t, j) according to Q' and D under the valuation φ'. Suppose B is not equal to F. Since "$r_1.C = r_2.A$" and $\bar{y_k}[i] = \bar{x}[j]$ in $C(Q')$, it must be that $r_2.A$ is equal or transitively equal to the output attribute F (according to Q_0^i). In other words, let "$r_3.E$ TO F" be the select clause for F (which is among the *selectlist* of Q_0^i) where r_3 is a tuple variable in Q_0^i and E is an attribute. We have $r_3.E$ is either equal or transitively equal to $r_2.A$ according to the *wherelist* of Q_0^i. Hence, the following query Q'' from Q_0^i will be generated: Q'' has an extra relation "$S_k\ r_1$" in the *fromlist*, the added condition "$r_1.C = r_3.E$" in the *wherelist* and the added propagate clause "$r_1.C$ TO F". Since under φ', we have $r_1.C = r_2.A$ and $r_2.A$ is equal or transitively equal to $r_3.E$, it follows that $r_1.C = r_3.E$. Therefore the valuation φ' is also a valuation for Q''. Hence, the annotations at (s, i) are part of the annotations at (t, j) according to Q'' and D. □

Our proof for the following Lemma 2 uses a result from [31].

Fact 2 ([31]) Let $Q = \bigcup_{i \in [1,m]} Q_i$ and $Q' = \bigcup_{i \in [1,n]} Q'_i$ be unions of conjunctive queries. Then $Q \subseteq Q'$ if and only if for every Q_i, $i \in [1, m]$, there exists Q'_j, $j \in [1, n]$, such that $Q_i \subseteq Q'_j$.

Lemma 2 *Let $\mathcal{B}(Q)$ denote the result produced by the algorithm Generate-Containment-Basis(Q), where Q is a pSQL query and let Q' denote a pSQL query 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 Let $Q = Q_1 \cup \ldots \cup Q_l$, $\bigcup_{q \in \mathcal{B}(Q)} q = q_1 \cup \ldots \cup q_m$ and let $Q' = Q'_1 \cup \ldots \cup Q'_n$.

We shall show next that for every distinguished variable x at the ith position in the head of $C(Q'_f)$ where f in $[1, n]$ and its occurrence at the jth position of the kth subgoal $S(\bar{u})$ (i.e., the jth variable for $\bar{u}$ is x) of $C(Q'_f)$, there is a generated query $q_g \in \mathcal{B}(Q)$ such that there is a homomorphism $h : C(q_g) \rightarrow C(Q'_f)$ that satisfies conditions (1) and (2) of Fact 1. Then by the Fact 1, we have $C(Q'_1) \cup \ldots \cup C(Q'_n) \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} C(q)$. For every pSQL query fragment Q, it is the case that $Q \subseteq_a C(Q)$. So we have $Q' \subseteq_a C(Q'_1) \cup \ldots \cup C(Q'_n)$ and therefore $Q' \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} C(q)$. By Proposition 4, $\bigcup_{q \in \mathcal{B}(Q)} C(q) \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} q$. Hence, we have $Q' \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} q$, which was to be shown.

Pick a distinguished variable x at the ith position in the head of $C(Q'_f)$ where $f \in [1, n]$ and at the jth position of the kth subgoal $S(\bar{u})$ of $C(Q'_f)$. That is, x occurs at the ith position in $H(\ldots)$, S is the kth subgoal and x occurs at the jth position in $S(\ldots)$.

$C(Q'_f) : H(\ldots x \ldots)$:- $\ldots, S(\bar{z_1}, x, \bar{z_2}), \ldots$

By Fact 2, since $C(Q)$ is equivalent to $C(Q')$, there exists a query $C(Q_g)$ for some $g \in [1, l]$ such that $C(Q'_f) \subseteq C(Q_g)$. Consequently, there is a containment mapping $h' : C(Q_g) \rightarrow C(Q'_f)$. Accordingly, we know that Generate-Containment-Basis would generate a query q_g according to Q_g such that $C(q_g)$ is identical to $C(Q_g)$ but has an additional subgoal $S(\bar{w_1}, y, \bar{w_2})$ where $\bar{w_1}$ and $\bar{w_2}$ are vectors of fresh variables that do not occur elsewhere in $C(q_g)$. That is, $C(q_g)$ has the form shown below where y is the distinguished variable that occurs in the ith position in the head of $C(Q_g)$ and the j position in $S(\ldots)$.

$C(q_g) : H(\ldots y \ldots)$:- body of $C(Q_g)$, $S(\bar{w_1}, y, \bar{w_2})$.

It is easy to see that there is a homomorphism $h : C(q_g) \rightarrow C(Q'_f)$ that satisfies conditions (1) and (2) of Fact 1. The homomorphism h is such that $h(x) = h'(x)$ for every $x \in \text{var}(Q_g)$, $h(\bar{w_1}) = \bar{z_1}$, and $h(\bar{w_2}) = \bar{z_2}$. Clearly, h is consistent with h' and is a homomorphism from $C(q_g)$ to $C(Q'_f)$. Hence, by Fact 1, $C(Q'_f) \subseteq_a C(q_g)$. □

Theorem 2 *Given a pSQL query Q, the algorithm Generate-Containment-Basis(Q) generates a query basis of Q.*

Proof Let Q be a pSQL query $Q_1 \cup \ldots \cup Q_l$. Let Q_R denote the union of all queries in $\mathcal{B}(Q)$. That is, $Q_R = \bigcup_{q \in \mathcal{B}(Q)} q$. Clearly, Q_R is contained in Q since each $\mathcal{B}_i(Q_i)$, $i \in [1, l]$, contains a representative query which is equivalent to Q_i and every other query in $\mathcal{B}_i(Q_i)$ is contained in Q_i. The query Q is also contained in Q_R since for every Q_i, $i \in [1, l]$, Q_i is equivalent to the representative query of Q_i in Q_R. Let $\mathcal{E}(Q)$ denote the set of all equivalent queries of Q where each query in $\mathcal{E}(Q)$ propagates using the default scheme. Since Q_R is equivalent to Q, we have $Q_R \in \mathcal{E}(Q)$. Hence, $Q_R \subseteq_a \bigcup_{q \in \mathcal{E}(Q)} q$. From Lemma 2, we know that for every query $q \in \mathcal{E}(Q)$, we have $q \subseteq_a Q_R$. Therefore $\bigcup_{q \in \mathcal{E}(Q)} q \subseteq_a Q_R$ and hence $Q_R =_a \bigcup_{q \in \mathcal{E}(Q)} q$. □

Acknowledgements We thank Xinyu Hua for her help during the initial implementation of this system and Ariel Fuxman for helpful suggestions. We also thank the reviewers for their helpful suggestions. Supported in part by NFS CAREER Award IIS-0347065 and NFS grant IIS-0430994.

References

1. Apweiler, R., Bairoch, A., Wu, C., Barker, W., Boeckmann, B., Ferro, S., Gasteiger, E., Huang, H., Lopez, R., Magrane, M., Martin, M., Natale, D., O'Donovan, C., Redaschi, N., Yeh, L.: Uniprot: the universal protein knowledgebase. Nucleic Acids Res. **32**, D115–D119 (2004)
2. Bairoch, A., Apweiler, R.: The SWISS-PROT protein sequence database and its supplement TrEMBL. Nucleic Acids Res. **28**, 45–48, 2000
3. DBCAT, The Public Catalog of Databases. http://www.infobiogen.fr/services/dbcat/. Cited 5 June 2000
4. Denning, D.E., Lunt, T.F., Schell, R.R., Shockley, W.R., Heckman, M.: The seaview security model. In: Proceedings of the IEEE Symposium on Security and Privacy, Washington, DC, pp. 218–233, (1988)
5. Jajodia, S., Sandhu, R.S.: Polyinstantiation integrity in multilevel relations. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, pp. 104–115, (1990)
6. Myers, A.C., Liskov, B.: A decentralized model for information control. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), Saint-Malo, France, pp. 129–142, (1997)
7. Tan, W.: Containment of relational queries with annotation propagation. In: Proceedings of the International Workshop on Database and Programming Languages (DBPL), Potsdam, Germany, pp. 3‘7–53, (2003)
8. Lee, T., Bressan, S., Madnick, S.: Source attribution for querying against semi-structured documents. In: Workshop on Web Information and Data Management (WIDM), Washington, DC (1998)
9. Wang, Y.R., Madnick, S.E.: A polygen model for heterogeneous database systems: The source tagging perspective. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), Brisbane, Queensland, Australia, pp. 519–538, (1990)
10. Cui, Y., Widom, J., Wiener, J.: Tracing the lineage of view data in a warehousing environment. ACM Trans. Database Syst. (TODS) **25**(2), 179–227 (2000)
11. Buneman, P., Khanna, S., Tan, W.: Why and where: A characterization of data provenance. In: Proceedings of the International Conference on Database Theory (ICDT), London, United Kingdom, pp. 316–330, (2001)
12. Bernstein, P., Bergstraesser, T.: Meta-data support for data transformations using microsoft repository. IEEE Data Eng. Bull. **22**(1), 9–14 (1999)
13. Maier, D., Delcambre, L.: Superimposed information for the internet. In: Proceedings of the International Workshop on the Web and Databases (WebDB), Philadelphia, Pennsylvania, pp. 1–9, (1999)
14. Kahan, J., Koivunen, M., Prud'Hommeaux, E., Swick, R.: Annotea: An open rdf infrastructure for shared web annotations. In: Proceedings of the International World Wide Web Conference(WWW10), Hong Kong, China, pp. 623–632, (2001)
15. LaLiberte, D., Braverman, A.: A protocol for scalable group and public annotations. In: Proceedings of the International World Wide Web Conference(WWW3), Darmstadt, Germany (1995)
16. Phelps, T.A., Wilensky, R.: Multivalent documents. In: Proceedings of the Communications of the Association for Computing Machinery (CACM) **43**(6), 82–90 (2000)
17. Schickler, M.A., Mazer, M.S., Brooks, C.: 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)
18. W3C. Annotea Project. http://www.w3.org/2001/Annotea
19. biodas.org. http://biodas.org.
20. Dowell, R.: A distributed annotation system. Technical report, Department of Computer Science, Washington University in St. Louis (2001)
21. Kent, W.J., Sugnet, C.W., Furey, T.S., Roskin, K.M., Pringle, T.H., Zahler, A.M., Haussler, D.: The human genome browser at UCSC. Genome Res. **12**(5), 996–1006 (2002)
22. Phelps, T.A., Wilensky, R.: Multivalent annotations. In: Proceedings of the First European Conference on Research and Advanced Technology for Digital Libraries, Pisa, Italy, pp. 287–303, (1997)
23. Phelps, T.A., Wilensky, R.: Robust intra-document locations. In: Proceedings of the International World Wide Web Conference(WWW9), Amsterdam, The Netherlands, pp. 105–118, (2000)
24. Buneman, P., Khanna, S., Tan, W.: On propagation of deletions and annotations through views. In: Proceedings of the ACM Symposium on Principles of Database Systems (PODS), Wisconsin, Madison, pp. 150–158, (2002)
25. Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases. Addison-Wesley Co., Reading, MA (1995)
26. Kementseitsidis, A., Arenas, M., Miller, R.J.: Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), San Diego, CA, pp. 325–336, (2003)
27. Tan, W.: Containment of relational queries with annotation propagation. Technical report, Department of Computer Science, UC Santa Cruz (2003)
28. Chiticariu, L., Tan, W.-C., Vijayvargiya, G.: DBNotes: A post-it system for relational databases based on provenance. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD) '05, pp. 942–944, (2005)
29. TPC Transaction Processing Performance Council. http://www.tpc.org
30. Chaudhuri, S., Vardi, M.Y.: Optimization of *real* conjunctive queries. In: Proceedings of the ACM Symposium on Principles of Database Systems (PODS), Washington, DC, pp. 59–70, (1993)
31. Sagiv, Y., Yannakakis, M.: Equivalence among relational expressions with union and difference operators. J. Assoc. Comput. Machine. (JACM) **27**(4), 633–655 (1980)