Supporting Annotations on Relations *

Mohamed Y. Eltabakh, Walid G. Aref, Ahmed K. Elmagarmid, Mourad Ouzzani, Yasin N. Silva
Computer Science Department, Purdue University
{meltabak, aref, ake, mourad, ysilva}@cs.purdue.edu

ABSTRACT

Annotations play a key role in understanding and curating databases. Annotations may represent comments, descriptions, lineage information, among several others. Annotation management is a vital mechanism for sharing knowledge and building an interactive and collaborative environment among database users and scientists. What makes it challenging is that annotations can be attached to database entities at various granularities, e.g., at the table, tuple, column, cell levels, or more generally, to any subset of cells that results from a select statement. Therefore, simple comment fields in tuples would not work because of the combinatorial nature of the annotations. In this paper, we present extensions to current database management systems to support annotations. We propose storage schemes to efficiently store annotations at multiple granularities, i.e., at the table, tuple, column, and cell levels. Compared to storing the annotations with the individual cells, the proposed schemes achieve more than an order-of-magnitude reduction in storage and up to 70% saving in the query execution time. We define types of annotations that inherit different behaviors. Through these types, users can specify, for example, whether or not an annotation is continuously applied over newly inserted data and whether or not an annotation is archived when the base data is modified. These annotation types raise several storage and processing challenges that are addressed in the paper. We propose declarative ways to add, archive, query, and propagate annotations. The proposed mechanisms are realized through extensions to the standard SQL. We implemented the proposed functionalities inside PostgreSQL with an easy to use Excel-based front-end graphical interface.

1. INTRODUCTION

The growth in scientific information has made databases integral to many scientific disciplines including physics, earth and atmospheric sciences, chemistry, and biology. These disciplines pose new data management challenges to current DBMSs. One of the key challenges is to overcome the limited ability of database systems in manipulating annotations and provenance data. Annotations play a key role in understanding and curating scientific databases as well as databases in other domains. Annotations may represent comments, descriptions, warning or error messages, questions about any subset of the data, lineage information, among several others. Therefore, annotation management becomes a vital mechanism for sharing knowledge and building an interactive and collaborative environment among database users and scientists.

Annotation management involves several challenges including: (1) ***Handling multi-granular annotations.*** Annotations can be large in size and attached to the data at various granularities, e.g., cell, tuple, column, or table. Therefore, efficient storage schemes are needed to avoid replicating the annotations. The storage overhead becomes more critical in the context of coarse-granular annotations such as provenance where one provenance record can be attached to many tuples or even entire columns or tables. (2) ***Propagating annotations seamlessly.*** Users want to propagate the annotations without complicating their queries. If annotation propagation is delegated to users (or applications) without any database system support, then users' queries may become complex and user-unfriendly. (3) ***Adding annotations and defining behaviors.*** Key requirements in annotation management are to provide declarative mechanisms to annotate the data and to specify how annotations behave under different database operations. For example, users may want the newly inserted data to be annotated automatically if it satisfies certain criteria, or may want annotations to be deleted automatically when the base data gets modified. Annotation management systems need to efficiently support these various behaviors. Providing interfaces to help users manage the annotations is important. However, without adequate support from the underlying database system, it becomes very complex and inefficient to design these interfaces. Annotation management in relational databases has been addressed in previous systems, e.g., [4, 2, 10, 15]. However, we believe that several of the highlighted challenges are still open and require further investigation.

In [7, 8], we introduce *bdbms* as an extended database system for scientific data management. *bdbms* extends the functionalities of current database management systems to include: (1) annotation management, (2) tracking depen-

*This research was partially supported by NSF Grant Number IIS-0811954 and by NIH Grant Number NIGMS U24 GM077905 for the EcoliHub project.

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

dencies that involve external modules among data items, (3) authorizing database operations based on the content of the data in addition to the identity of the user, and (4) supporting novel and non-traditional access methods. [7] presents the overall system and the challenges involved in each of the proposed components. In this paper, we focus on the annotation management component. We provide compact representations of annotations that significantly reduce the storage overhead and I/O cost of queries. We define three types of annotations: *snapshot*, *view*, and *join* annotations that inherit different behaviors. In contrast to *snapshot annotations* that apply to a data instance, *view annotations* automatically annotate newly inserted data if they satisfy certain conditions, whereas *join annotations* are attached to data across relations. These annotation types raise several storage and query processing challenges that are addressed in the paper. Since most scientists prefer to use graphical interfaces over using direct SQL commands, we provide an easy to use and intuitive GUI using Excel sheets that facilitate performing the proposed functionalities [8].

The contributions of this paper are summarized as follows:

1. We propose declarative mechanisms to support adding, storing, archiving, and querying snapshot, view, and join annotations. We propose query processing techniques and optimizations to efficiently support these annotation types.

2. We propose the *Mapped-Space* storage scheme that allows efficient and compact representation of multi-granular annotations. The Mapped-Space scheme achieves more than an order of magnitude saving in storage over the straightforward scheme where annotations are replicated and stored with each individual cell. The Mapped-Space scheme reduces query execution times by up to 70%.

3. We define new constructs such as *ON UPDATE PROPAGATE* and *ON AGGREGATION PROPAGATE* that allow users to control how annotations behave under different database operations. All the proposed constructs and functionalities are realized inside PostgreSQL. The experimental results are drawn from this realization. Although not part of this paper, the system has an Excel-based front-end graphical user interface [8].

The rest of the paper proceeds as follows. Section 2 overviews the related work. Section 3 introduces the *Mapped-Space* storage scheme. Sections 4, 5, and 6 present mechanisms for adding, archiving, querying and propagating annotations, respectively. The performance analysis is presented in Section 7. Section 8 contains concluding remarks.

2. RELATED WORK

Annotation management has been the focus of recent research as a key requirement in supporting scientific databases as well as many other database applications [4, 11, 13, 14]. Commercial databases such as Oracle and DB2 have added new features and functionalities inside the database engine to support life science applications [1, 12] such as accessing data stored in heterogeneous data sources, integrating varieties of data types, and embedding/integrating

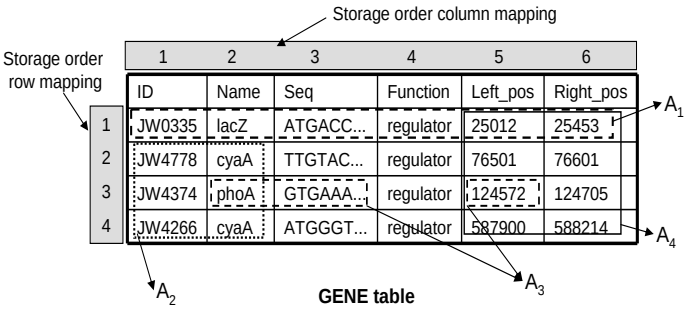


Figure 1: Example of annotations

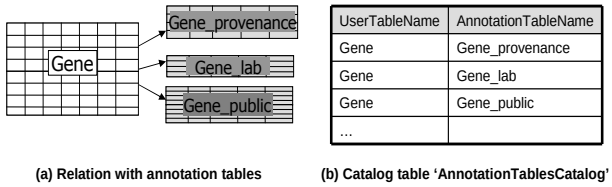


Figure 2: Annotation tables

data mining techniques inside the database engine. However, managing annotations has not been addressed yet by these systems.

Annotation management in the context of relational databases has been addressed in previous works, e.g., [2, 5, 6, 10, 15]. The two main systems are DBNotes [2, 6, 3], and MONDRIAN [9, 10]. DBNotes proposes an extension to SQL, termed *pSQL*, that extends the querying capabilities by adding a PROPAGATE clause to the SELECT statement that allows users to specify how to propagate the annotations along with the query answers. DBNotes proposes several propagation schemes that allow propagating the annotations on equivalent values resulted from operations such as union and equi-joins. MONDRIAN [9, 10] proposes an algebra, termed *color algebra*, that extends annotating single values to annotating multiple related values with the same annotation. Both systems allow users to add conditions to restrict the propagated annotations and to query the data based on the annotation values. The storage mechanisms in DBNotes and MONDRIAN are straightforward where annotations on Table R are stored in a separate table RA that references tuples' unique identifiers and column names in R.

DBNotes and MONDRIAN systems put notable effort on querying and propagating the annotations. The key distinctions between the proposed system and the previous systems are, as highlighted in Section 1, addressing storage optimization techniques, processing and managing different types of annotations, and proposing mechanisms for adding and archiving annotations.

3. STORAGE OF ANNOTATIONS

The size of annotations can be very large and may even exceed the size of the base data in the database. As a result, the retrieval and propagation of annotations at query time may involve overhead higher than that for retrieving the base data itself. Optimizing the storage of annotations is not straightforward since annotations are attached to the data at various granularities such as one cell, entire row or column, entire table, or an arbitrary set of cells.

In this section, we propose a storage scheme, called

CREATE ANNOTATION TABLE <*ann_table_name*>
ON <*user_table_name*>

Figure 3: Extended SQL command CREATE

Annotation Id	Curator	Timestamp	Annotation Value	Annotation CoveredCells	Archived Flag	OnUpdate PropagateFlag	OnAggregation PropagateFlag	ViewAnnotation Flag
10	Admin		A1	((1,1),(1,6))	F	F	F	F
20	Admin		A2	((1,2), (2,4))	F	F	F	F
30	User		A3	((2,3), (3,3))	F	F	F	F
30	User		A3	((5,3), (5,3))	F	F	F	F
40	User		A4	((5,1), (6,4))	F	F	F	F

Example of Gene_public annotation table

Figure 4: Structure of annotation tables

Mapped-Space scheme, that is based on mapping the columns and rows in a given table into an ordered domain, e.g., integer values. A table in the *Mapped-Space* scheme is viewed as a two-dimensional space where columns represent the X-axis and rows represent the Y-axis as illustrated in Figure 1. A cell in column C and row R is mapped to point (ColumnMapping(C), RowMapping(R)) in the two-dimensional space, where *ColumnMapping()* and *RowMapping()* are the column and row mapping functions, respectively. For example, the cell with value 'lacZ' is mapped to point (2,1) according to the mapping illustrated in Figure 1.

The *Mapped-Space* scheme gives us the opportunity to store multi-granular annotations efficiently where annotations on a set of cells are represented as rectangles covering the two-dimensional points corresponding to the annotated cells. The advantage of this scheme is that one rectangle may cover a single cell, entire row or column, or any set of consecutive cells without the need to replicate the annotation with every annotated cell. A single annotation is represented by one or more maximum-bounding rectangle(s) covering the target cells to be annotated. For example, annotations A_1 and A_2 in Figure 1 are each represented by a single rectangle while annotation A_3 is represented by two rectangles.

The *Mapped-Space* scheme avoids replicating the annotations whenever possible. However, the gain in storage compression and query performance depends on the quality of the mapping from the logical rows and columns to an ordered domain in the two-dimensional space. At the creation time of each table, there is no prior knowledge on which mapping is more efficient and hence we start with an initial mapping termed the *storage_order* mapping. The Storage_Order mapping is then improved by collecting statistics on how annotations are attached to rows and columns.

3.1 Annotation Tables

At the conceptual level, annotations are separate tables called *annotation tables* that have a pre-defined structure (See Figure 4). Each user relation may have one or more annotation tables attached to it. For example, in Figure 2, Relation *Gene* has three annotation tables *Gene_lab*, *Gene_public*, and *Gene_provenance* attached to it.

Annotation tables act as place holders that database developers create to organize and categorize the annotations over the database. For example, *Gene_lab* may store the annotations from lab members, *Gene_public* may store annotations from the public, while *Gene_provenance* may store the provenance of Gene's data. Annotation tables also hide the complexity of the underlying storage scheme because in most cases end-users will only need to reference the names of the annotation tables for adding and querying the annotations as will be explained.

To create an annotation table for a given user relation, the *CREATE ANNOTATION TABLE* command (Figure 3) is used. The structure of an annotation table is illustrated in Figure 4. Each annotation has a unique identifier *AnnotationId* that is attached to all rectangles representing this annotation, the curator who added the annotation, the timestamp, and the rectangle that represents the covering area of the annotation. The flags *ArchivedFlag*, *OnUpdatePropagateFlag*, *OnAggregationPropagateFlag*, and *ViewAnnotationFlag*, specify the behavior of the annotation under different operations as will be described in Sections 4 and 5. The catalog table *AnnotationTablesCatalog* (Figure 2(b)) is used to track the annotation tables defined in the system and on which user relations.

3.2 Storage_Order Mapping

The *Storage_Order* mapping is used as the default mapping when a table is created. In *Storage_Order* mapping, columns are mapped to sequential numbers based on the physical order of columns inside the table, i.e., the order of columns in the CREATE TABLE statement. Rows are mapped to sequential numbers based on their insertion order, i.e., a unique tuple identifier that is incremented with each insertion. The rows' mapping is stored in an additional column, called *Tuple_OID*, which is automatically added to each table at the creation time. An example of the *Storage_Order* mapping is illustrated in Figure 1 and the corresponding annotation table is presented in Figure 4.

Storage_Order mapping is simple and easy to implement. However, other mappings may be more efficient. For example, if the *Name* and *Function* columns in table Gene always get annotated together, then mapping these two columns into adjacent values in the mapping domain will result in a more compact representation of annotations, and hence less storage overhead and less I/O cost at query time. This is because every annotation on these two columns can be represented by one rectangle instead of two. The following section discusses an improvement over the Storage_Order mapping.

3.3 Correlated_Columns Mapping

In order to improve the columns' mapping, we maintain statistics on how frequently columns are annotated together. Columns that are frequently annotated together should be mapped into adjacent ids in the mapping domain, if possible, independent of their physical order in the table.

A straightforward approach to track the distribution of annotations over columns is to store all possible combinations among columns, i.e., individual columns, pairs of columns, triplets of columns, etc. When a new annotation is added on a set of columns, we increment the counter of this columns' combination by one. These statistics are then used to determine which columns are highly correlated and hence it would be better to make them adjacent to each other in the mapping domain. The drawback of this approach is that it does not scale well because the storage is

Columns_Mapping Algorithm

Inputs:

- Matrix T_AnnColCorrelation

Outputs:

- Suggested mapping for columns in T

Steps:

1 - **For** each column in T
2 - Create a node that has left and right neighbor pointers set to NULL
3 - crnt_max_correlation = *Max_INT*
4 - **Loop**
5 - Get columns (C_m and C_n) corresponding to the highest correlation less than crnt_max_correlation
6 - **IF** (each of C_m and C_n has at least one empty neighbor pointer) THEN
7 - Link C_m and C_n
8 - **Else**
9 - C_m and C_n cannot be mapped to adjacent numbers
10 - **End IF**
11 - crnt_max_correlation = *T_AnnColCorrelation[C_m][C_n]*
12 - **Until** all nodes are linked

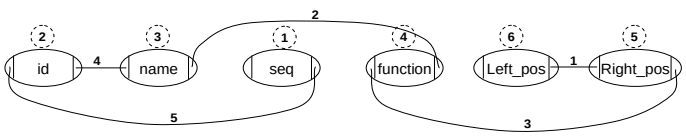
/* Assign the mapping Ids */

13 - Assign the last linked node a mapping id = 1
14 - **Loop**
15 - Follow the backward link to the previous node N
16 - Assign the next mapping id to node N
17 - **Until** the end of the list is reached

Figure 5: Algorithm of Correlated_Columns mapping

	Id	Name	Seq	Function	Left_pos	Right_pos
Id		80	50	20	10	0
Name			65	250	0	0
Seq				100	30	10
Function					5	230
Left_pos						380
Right_pos						

Correlation matrix for Gene table (Gene_AnnColCorrelation)



The order of linking the nodes based on the correlation values

Figure 6: Example of Correlated_Columns mapping

exponential in the number of columns in each table.

In order to collect the annotation correlation statistics more efficiently, we maintain for each table T a matrix named *T_AnnColCorrelation* (See Figure 6), where each dimension of the matrix represents the columns of T. *AnnColCorrelation* matrices are symmetric and hence only the upper triangle is maintained. With the insertion of each new annotation over certain columns, say T.x, T.y, and T.z, we increment the correlation of each pair of the annotated columns, e.g., increment T_AnnColCorrelation[x][y], T_AnnColCorrelation[x][z], and T_AnnColCorrelation[y][z]. As more annotations get inserted, columns that are frequently annotated together will have higher correlation than those of the other columns.

Given an instance of *T_AnnColCorrelation*, the best mapping for T's columns is derived using the algorithm presented in Figure 5. The algorithm iterates over the matrix values (Lines 4-12) and in each iteration it finds the pair of columns with the next highest correlation, e.g., (C_m , C_n). It creates a link between C_m and C_n, if possible, which means that C_m and C_n will be adjacent in the mapping domain (Lines 6-7). The algorithm terminates when all columns are linked together.

In Figure 6, we illustrate an example on how the *Correlated_Columns* mapping algorithm works. The labels on the edges specify the order of connecting the nodes together. Initially, each column has no neighbors. The highest correlated pair of columns is (Left_pos, Right_pos) and hence a link is added between the corresponding nodes. The next highest correlated pairs are (Name, Function) followed by (Function, Right_pos) and hence the links labeled by 2 and 3 are added between these pairs, respectively. The next highest correlated pair is (Seq, Function), however no link can be established between these two nodes because the Function node already has two neighbors. Therefore, this pair is skipped. The algorithm continues processing the pairs with the next highest correlation until all the nodes are linked. The final suggested mapping order is labeled with dotted circles in Figure 6.

Given the columns' mapping currently in place, say CM_1, and a new mapping suggested by the *Correlated_Columns* mapping algorithm, say CM_2, the system needs to decide whether or not to replace CM_1 by CM_2. The decision is a tradeoff between the cost of re-constructing all the annotations according to the new mapping, and the gain of reducing the storage overhead and I/O cost of queries. For this purpose, we define the *Columns Mapping Quality (CMQ)* metric to measure the relative quality between two mappings CM_1 and CM_2 and change the mapping only if the relative quality of CM_1 is below a certain threshold.

$$CMQ(CM_1/CM_2) = 100 * \frac{\sum T\_AnnColCorrelation[C_k][C_l],\ \forall adjacent\ pairs\ (C_k,C_l)\ in\ CM_1}{\sum T\_AnnColCorrelation[C_m][C_n],\ \forall adjacent\ pairs\ (C_m,C_n)\ in\ CM_2}$$

CMQ measures the relative quality of mapping CM_1 to CM_2 as the sum of the correlation of the adjacent columns in CM_1 over the sum of the correlation of the adjacent columns in CM_2. The less the $CMQ(CM_1/CM_2)$ percent, the less the quality of CM_1 compared to CM_2.

As an example, assume CM_1 is the Storage_Order mapping of columns in table Gene (Figure 1) and CM_2 is the mapping in Figure 6, then:

$$CMQ(CM_1/CM_2) = 100 * \frac{80+65+100+5+380}{50+80+250+230+380} = 64\%$$

If $CMQ(CM_1/CM_2)$ is less than a pre-defined threshold *CMQ_MIN*, then CM_1 is considered poor and replaced by CM_2.

It is important to point out that when the mapping changes, the physical structure of the table does not change. Only the mapping ids assigned to the columns get modified. Those mapping ids are stored in a catalog table named *ColumnsMappings(TableName, ColumnName, ColumnMappingId)*.

3.4 New Mapping: Re-construct Annotation Tables

We maintain counters in the database to track the number of added annotations to each table. With the addition of every m annotations to table T, the *Correlated-Columns* algorithm is triggered to check whether or not the current columns mappings of T need to be replaced. If the system

```
ADD ANNOTATION
[ AS VIEW ]
TO <annotation_table_names>
VALUE <annotation_body>
[ON AGGREGATION PROPAGATE]
[ON UPDATE PROPAGATE]
ON <select_statement> ;
```

Figure 7: ADD ANNOTATION command

Identify_and_Map_Target_Cells Procedure

Inputs:

- select_statement

Outputs:

- set of rectangles representing the target cells

Steps:

1- Extract the user relation from the *select_statement* → R
2- Extract the columns of R from the projection list → Set P = {c_1, c_2, c_3, ...}
3- Query the catalog table *ColumnsMappings* to map P and return a sorted set of mapping Ids → Set A
4- ColumLevelAnnotationFlag ← set to true if *select_statement* has no WHERE clause
5- Set B = {}
6- **IF** (ColumLevelAnnotationFlag = False) **THEN**
   /* specific rows are selected */
7- Execute the *select_statement* and identify the returned set of tuples ordered by the Tuple_OID → Set B
8- **END IF**
9- Call Construct_Maximum_Bounding_Rectangles (A, B, ColumLevelAnnotationFlag) that returns the set of rectangles covering the target cells

Figure 8: Identifying the target cells

decides to replace the current mapping CM_1 by a new mapping CM_2 for table T, then the annotation tables attached to T need to be scanned to re-construct the annotations. As a first step, we read once the catalog table *ColumnsMapping* to get the CM_1 mapping values. Then, for each annotation table AT attached to T, we retrieve the annotations ordered by the *AnnotationId* column such that multiple entries of the same annotation are grouped together. For each group (multiple rectangles of a single annotation), we map the current columns' mapping to the new mapping and form a new set of rectangles. Notice that the rows' mapping remains the same.

During the construction, the annotation table AT is *READ* locked, therefore it will be available for queries but not for insertion of new annotations. The newly formed rectangles are inserted into a temporary table AT_temp. After all annotation tables are processed, the catalog table *ColumnsMapping* is updated to reflect the CM_2 mapping and the annotation tables, e.g., AT, are dropped and the temporary ones, e.g., AT_temp, are renamed to become the current annotation tables.

4. ADDING ANNOTATIONS AT MULTIPLE GRANULARITIES

Adding annotations involves two main aspects, (1) specifying the target cells to be annotated, and (2) specifying the behavior of the annotation under various operations. We propose the *ADD ANNOTATION* command (Figure 7)

Construct_Maximum_Bounding_Rectangles Procedure

Inputs:

- Sorted set of columns' mapping → Set *A* (a_1, a_2, a_3, ...)
- Sorted set of tuples' mapping → Set *B* (b_1, b_2, b_3, ...)
- ColumnLevelAnnotationFlag

Outputs:

Set of MBRs covering only the points corresponding to *A* and *B*

Steps:

1- Scan set A sequentially and identify the maximal intervals of consecutive values.
   - Two values a_1 and a_2 are considered consecutive if $a_2 = a_1 + 1$
   - Each interval X has start value a_{min} and end value a_{max}
   - Since A is sorted, then finding all **$[a_{min}, a_{max}]$ intervals is done in one scan**

IF (ColumnLevelAnnotationFlag = True) **THEN**
  2- For each interval X, construct a covering rectangle L, where
     L = ((a_{min}, *Row_Min_Mapping*) , (a_{max}, *Row_Max_Mapping*))
ELSE
  3- Scan set B sequentially and identify the maximal intervals of consecutive values.
     - Two values b_1 and b_2 are considered consecutive if $b_2 = b_1 + 1$
     - Each interval Y has start point b_{min} and b_{max}
     - Since B is sorted, then finding all **$[b_{min}, b_{max}]$ intervals is done in one scan**

  4- For each pair of intervals X and Y, construct a covering rectangle L, where
     L = ((a_{min},b_{min}) , (a_{max}, b_{max}))
END IF

Figure 9: Constructing the rectangular representation

to annotate the data inside the database. The *annotation_table_names* parameter in the TO clause specifies the annotation tables(s) in which the annotation will be stored. The *annotation_body* parameter in the VALUE clause specifies the annotation value. The *select_statement* parameter is a simple SELECT-FROM-WHERE SQL query that does not contain aggregations or nested sub-queries in the FROM clause. The *select_statement* query specifies the target cells to be annotated. The annotation will be attached to the cells corresponding to the query answer.

The optional clauses in the ADD ANNOTATION command specify the behavior of the annotation. The AS VIEW clause specifies that the annotation will be evaluated continuously on the newly inserted or updated data, otherwise the annotation command is executed once of the current database instance. The ON AGGREGATION PROPAGATE clause specifies that at the time of querying the data, if the user query contains aggregation, e.g., GROUP BY or DISTINCT, then keep propagating the annotation along with the aggregated tuples, otherwise the annotation will not be propagated with the aggregated tuples. The ON UPDATE PROPAGATE clause specifies that if the base data to which the annotation is attached is modified, then keep the annotation attached to the new value, otherwise the annotation is archived. In the following subsections, we discuss each clause in more details.

4.1 ADD ANNOTATION Query Processing

The *select_statement* parameter is a simple SELECT-FROM-WHERE query. The FROM clause is restricted in this section to have a single relation and will be extended in Section 4.4 to have multiple relations. The WHERE clause may contain sub-queries without restrictions, e.g.,

contain joins and/or aggregations. The projection list in *select_statement* is limited only to column names, i.e., no functions are allowed in the projection list. The annotation tables in the *annotation_table_names* parameter are checked against the catalog table to make sure they correspond to the user relation(s) in the FROM clause of *select_statement*.

The select statement is a powerful declarative mechanism that enables users to specify the target cells to be annotated at various granularities. The following ADD ANNOTATION commands illustrate how to add the annotations presented in Figure 1, where A1 is a tuple-level annotation, A2 is a cell-level annotation, and A4 is a column-level annotation.

```
ADD ANNOTATION TO gene.Gene_lab VALUE A1
    ON (Select * From gene Where id = 'JW0335');

ADD ANNOTATION TO gene.Gene_lab VALUE A2
    ON (Select id, name From gene Where id like 'JW4%');

ADD ANNOTATION TO gene.Gene_lab VALUE A4
    ON (Select left_pos, right_pos From gene);
```

It is important to point out that the annotation is always attached to the base data inside the tables and not to the temporary query answer. The query is only a mechanism to select which data to annotate.

The target cells to be annotated are mapped to points in the two-dimensional space using the procedure presented in Figure 8. In Lines 1-2, we parse the *select_statement* parameter and extract the table name to which the annotation will be attached and the projection attributes. In Line 3, we query the catalog table *ColumnsMapping* to retrieve the mapping ids of the projected columns. If *select_statement* does not contain a WHERE clause, then the annotation is column-level (Line 4). If the annotation is column-level, then the *select_statment* will not be executed since all tuples will be annotated, otherwise Lines 6-8 are executed. In Line 7, we execute the select statement to identify which tuples satisfy the query predicates. The Tuple_OIDs of those tuples are returned sorted in set B. The columns' mapping (list A) and rows' mapping (list B) provide the mapping from the target cells to the corresponding points in the two-dimensional space. The maximum bounding rectangles that will represent the annotation are constructed by calling the procedure presented in Figure 9. In the first step of the procedure we group the consecutive ids of the columns' mapping into intervals, each interval is defined by start and end points. For example, if the ids are $\{1, 2, 3, 5, 7, 8, 9\}$, then Step 1 will generate three intervals [1-3], [5-5], and [7-9]. If the annotation is column-level, then Step 2 will be executed, otherwise, Steps 3-4 will be executed. In Step 2, we cross product interval [*Row_Min_Mapping*, *Row_Max_Mapping*] that covers the entire space of the rows' mapping ids with the intervals generated from Step 1 to produce the output rectangles. If the annotation is not column-level, then in Step 3 we group the consecutive ids of the rows' mapping into intervals, each interval is defined by start and end points. These intervals are cross product with the intervals generated from Step 1 to produce the output rectangles which will be stored in the annotation tables (Step 4).

4.2 Snapshot versus View Annotations

TablesLastModification

TableName	LastModificationTimestamp
Gene	2008-02-10 02:05:20
Gene_test	2008-02-15 02:50:37
Gene_Mutation	2008-03-10 03:20:20

ViewAnnotationsDependentTables

AnnotationId	TableName
1	Gene_test
1	Gene_Mutation

ViewAnnotationsTables

AnnotationId	AnnotationTable
1	Gene_lab

ViewAnnotations

AnnotationId	AnnotationCommand	TableName	LastExecution Timestamp
1	ADD ANNOTATION TO gene.Gene_lab VALUE A3 ON (Select name, seq, left_pos From gene Where id IN (Select gene_id from Gene_Test, Gene_Mutation Where test_id = mutation_id));	Gene	2008-03-15 07:15:10
...			

Figure 10: Annotations catalog tables

View annotations are defined by adding the *AS VIEW* optional clause to the ADD ANNOTATION command, otherwise the annotations are defined as *snapshot annotations*. For snapshot annotations, the ADD ANNOTATION command is executed once where the target cells are identified and annotated, and then the command is discarded. In this case, the annotation rectangles inserted into the annotation tables are marked with *ViewAnnotationFlag* set to False (Refer to Figure 4). In contrast, for view annotations, the ADD ANNOTATION command is executed and annotation rectangles are marked with *ViewAnnotationFlag* set to True. Moreover, we store the ADD ANNOTATION command inside the database to be continuously applied over newly inserted or modified tuples.

When view annotations are attached to a table T, and then T gets modified, e.g., a new tuple is inserted or an existing tuple is modified, then view annotations attached to T are incrementally re-evaluated on the new/modified tuple(s) (the new delta). Maintaining view annotations up-to-date is similar to maintaining materialized views up-to-date. The two main approaches are *eager* and *lazy* approaches. In the eager approach, view annotations are applied over the newly inserted or modified tuple at the time of the insertion or modification, respectively. Whereas in the lazy approach, view annotations are applied over the newly inserted or modified tuple(s) at the time of querying the annotations. In our system we deploy the lazy approach for the following reasons, (1) Lazy approach accommodates for multiple insertions or modifications at once. (2) The cost of re-evaluating the view annotations becomes a part of querying the annotations rather than part of the insertion or update operations. (3) Lazy approach allows for optimization such as distributing the refreshing overhead over multiple queries as discussed in Section 6.1.

To support view annotations, we create the catalog tables presented in Figure 10. *TablesLastModification* stores the timestamp of the last modification (insert, update, delete) on each user table in the database. *ViewAnnotations* stores a unique identifier for each view annotation, the annotation command, the table name to which the annotation is attached, i.e., the table name in the FROM clause, and the timestamp of the last execution of this command. *ViewAnnotationsDependentTables* stores an annotation identifier that references a specific annotation in ViewAnnotations,

and the table name(s) on which the annotation depends, i.e., table names in the WHERE clause sub-query (if any). Notice that an annotation is attached to one table but may depend on zero or more tables. *ViewAnnotationsTable* stores the names of the annotation tables related to each view annotation. For example, consider the following view annotation command:

ADD ANNOTATION AS VIEW TO gene.Gene_lab VALUE A3
ON (Select name, seq, left_pos From gene Where id In (Select gene_id from Gene_Test, Gene_Mutation Where test_id = mutation_id));

After executing the command, the records presented in Figure 10 will be inserted into the catalog tables. Notice that the stored command does not contain the *AS VIEW* clause, and hence when it is re-executed it will not be inserted into the catalog table again.

If the table to which the annotation is attached or the table(s) on which the annotation depends get modified, then the view annotation needs to be refreshed. The refreshing event takes place when the annotations are queried. In Section 6, we present mechanisms for refreshing view annotations as part of the annotation propagation and querying.

4.3 ADD ANNOTATION Update Behavior

When users' data get modified, what will happen to the annotations attached to the modified data? The system provides two options, either archive the annotations or keep the annotations attached to the data. The default behavior is to archive the annotations when the base data is modified. In this case when the annotation is inserted into the annotation tables, it will have the flag *OnUpdatePropagateFlag* set to False (Refer Figure 4). If the user wants to keep the annotation attached to the base data even if the data is modified, then the ON UPDATE PROPAGATE clause is included in the ADD command. In this case the flag *OnUpdatePropagateFlag* is set to True.

The ON UPDATE PROPAGATE clause is allowed only with snapshot annotations, that is, ON UPDATE PROPAGATE clause cannot be used in conjunction with AS VIEW clause. The reason is that view annotations already have a defined behavior under the update operations which is re-executing the view annotation on the modified tuple to check whether or not it is still satisfying the annotation query.

The actions taken when a cell in Table T, specified by Column c and Row r, is modified, are: (1) Find the annotation tables attached to T. This is performed by querying the catalog table *AnnotationTablesCatalog* (Figure 2(b)), and (2) For each annotation table AT, archive the snapshot annotations attached to the modified cell by issuing the following ARCHIVE command (See Section 5):

ARCHIVE ANNOTATION FROM *AT*
WHERE ViewAnnotationFlag = false And OnUpdatePropagateFlag = false And ArchiveFlag = false
ON (Select T.c From T Where T.Tuple_OID = r.Tuple_OID)

In some cases users may want to either archive or keep the annotations based on the new value of the updated cell. For instance, if the new value is greater than zero, then keep the annotation, otherwise archive the annotation. In such cases where the archiving decision is based on a condition, users can simply define the annotation to be propagated on update, i.e., include the ON UPDATE PROPAGATE clause in the ADD command. Then, using database triggers, users can check if the desired condition is true, then an explicit ARCHIVE ANNOTATION command is issued to archive the annotation.

ARCHIVE ANNOTATION
FROM <*annotation_table_names*>
[WHERE <*annotation_conditions*>]
ON <*select_statement*> ;

Figure 11: ARCHIVE ANNOTATION command

4.4 Annotations across Relations

In the previous sections, we restricted the ADD ANNOTATION command to annotate one relation at a time, i.e., only one relation is allowed in the FROM clause of the select statement. In this section, we extend our annotation mechanism to support annotations across relations, that is, annotations over joined tuples.

Annotations across relations (also referred to as *join annotations*) are handled in a way different from handling annotations on single relations because they have different semantics, storage requirements, and propagation mechanism. We implemented a straightforward approach to support *snapshot join annotations*. Supporting *view join annotations* and optimizing the storage and query processing for *join annotations* in general are left for future work.

***Semantics*:**

- A *Join annotation* A on relations T_1, T_2, ..., T_n means that A is attached to the joined tuples r_1, r_2, ..., r_n from those relations. Annotation A propagates along with the query answer only if T_1, T_2, ..., T_n are included in the query and tuples r_1, r_2, ..., r_n are joined together.
- *Join annotations* are tuple-level annotations and hence they propagate with the joined tuples r_1, r_2, ..., r_n regardless of which attributes are selected in the projection list.

Join Annotations are meant to be annotations that make sense only if several pieces of data came together from several tables. Therefore, they propagate only if these pieces of data appear in the query answer.

***Storage*:**

Join annotations need to be attached to the Tuple_OIDs of the joined tuples. Given an ADD ANNOTATION command with relations T_1, T_2, ..., T_n in the FROM clause, we assign a unique id to the annotation and insert it into a table named *JoinAnnotations* along with the curator name, timestamp, and the flags that specify the annotation behavior under the update and aggregation operations. Then, the ADD procedure executes the select statement to identify the Tuple_OIDs of the joined tuples. The annotation id and the resulted Tuple_OIDs are inserted into a table named *JoinAnnotationsTuples* which has one column for each table in the database. For a given annotation, we fill the Tuple_OIDs only in the columns corresponding to the annotated tables.

Join annotations over certain tables, e.g., T and R, can be expressed as selecting the tuples from *JoinAnnotationsTuples* where the columns corresponding to T and R are not null and their remaining columns are nulls.

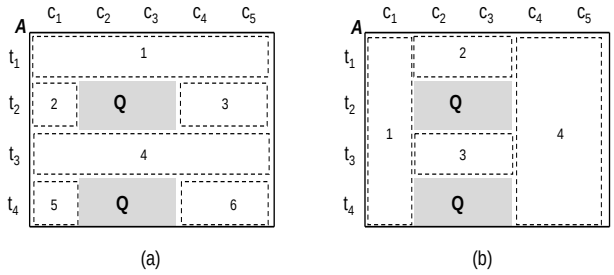


Figure 12: Re-constructing rectangles of archived annotations

SELECT [DISTINCT] C_i , C_j , ..., [PROMOTE (C_k, C_m, ...)]
FROM *Relation_name* [ANNOTATION(S_1, S_2, ...)], ...,
[JoinANNOTATION((R_i,R_j,..., R_k), ...)]
[WHERE <*data_annotation_conditions*>]
[GROUP BY <*data_columns*>
[HAVING <*data_annotation_condition*>]
[ORDER BY <*data_columns*>]

Figure 13: The Extended SELECT

5. ARCHIVING ANNOTATIONS

The archival mechanism is used to isolate outdated, invalid, or worthless annotations from recent and valuable ones. When annotations are first added to the database, the flag *ArchivedFlag* is set to False (Refer to Figure 4). When an annotation is archived, the flag *ArchivedFlag* is set to True. Archived annotations will not propagate to users along with query results.

The *ARCHIVE ANNOTATION* command presented in Figure 11 is used to archive annotations. The *annotation_table_names* parameter specifies the annotation tables from which annotations will be archived. The optional WHERE clause specifies conditions that archived annotations have to satisfy. These conditions reference the columns in annotation tables, e.g., *AnnotationId*, *Curator*, *Timestamp*, etc., (Refer to the structure of annotation tables in Figure 4). The cells over which annotations will be archived are specified using the *select_statement* parameter.

The first step in executing the ARCHIVE ANNOTATION command is to identify the target cells on which annotations will be archived. This is done by passing the *select_statement* to the Identify_and_Map_Target_Cells() procedure (Figure 8). The procedure returns a set of rectangles, referred to as *query rectangles*, representing the target cells. Then, for each annotation table AT in *annotation_table_names*, we update AT and set $AT.ArchivedFlag$ to True where $AT.AnnotationCeoveredCells$ overlaps with any of the query rectangles and the conditions in *annotation_conditions* are satisfied.

The archival procedure becomes a bit more complex when an annotation is partially archived, i.e., an annotation rectangle overlaps but not contained in a query rectangle. In this case the annotation needs to be broken down into multiple pieces. For example, consider annotation A_2 that is represented by one tuple in the Gene_public annotation table in Figure 4. If the *select_statement* selects a subset of the cells to which A_2 is attached, e.g., the cell corresponding to JW4778, then, in addition to archiving the record corresponding to A_2 in Gene_public, we need to construct two new rectangles for A_2 to cover the cells on which A_2 is not archived and insert them into Gene_public. For instance, create two new records with rectangles ((1,3),(2,4)) and ((2,2),(2,2)) and insert them into Gene_public.

Different objective functions may lead to different construction approaches of the new rectangles. Due to space limitation, we highlight that issue through an example without going into the details. Consider the example in Figure 12, the annotation rectangle is denoted by A and the query rectangles resulted from the ARCHIVE command are denoted by Q. A is initially covering tuples t_1 through t_4 and columns c_1 through c_5. Since A is not contained in Q, then A will be broken down into multiple maximum bounding rectangles. If the objective function is to minimize the storage overhead, then the division in Figure 12(b) is preferred because it generates less number of rectangles. In contrast, If the objective function is to minimize the query processing overhead, then the division in Figure 12(a) is preferred because each tuple will require less number of joins to propagate annotation A. For instance, tuples t_1 and t_3 will each require only one join with rectangles labeled 1 and 4, respectively, to propagate A with the tuples. Whereas, the division in Figure 12(b) will require three joins with each tuple to propagate A. In our system, we studied two objective functions, (1) minimizing the total storage overhead, and (2) minimizing the query processing overhead. The former objective function minimizes the total number of generated maximum bounding rectangles, while the latter one minimizes the average number of generated rectangles per tuple. The experimental analysis shows that the difference in the storage overhead between the two objective functions is relatively small compared to the total size of the annotations, and hence, they have comparable effect on the queries performance.

6. ANNOTATION PROPAGATION AND QUERYING

Annotation tables have pre-defined structure that enables compressed and compact representation of annotations. Because of that the join operation between a user table and an annotation table to propagate the annotations is not straightforward. The tuples in Table T are viewed as line segments in the form $((T.Tuple\_OID,\ 0),\ (T.Tuple\_OID,\ MaxCol))$, where $MaxCol$ is the number of columns in T. Annotations over a given tuple are the rectangles that intersect the tuple's line segment.

To enable efficient propagation and querying of annotations along with query answers we introduce the extended SELECT statement presented in Figure 13. The semantics of the new qualifiers are as follows. ANNOTATION, which may follow a relation's name in the FROM clause, specifies which annotation table(s) to consider in the query. For example, $R[ANNOTATION(A_1, A_2)]$ indicates propagating the annotations from annotation tables A_1 and A_2 that are attached to the user relation R. JoinANNOTATION, which may be added to the end of the FROM clause, specifies which *join annotations* to consider in the query. For example, *JoinANNOTATION((R,S), (T,W,Z))* indicates propagating annotations that are on joined tuples from R and S as well as annotations on joined tuples from the triplet T,

Actions_on_Table_Modification Procedure

Triggering Event: **Modification in table T**

Actions common to all operations (insert, update, delete):

```
1-  Update TablesLastModification set LastModificationTimestamp = now() Where TableName = 'T'
2-  Query table ViewAnnotationsDependentTables to get the set of annotation ids that
        depend on T → Set AnnIDs
3-  For each id in AnnIDs Loop
4-      For each annotation table AT Loop
5-          Update AT set ArchivedFlag= True where AnnotationId = id
6-      End Loop
7-  End Loop
```

Actions on inserting new tuple t:

```
8-  t.CreationTimestamp = now()
9-  t.LastModifiedTimestamp = now()
```

Actions on updating existing tuple t:

```
10- t.LastModifiedTimestamp = now()
11- For each annotation table AT on T Loop
12-     Execute an Archive Annotation command to archive view annotations attached to t
            "Archive Annotation From AT Where ViewAnnotaionFlag = True
            ON ( Select * from T where Tuple_OID = t.Tuple_OID)"
13- End Loop
```

Actions on deleting existing tuple t:

```
14- For each annotation table AT on T Loop
15-     Execute an Archive Annotation command to archive annotations attached to t
            "Archive Annotation From AT
            ON ( Select * from T where Tuple_OID = t.Tuple_OID)"
16- End Loop
```

Figure 14: Tables modifications Procedure

W, and Z. Those relations has to appear in the FROM clause. PROMOTE, which may be added to the projection list, propagates annotations from columns that are not in the projection list. As a result, annotations over non-projected attributes can be kept and propagated in the query pipeline. The WHERE and HAVING clauses are also extended to allow for annotation predicates as well as data predicates. If the extended qualifiers are not added, then the statement is considered as a standard SELECT statement that returns the query's answer without annotations.

Before executing an extended SELECT statement, a procedure for refreshing view annotations is called to make sure view annotations are up-to-date (Section 6.1). The statement is then passed to a re-writing phase that translates the new qualifiers into standard SQL operations (Section 6.2).

6.1 Refreshing View Annotations

View annotations will require refreshing or re-evaluation if any of the following two events occur. (1) The table to which the annotation is attached gets modified, or (2) The table(s) on which the annotation depends get(s) modified. In the former case view annotations will be evaluated incrementally, whereas in the latter case view annotations will be evaluated from scratch. For instance, referring to the view annotation example in Figure 10, if table Gene gets modified, then the view annotation can be re-evaluated only on the new or modified tuples. Whereas if tables Gene_Test or Gene_Mutation get modified, then the view annotation will

Refresh_View_Annotations Procedure

Triggering Event: **Querying annotation table *AT***

Actions:

```
1-  Query table ViewAnnoationTables to get the set of view annotations on AT → Set VA
2-  For each view annotation r in VA Loop
3-      Query table ViewAnnoationsDependentTables to get the set of tables on which r
            depends → Set r_depenet_tables
4-      Query table TablesLastModifications to get the count of tables in r_depenet_tables
            whose LastModificationTimestamp > r.LastExecutionTimestamp → depend_count
5-      IF (depend_count = 0) THEN
            /* r is re-evaluated incrementally */
6-          Query table TablesLastModifications to get the last modification time of T
                → Last_Modification_of_T
7-          IF (Last_Modification_of_T > r.LastExecutionTimestamp) THEN
8-              Modify r.AnnotationCommand by adding condition:
                    (T.LastModificationTimestamp > r.LastExecutionTimestamp)
                    to the WHERE clause
9-              Executed the modified command
10-         END IF
11-     ELSE
            /* r has to be re-evaluated from scratch */
12-         Execute r.AnnotationCommand
13-     END IF
14-     Update r.LastExecutionTimestamp = now()
15- End Loop
```

Figure 15: Refreshing view annotations procedure

be re-evaluated on all tuples in the Gene table even if the Gene table itself is not modified.

In Figure 14, we present the actions performed when Table T gets modified. In Line 1, we update *LastModificationTimestamp* of T in the catalog table *TablesLastModifications*. In Line 2, we get the view annotations that depend on T. These annotations will be archived since they will be re-evaluated from scratch when later referenced in a query (Lines 3-7). If the modification is an insert of Tuple t, then $t.CreationTimestamp$ and $t.LastModifiedTimestamp$ are both set to the insertion timestamp (Lines 8-9). If the modification is an update of Tuple t, then $t.LastModifiedTimestamp$ is set to the update timestamp (Line 10). We also archive any view annotation attached to t (Lines 11-13). Notice that in Lines 4-6 we directly modify the *ArchivedFlag* column, whereas in Lines 11-13 we use ARCHIVE command. The reason is that in the latter case we do not have the ids of the annotations to be archived, we only have the tuple t. Therefore, we use the ARCHIVE ANNOTATION command which takes care of finding the annotations attached to t and archive them. If the modification is a deletion of tuple t, then we archive both snapshot and views annotations on t.

When an annotation table AT is referenced in a query, we check whether or not the view annotations on AT need refreshing. The procedure for refreshing the view annotations on AT is presented in Figure 15. In Line 1, we get the view annotations attached to AT. For each view annotation r, we find the count of tables on which r depends and got modified after the r's last execution (Lines 3-4). If the count is zero, then we check whether or not r needs incremental evaluation (Lines 6-9). If the count is greater than zero, then r will be re-evaluated from scratch (Line 12). In

the incremental evaluation, we retrieve the last modification timestamp of relation T to which the annotation is attached (Line 6). If this timestamp is more recent than r's last execution timestamp, then we execute r only on the newly inserted or updated tuples in T. This is performed by modifying the stored ADD ANNOTATION command on the fly and then executing the modified command (Lines 8-9). After refreshing the annotation, we update r's last execution timestamp (Line 14).

In the procedure above, we make sure that all view annotations on AT are up-to-date before executing the new query Q. In this case, Q pays the whole refreshing cost even if it will not touch any of the newly inserted or modified tuples. We propose the following optimization to distribute the refreshing cost over multiple queries whenever possible.

***Distributing the Refreshing Overhead over Queries*:** The insight is that the number of delta tuples on which view annotations need to be refreshed can be large, and at the same time, the new query Q may touch few or even none of the delta tuples. Therefore, it is not fair that Q pays the whole refreshing cost. The idea is to estimate the most recently inserted or modified tuple that will be touched by Q and then refresh the view annotations up to this tuple only.

Assume that Q arrives at time t_Q and references user Table T and corresponding annotation tables AT_1, AT_2, ..., AT_m that are attached to T. >From the catalog tables, we find the list V_1, V_2,..., V_n of view annotations that need to be refreshed with last execution timestamps t_{v1}, t_{v2}, ..., t_{vn}, respectively, where $t_{v1} < t_{v2} < ... < t_{vn} < t_Q$. To estimate the most recently inserted or modified tuple in T that will be touched by Q we apply query Q' on T that contains all single predicates in Q over T and select the maximum *LastModifiedTime* of the qualified tuples, say $t_{touched}$. Since Q' is intended to be a fast estimation, it will not contain any complex predicates such as joins, grouping, or ordering, and hence Q' results in a conservative estimation.

View annotations on T are then refreshed up to Timestamp $t_{touched}$. For example, if $t_{touched} < t_{v1}$, then none of the view annotations need to be refreshed although they are not up-to-date. Similarly, if $t_{vk} < t_{touched} < t_{vk+1}$, then only view annotations V_1, V_2,..., V_k need to be refreshed up to $t_{touched}$. After (*partially*) refreshing those annotations, we update their *LastExecutionTimestamp* to $t_{touched}$.

6.2 Query Re-writing and Translation

An annotation qualifier $T_1[ANNOTATION(A_1,\ A_2)]$ indicates propagating the annotations from annotation tables A_1 and A_2 along with T_1. The system first checks that the annotation tables A_1 and A_2 are previously defined on T_1. Then, the ANNOTATION qualifier is re-written as a left join operation between the user table and the annotation table(s). The join predicate has the form:
$((T_1.Tuple\_OID,\ 0),\ (T_1.Tuple\_OID,\ MaxCol))$?#
$A_1.CoveredCells$, where ?# is the intersection operator between a line segment and a rectangle, and *MaxCol* is the number of columns in T_1. For each annotation table A_i in the FROM clause, a column with name A_i is automatically added to the projection list to hold the annotations from A_i.

A join annotation qualifier in the form $JoinAnnotation[(R, S), (T, W, Z)]$ indicates propagating the join annotations over the specified groups. For each group, a select statement is formed that retrieves the join annotations on that group. The tables in the group are then left joined with the formed select statement based on the equalities of the *Tuple_OIDs*. For each group, a column is added to the projection list to hold the join annotations on that group. For example, the qualifier above will add two columns *R_S_Annotation* and *T_W_Z_Annotation* to the projection list.

Annotations are typically propagated only on the projected attributes as well as the attributes specified by PROMOTE. In order to filter out annotations on the other attributes as early as possible, we translate the attributes in the projection list and PROMOTE to conditions that are added to the WHERE clause. Recall that a column in a table maps to a vertical line segment in the two-dimensional space. Moreover, columns with adjacent mapping ids are grouped together to form rectangles instead of line segments.

In the case where the query contains aggregation or grouping, i.e., group by, distinct, or set operators, more conditions are automatically added to the WHERE clause that restrict the propagated annotations to the ones having the *OnAggregationPropagateFlag* set to True. Users can also add conditions on the propagated annotations in the WHERE and HAVING clauses in the form $A_i.attr\_name$, where A_i is the annotation table name and *attr_name* is an attribute in the annotation table, e.g., curator, timestamp, etc.

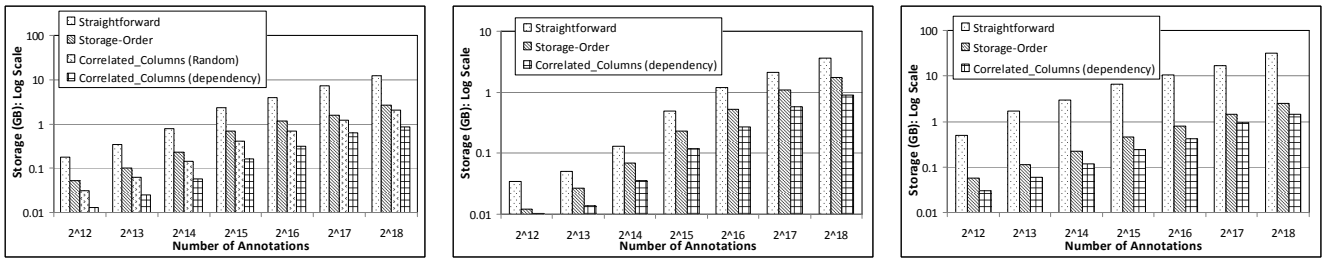
After the translation phase, the query becomes a standard SELECT query that is subject to the standard query optimization techniques such as pushing any annotation selection predicates to the annotation tables before joining them with the user tables.

7. EXPERIMENTS

Implementation: The annotation management functionalities are implemented inside PostgreSQL version 8.3. We extended the parser of PostgreSQL to process and translate the proposed commands. We implemented a graphical interface on top of our system using Excel sheets to visualize the annotations and allow users to add and query the annotations through GUI [8].

Datasets: We used real biological datasets from SWISSPROT database.We used two tables, *Gene* and *Protein*. Each table consists of around 200,000 tuples and occupies around 500MB on disk. Table *Gene* has six attributes while table *Protein* has eight attributes.

Annotation Generation: We designed a module that annotates the *Gene* and *Protein* tables. The three main factors of this module are (1) the number of annotations to generate, (2) the granularity of each annotation, i.e., how many rows and columns the annotation is attached to, and (3) the correlation among the annotated columns, i.e., which columns get annotated together more frequently. We varied the number of generated annotations over the values $2^{12}, 2^{13}, ..., 2^{18}$. The annotations are distributed equally over the *Gene* and *Protein* tables. For each annotation, the number of annotated rows, called *R-Values*, varies over the values $2^0, 2^2, 2^4, ..., 2^{12}$ and the number of annotated columns, called *C-Values*, varies from 1 to 6 in the case of table *Gene* and from 1 to 8 in the case of table *Protein*. The *R-Values* set is divided into two halves, the first half is called *R-Values-Small* and the second half is called *R-Values-Large* and the same is done for *C-Values*. These halves will be used to generate either fine-granular or coarse-granular annotations. The correlations among annotated columns are defined by



(a) Fine-granular 50%, Coarse-granular 50%

(b) Fine-granular 75%, Coarse-granular 25%

(c) Fine-granular 25%, Coarse-granular 75%

Figure 16: Storage overhead of the various storage schemes

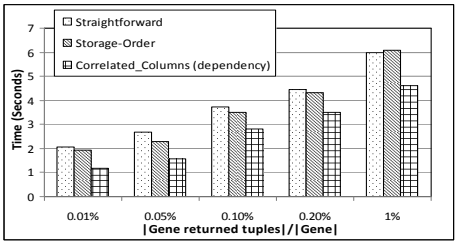
a set of rules, called *Correlated-Rules*. The rules have the form: *[If annotation X annotates column c_1 Then X annotates column c_2 with probability P and other columns with probability (1-P)]*.

Experimental Setup and Results: In the first set of experiments (Figure 16) we compare the storage overhead of three schemes (1) *Straightforward* scheme where annotations are stored on each cell independently (a cell is referenced by a Tuple_OID and column name), (2) *Storage-Order* scheme where annotations are stored in the proposed annotation tables without applying the correlated-columns algorithm, and (3) *Correlated-Columns* scheme where annotations are stored in the proposed annotation tables when applying the correlated-columns algorithm. Each of the *Gene* and *Protein* tables has one annotation table attached to it, namely *GeneAnnotation* and *ProteinAnnotation*, respectively.

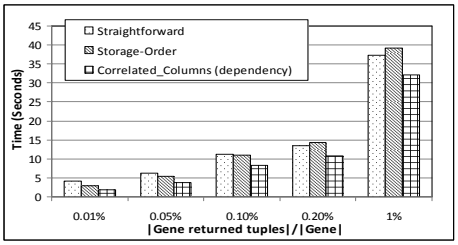
In Figure 16(a), the granularity of each annotation is uniformly distributed over *R-Values* and *C-Values*, that is, fine- and coarse-granular annotations have the same probability. We consider the case where *Correlated-Rules* set is empty, i.e., annotations over columns are totally random, labeled with *(Random)*, and the case where there are correlation rules, labeled with *(Dependency)*. The figure illustrates that the *Storage-Order* scheme achieves around 70% storage reduction compared to the *Straightforward* scheme because of the compact representation of the annotations. The figure shows that *Correlated-Columns* achieves more than an order-of-magnitude reduction in storage especially when there are correlations among the annotated columns.

In Figures 16(b) and 16(c), we study the cases where most annotations are fine-granular and coarse-granular, respectively. The results in Figure 16(b) are obtained by having the granularity of 75% of the annotations uniformly distributed over *R-Values-Small* and *C-Values-Small* while the granularity of the remaining 25% is uniformly distributed over *R-Values-Large* and *C-Values-Large*. The results in Figure 16(c) are obtained by switching the 75 and 25 percentages. The figures illustrate that *Correlated-Columns* scheme performs the best in both cases. It achieves around 60% storage saving in the case of fine-granular annotations and more than an order-of-magnitude storage saving in the case of coarse-granular annotations. The reason is that *Correlated-Columns* can reduce the storage even if annotations are attached to single tuples.

In the next set of experiments (Figures 17 and 18), we compare the I/O performance of the three schemes. The number of annotations is set to 2^{18} with a uniform distribution over *R-Values* and *C-Values*. In Figures 17, we query the data and propagate the annotations while in Figure 18, we query the data based on the annotation values. In Figure 17(a), we executed a set queries over the *Gene* table where the percentage of the returned tuples to the total tuples varies over 0.01%, 0.05%, 0.1%, 0.2%, and 1%. We run each query with varied number of projected attributes from 2 to 6. Our results show that the number of projected attributes does not have significant effect on the query performance, and hence we present the average execution time over the different sets of the projected attributes. The query plan involves a table scan on the *Gene* table followed by a left join with the *GeneAnnotation* table to annotate the returned tuples. The *Straightforward* scheme uses a B+-tree index on the Tuple_OID column in the annotation tables while the *Storage-Order* and *Correlated-Columns* use an R-tree index on the *AnnotationCoveredCells* column in the annotation tables. The figure illustrates that the I/O saving of the *Storage-Order* and *Correlated-Columns* schemes is not directly proportional to the storage saving. The reason is that the index performance is relative to the LOG of the data size which reduces the gap between the three schemes. In spite of that, the *Correlated-Columns* scheme achieves around 20% to 25% saving in the execution time compared

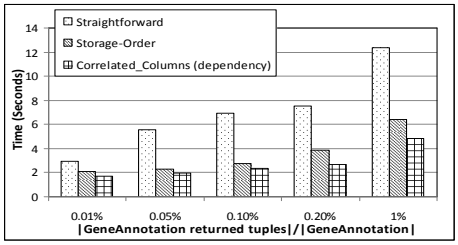


(a) Annotating a selection query on the Gene table

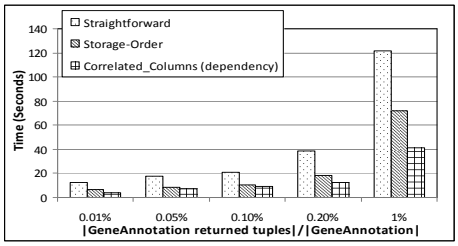


(b) Annotating a join query between the Gene and Protein tables

Figure 17: Execution time: Querying data and propagating annotations



(a) Querying the Gene table based on the annotations value



(b) Joining Gene and Protein tables based on the annotation vales

Figure 18: Execution time: Querying data based on annotation values

to the *Straightforward* scheme.

In Figure 17(b), we study the performance of joining the *Gene* and *Protein* tables and propagating the annotations on the returned tuples. We use the same set of queries as in Figure 17(a) with additional join condition between the two tables. The query plan involves a table scan on the *Gene* table followed by an inner join with the *Protein* table and then left joins with the *GeneAnnotation* and *ProteinAnnotation* tables to annotate the returned tuples. The figure illustrates that the *Correlated-Columns* scheme performs the best among the three schemes. The *Storage-Order* scheme has a better performance than the *Straightforward* scheme in the case of a relatively small query answer (0.05% is around 100 tuples). However, with a relatively larger query answer (0.1% is around 2000 tuples), the *Storage-Order* scheme shows a slower response time.

In Figure 18, we study the performance of queries that involve conditions of the annotation tables instead of the data tables. Figure 18(a) presents the execution time of queries that select genes from the *Gene* table that have certain annotations. The predicates on the *GeneAnnotation* table select the annotations whose values equal to a certain regular expression. The query plan involves a table scan on the *GeneAnnotation* table followed by a join with the *Gene* table to return the genes that satisfy the annotation conditions. The figure shows that *Correlated-Columns* achieves around 60% to 70% saving in the execution time. This is because the size of the annotation table in these experiments influence directly the number of I/O operations and hence the execution time.

Figure 18(b) presents the execution time of joining the *Gene* and *Protein* tables where the joined genes and proteins satisfy certain conditions on the annotation values. The query contains predicates on the *GeneAnnotation* and *ProteinAnnotation* tables. For consistency, we fix the number of proteins that satisfy the annotation conditions to 10 while varying the number of genes as indicated in the figure. The query plan involves table scans on the annotation tables followed by joins with the data tables to get the genes and proteins that have certain annotations, then a final join is performed the genes with the proteins. The figure illustrates the superiority of the *Correlated-Columns* scheme which is a direct result of the storage reduction.

8. CONCLUSION

In this paper, we presented mechanisms for supporting annotations on relational database management systems. We addressed storage optimization techniques, adding annotations and defining their behaviors, archiving, andă propagating annotations. We proposed several annotation types, e.g., *snapshot*, *view*, and *join* annotations, that give users the ability to specify the behavior of the annotations under different database operations. We addressed several storage and query processing challenges raised from supporting these types. We proposed a storage scheme, named *Mapped-Space*, for efficient storage of multi-granular annotations. The *Mapped-Space* scheme achieves more than an order-of-magnitude reduction in storage and up to 70% reduction in the queries execution time. We realized the proposed mechanisms through declarative extensions to the standard SQL inside PostgreSQL with an easy to use graphical interface.

9. REFERENCES

[1] Oracle life sciences platform, www.oracle.com/. *technology/industries/life_sciences/index.html*.

[2] D. Bhagwat, L. Chiticariu, W. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.

[3] D. Bhagwat, L. Chiticariu, W. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDB Journal*, 14(5), 2005.

[4] P. Buneman, J. Cheney, and W.-C. Tan. Curated databases. In *PODS*, pages 1–12, 2008.

[5] P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.

[6] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *SIGMOD*, pages 942–944, 2005.

[7] M. Eltabakh, M. Ouzzani, and W. Aref. bdbms: A database management system for biological data. In *CIDR*, pages 196–206, 2007.

[8] M. Eltabakh, M. Ouzzani, W. Aref, A. Elmagarmid, Y. Laura-Silva, M. Arshad, D. Salt, and I. Baxter. Managing biological data using bdbms. In *ICDE*, pages 1600–1603, 2008.

[9] F. GEERTS and J. V. D. BUSSCHE. Relational completeness of query languages for annotated databases. *DBPL*, 4797:127–137, 2007.

[10] F. Geerts, A. Kementsietsidis, and D. Milano. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE*, page 82, 2006.

[11] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[12] L. Haas, P. Schwarz, P. Kodali, E. Kotlar, J. Rice, and W. Swope. Discoverylink: A system for integrated access to life sciences data sources. *IBM System Journal*, 40(2):489–511, 2001.

[13] H. V. Jagadish and F. Olken. Database management for life sciences research. *SIGMOD Record*, 33(2):15–20, 2004.

[14] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.

[15] W. C. Tan. Containment of relational queries with annotation propagation. In *DBPL*, 2003.