2007

Supporting annotated Relations

M. Y. Eltabakh

M. Ouzzani

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Y. Laura-Silva

Report Number:
07-025

SUPPORTING ANNOTATED RELATIONS

M.Y. Eltabakh
M. Ouzzani
W.G. Aref
A.K. Elmagarmid
Y. Laura-Silva

Department of Computer Science
Purdue University
West Lafayette, IN 47907

CSD TR #07-025
November 2007

Supporting Annotated Relations

M.Y. Eltabakh, M. Ouzzani, W.G. Aref, A.K. Elmagarmid, Y. Laura-Silva
Computer Science Department, Purdue University
{meltabak, mourad, aref, ake, ylaurasi}@cs.purdue.edu

ABSTRACT

Annotations and provenance data play a key role in understanding and curating scientific databases. However, current database management systems lack adequate support for managing annotations and provenance data including: (1) handling annotations at multiple granularities, i.e., at the table, tuple, column and cell levels, (2) propagating annotations along with query answers, (3) querying data based on their annotations, and (4) providing declarative ways to add, archive, and restore annotations. In this paper, we propose to treat multi-granular annotations and provenance as first class objects inside the database. We introduce the concept of "Annotated Relations" along with new operators and extended semantics for the standard relational operators in support of annotated relations. We present an expressive and declarative extension to SQL to support the processing and querying of annotated tables. We study several schemes for storing and indexing annotations based on annotation granularity and annotation size. Extensions to PostgreSQL are introduced to support annotated relations and implementation challenges are discussed. Performance analysis illustrates the potential of annotated relations as they achieve up to an order-of-magnitude reduction in storage and I/O costs.

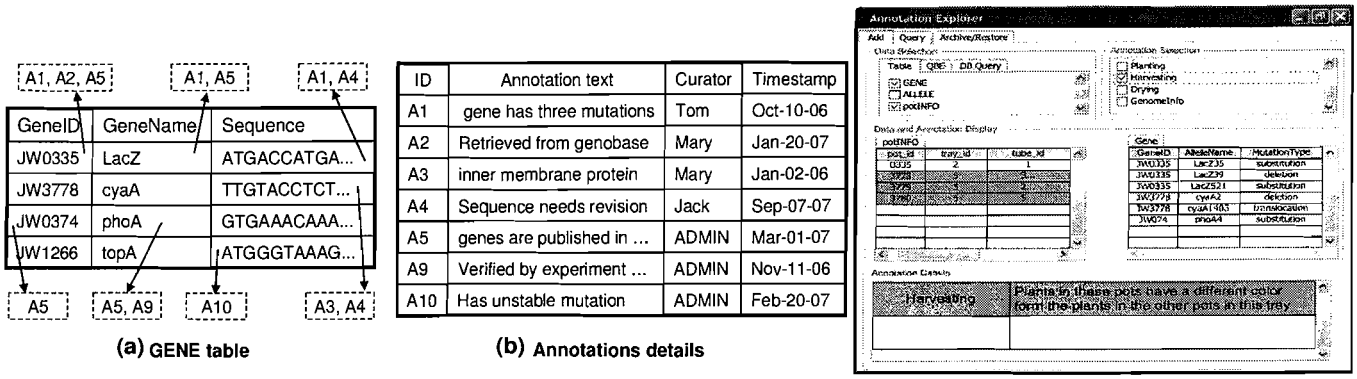
1. INTRODUCTION

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 and provenance data play a key role in understanding and curating scientific databases. Annotations allow users to better understand how a piece of data is obtained, why some values are being added or modified, and which experiment or analysis was performed to obtain a set of values. Moreover, provenance allows users to track the source of their data and to assess the credibility of the data based on its source.

In [10], we introduced *bdbms* as a database management system to support biological data and its emerging requirements. bdbms extends the functionalities of current database management systems to include: (1) annotation management, (2) tracking dependencies that involve external modules among data items, (3) authorizing database operations based on the content of the data, and (4) supporting novel and non-traditional access methods. [10] presented the overall system and challenges involved in each of the proposed functionalities. In this paper, we focus on the annotation management component. We study the extended SQL language, storage alternatives, implementation, and performance analysis.

Annotation management involves several challenges including: (1) ***Handling multi-granularity annotations.*** Annotations can be large in size and attached to the data at various granularities, e.g., at the level of cell, tuple, column, or table. Therefore, we need efficient storage schemes to avoid replicating the annotations. For example, annotations $A1$ (Figure 1) is attached to one tuple whereas annotation $A5$ is attached to four independent cells. The storage overhead becomes more critical in the context of 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, e.g., new querying capabilities, then users' queries may become complex and user-unfriendly. In addition to supporting the propagation of annotations with queries answers, the system needs also to support querying the data based on the annotation values. (3) ***Adding annotations in a declarative way.*** The goal is to annotate the data in an easy and declarative way, e.g., as if tables are visualized in grids and users are adding post-it notes. For example, how to select a group of cells to which the annotation will be attached. Without a declarative mechanism, adding annotations may not be an easy task.

In this paper, we propose to treat annotations as first class objects inside the database. We provide mechanisms for adding annotations at multiple granularities, i.e., at the table, tuple, column, and cell levels, archiving and restoring annotations, and querying the data based on the annotation values. We introduce the concept of "Annotated Relations" along with new operators and extended semantics for the standard relational operators to support annotations. We



(a) GENE table

(b) Annotations details

(c) Graphical User Interface

Figure 1: Annotating GENE table

present an extension to SQL to support the processing and querying of annotated relations. Since most scientists prefer to use graphical interfaces over using direct SQL commands, we need to provide an easy to use and intuitive ways to create and manipulate annotations. For this purpose, a graphical tool will be built on top of the proposed framework. Figure 1(c) shows an example of such a tool that would allow the end-user to browse the data and the annotations attached to it. The end-user can select the data of interest by highlighting specific table names and columns or writing a database query. The tool would then retrieve and display the data specified by the user and all the annotations attached to it. The tool also allows the user to filter, add, restore, or archive the annotations graphically.

The contributions of this paper are summarized as follows:

1. We propose to support annotations as first class objects in relational databases. We address several aspects in annotation management, e.g., adding, storing, archiving, restoring, and querying annotations.
2. We propose a declarative language based on SQL along with new operators and extended semantics for the standard operators to operate over the annotations.
3. We propose and study several storage schemes based on the granularity and size of annotations. Performance analysis illustrates that the proposed schemes can achieve up to an order-of-magnitude reduction in storage and I/O costs.

The rest of the paper is organized as follows. Section 2 overviews the related work. In Section 3, we present the proposed functionalities to manage annotations. The query re-writing and execution techniques are presented in Section 4. The performance analysis is presented in Section 5. Section 6 contains concluding remarks. The algebraic definition of the query operators is presented in Appendix A.

2. RELATED WORK

Managing annotations and provenance data is a key requirement in supporting scientific databases [12, 14, 15]. Commercial databases, e.g., Oracle and DB2, have added new features and functionalities inside the database engine to support life science applications [1, 13], e.g., accessing data stored in heterogeneous data sources via wrappers, integrating varieties of data types, and embedding/integrating data mining and analysis techniques inside the database engine. However, managing annotations has not been addressed by these systems.

Managing annotations in the context of relational databases has been addressed in previous works, e.g., [2, 5, 6, 11, 16]. An extension to SQL, termed *pSQL*, is proposed in [2, 6]. *pSQL* adds a PROPAGATE clause to SQL that allows users to specify how to propagate the annotations along with the query answers. The storage mechanism proposed in [2] simply assume that each cell in the database has a corresponding cell to hold the annotations. There are several key distinctions between our framework and the technique proposed in [2]: (i) our approach has a broader range of functionalities such as adding, archiving, and restoring the annotations, (ii) the querying capabilities proposed by our framework are more powerful where users can apply conditions to specify which annotations to propagate and also select the data based on the annotation values, (iii) we also study several storage optimizations to efficiently store the annotations. The technique in [2], however, can generate all queries that are equivalent to a given user query and propagate the annotations with those queries, which is not part of our framework.

Propagating annotations through views has been addressed in [5, 16]. MONDRIAN [11] proposes an algebra, termed *color algebra*, that extends annotating single values to annotating multiple related values with the same annotation. The *color algebra* also allows querying the data based on its annotation values. MONDRIAN, however, does not allow users to apply conditions to specify which annotations to propagate. MONDRIAN does not also address the issue of handling multi-granularity annotations. It does not address how to add or store annotations at various granularities. Although, a normalized storage is proposed by MONDRIAN to avoid repeating a tuple with each of its annotations, still an annotation that is attached to N tuples will be stored N times. Moreover, we present mechanisms to add and store annotations efficiently.

Extensive research has been conducted to track and compute

the provenance (lineage) of data. Provenance management techniques are divided into two main categories, annotation-based and inversion-based techniques. Annotation-based techniques, e.g., [2, 3, 6, 16], treat provenance data as a kind of annotations, i.e., provenance information are pre-computed and stored as annotations with the data. Inversion-based techniques, e.g., [4, 5, 8, 9, 17, 18], use derivation properties such as inverted functions and query operators properties, to derive the provenance of the data at run-time.

3. ANNOTATION MANAGMENT

We provides several functionalities to support the management of annotations. These functionalities include:

- **Modeling and storing annotations**
  Providing storage schemes for efficient manipulation and querying of annotations.
- **Adding annotations at multiple granularities**
  Allowing users to add annotations at multiple granularities in a declarative way.
- **Archiving and restoring annotations**
  Annotation archival is used to isolate outdated, invalid, or worthless annotations from recent and valuable annotations. Archived annotations will not be propagated to end-users along with the query answers. Restoring annotations is the inverse operation of archiving annotations.
- **Propagating annotations with query answers**
  Users have the flexibility of propagating or not the annotations and selecting which annotations to propagate along with the query answer.
- **Annotation-based querying**
  Users are able to query the data not only by specifying conditions on the actual data but also on the annotation values themselves.

Annotations usually contain important information that users want to retrieve and query. For example, annotations may refer to certain publications, reference other objects in the database, specify the experiments that triggered the annotation, or specify how confident the annotator is about the annotation, etc. While in some cases, users will be satisfied with having their annotations represented as simple string data, they may be interested in representing and organizing their annotations in a more flexible way. We support XML-formatted annotations where annotations are stored as XML documents inside the relational database. The use of XML has the advantages of allowing a better organization of the annotation content, and using the XML querying capabilities offered by the database to retrieve and query the annotations.

3.1 Annotation Data Model

We extend the concept of a *relation* to an *annotated relation*, i.e., a relation that has annotations attached to it. The annotations attached to a relation are organized and stored in one or more annotation tables. For example, in

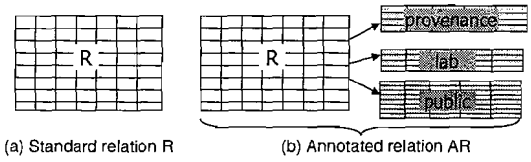


Figure 2: Annotated relations

```
CREATE ANNOTATION TABLE <ann_table_name>
ON <user_table_name>
SCHEME [Off-table|In-table]
```

Figure 3: Extended SQL command CREATE

Figure 2, a standard relation R is extended to an annotated relation AR that has three annotation tables *AR.lab*, *AR.public*, and *AR.provenance*. To create an annotation table over a given user relation, the command *CREATE ANNOTATION TABLE* (Figure 3) is used. The CREATE command creates an annotation table *ann_table_name* and inserts into the catalog table *annotation_catalog* a record that links the annotation table *ann_table_name* to the user table *user_table_name*. The SCHEME clause specifies how the annotations in *ann_table_name* are actually stored. The *Off-table* option means that the annotations will be stored in a separate table with name *ann_table_name*, whereas the *In-table* option means that the annotations will be stored in the user table in an additional attribute with name *ann_table_name*. This choice is made by the user based on the granularity and size of the annotations as we will explain in Section 3.2.

Having multiple annotation tables attached to a user relation has several advantages such as: (1) Different types of annotations can be stored separately, e.g., *AR.lab* may store the annotations from lab members, *AR.public* may store annotations from the public, and *AR.provenance* may store the provenance of R's data. (2) Different annotation tables can have different permissions and privileges. For example, adding annotations to *AR.lab* can be limited to lab members, adding annotations to *AR.public* can be open to the public, while adding annotations to *AR.provenance* can be limited to the integration tools.

3.2 Annotation Storage Schemes

Annotations can be attached to the data at multiple granularities, hence efficient storage and indexing schemes are needed. We propose three storage schemes based on the granularity of the annotations, namely *Off-table*, *In-table*, and *Hybrid* schemes. Each scheme has its pros and cons w.r.t. the storage and query processing overheads.

***Off-table Storage Scheme*:**
In this scheme, the SCHEME clause in the CREATE ANNOTATION TABLE command is always set to *Off-table*, i.e., the annotations over a given relation are stored in separate annotation tables. The *off-table* scheme is based on viewing a user relation as a two-dimensional space, e.g., columns represent the X-axis and tuple identifiers represent the Y-axis. Since annotations are associated with time, *Time* is the third dimension in the proposed scheme. Annotations can then be attached to multiple rectangles in this

3-D space. These rectangles may represent a cell, tuple, column, table, or any set of contiguous cells in the table. Rectangles are allowed to overlap. For example, annotations $A1$ and $A4$ (Figure 1(a)) each will be attached to a single rectangle that covers the annotated cells. Annotation $A5$, on the other hand, will be attached to two rectangles. The construction of the rectangles will be described in Section 3.3.

The structure of an annotation table is: *(TupleCol BOX, AnnotationBody XMLText)*, where *TupleCol* represents a set of cells (rectangle) to which the annotation is attached, and *AnnotationBody* is an XML formatted text that contains the annotation information. For a given annotation, the annotation table will have as many tuples as required rectangles to cover the annotated region. These tuples are inserted by the ADD ANNOTATION command as described in Section 3.3.

The advantage of the *Off-table* scheme is that it allows compact representation of annotations at various granularities. For example, an annotation over any group of contiguous cells is represented by a single record in the annotation table. Moreover, the *Off-table* scheme does not require any change in the structure of the user relations. The annotation tables can even be stored in a separate database. The only requirements that need to be maintained by the user database are: (1) each relation has a unique sequential identifier for each tuple, e.g., OID in PostgreSQL tables and (2) the structures of the user relations do not change. One disadvantage is that at query execution time, we need to join the data relation with the annotation relation(s) to form the annotated relation. This join operation can be I/O intensive as it joins each data tuple with all its annotation tuples.

In-table Storage Scheme:
In this scheme, the SCHEME clause in the CREATE ANNOTATION TABLE command is always set to *In-table*, i.e., the annotations over a given user relation, say R, are stored in R in additional annotation attributes created by the CREATE ANNOTATION TABLE command. No separate annotation tables are created.

To insert an annotation, we identify the data tuples to which the annotation will be attached (See Section 3.3). Then, a copy of the added annotation will be inserted into each of these tuples in the annotation attributes.

The advantage of the *In-table* scheme is that the user relation has all the annotations stored in it. Therefore, at query execution time, we directly select the desired data tuples along with their annotations. The disadvantage of the *In-table* scheme is in the storage overhead. The storage overhead can be very high because a single annotation can be replicated over many of the table's tuples. Moreover, having one huge table that stores the user's data as well as all the annotations over that table will result in performance degradation of that table's operations.

Hybrid Storage Scheme:
The *Hybrid* scheme combines the *off-table* and *In-table* schemes. It allows users to store some annotations in separate tables and other annotations in additional attributes in the users' tables. This decision is made based on the

```
ADD ANNOTATION
TO <annotation_table_names>
VALUE <annotation_body>
ON <select_statement> ;
```
(a)

```
ARCHIVE ANNOTATION
FROM <annotation_table_names>
[BETWEEN <time1> AND <time2>]
ON <select_statement> ;
```
(b)

```
RESTORE ANNOTATION
FROM <annotation_table_names>
[BETWEEN <time1> AND <time2>]
ON <select_statement> ;
```
(c)

Figure 4: Commands: ADD, ARCHIVE, RESTORE

```
Procedure CellGrouping
Inputs:
    List A of sorted attributes' identifiers (X-axis integer values)
    List T of sorted tuples' identifiers (Y-axis integer values)
Outputs:
    List of rectangles representing groups of contiguous points (cells)
Steps:
    1- Scan list A and identify the maximal intervals of contiguous values.
       Each interval X is defined by X.min and X.max
    2- Scan list T and identify the maximal intervals of contiguous values.
       Each interval Y is defined by Y.min and Y.max
    3- For each pair of X and Y construct rectangle ((X.min, Y.min) , (X.max, Y.max))
```

Figure 5: Constructing groups of contiguous cells

granularity and size of the annotations. Annotations can be categorized into two types based on their granularities, *fine-granularity* and *coarse-granularity* annotations. *Fine-granularity* annotations are attached to a specific cell, tuple, or few related cells or tuples in the user relation. For example, users' comments and annotations are usually *fine-granularity* annotations. *Coarse-granularity* annotations are attached to a large number of cells or tuples or even to entire tables or columns. For example, provenance is usually a *coarse-granularity* annotation. *Coarse-granularity* annotations are best stored in separate tables, i.e., the *off-table* scheme, while *Fine-granularity* annotations are best stored in the same user relation in additional annotation attributes, i.e., the *In-table* scheme. Currently, the user specifies either to store the annotations *In-table* or *Off-table*. In our future work, we plan to make this decision systematic, i.e., based on the size of the annotation and the number of tuples it is attached to, the system decides how to store the annotation.

The *Hybrid* scheme combines the advantages of both the *off-table* and *In-table* schemes. It allows compact representation of the *coarse-granularity* annotations and, at the same time, allows fast querying for the *fine-granularity* annotations.

3.3 Adding Annotations at Multiple Granularities

The new command *ADD ANNOTATION* (Figure 4(a)) is used to add annotations to the database. Users can also use the graphical tool (Figure 1(c)) to add the annotations, where the *ADD ANNOTATION* command will be automatically generated based on the user selection. The *annotation_body* parameter is an XML-formatted text that specifies the annotation value. The *annotation_table_names* parame-

ter specifies the annotation tables(s) in which the annotation will be stored. The *select_statement* parameter is a simple SELECT-FROM-WHERE SQL query that does not contain any aggregation or nested sub-queries. The projection list in *select_statement* is limited only to column names, i.e., no functions are allowed in the projection list. The annotation will be attached to the data in the base tables, the SQL query is only used as a powerful mechanism to specify the regions, i.e., groups of cells in the base tables, over which the annotation will be created. These regions can specify certain cells, entire tuples or columns, or even entire tables. For example, an entire table can be annotated using *SELECT * FROM TableName* query, while an entire column can be annotated using *SELECT ColumnName FROM TableName* query. The *ADD ANNOTATION* command allows users to also annotate multiple relations simultaneously.

The following ADD command illustrates adding annotation *A1* to gene *LacZ* (Figure 1(a)).

```
ADD ANNOTATION
TO Gene.Glab
VALUE '< Annotation >
          < Comment >
               gene has three mutations
          < /Comment >
       < /Annotation >'
ON (Select G.*
     From Gene G
     WHERE G.GeneID = JW0335);
```

The command stores the annotation in the annotation tables *Glab* that is linked to table GENE. The annotation will be attached to the entire tuple because all the attributes in the table are selected.

Notice that an annotation table, e.g., *Glab*, is allowed to appear in the *annotation_table_names* list, only if the corresponding user relation, e.g., GENE, appears in the *select_statement*.

The annotated region is in general a set of disjoint cells in the base table. However, contiguous cells within a region can be grouped together as one unit (rectangle). The *ADD ANNOTATION* command generates as many insert statements as the number of rectangles for each annotations. The procedure for translating the *ADD ANNOTATION* command to the corresponding insert statements (based on the *Off-table scheme*) is as follows:

1. Identify the user relations in the *select_statement* and their attributes in the projection list.

2. Map the attributes of each relation to integer values over the X-axis of a conceptual two-dimensional space. The mapping is based on the order of the attributes in the relation. For example, columns GeneID, GeneName, and Sequence in table GENE (Figure 1(a)) map to values 1, 2, and 3, respectively.

3. Execute the *select_statement* to retrieve the identifiers of tuples to which the annotation will be attached. These identifiers are mapped to integer values over the Y-axis in the two-dimensional space. The identifiers can be the system auto-generated identifiers for each tuple in the relation. Consecutive tuples should have consecutive identifiers.

4. Group the points in the two-dimensional space to form rectangles, i.e., a group of contiguous cells form one rectangle. The algorithm for constructing these rectangles is illustrated in Figure 5.

5. For each identified rectangle, form and execute an insert statement on the corresponding annotation table.

The insertion procedure in the case of the *In-table* scheme is more straightforward. The select statement is first executed to determine the output tuples, then the annotation is inserted into the annotation attributes specified in the TO clause of those tuples.

For each inserted annotation, the system adds automatically the following XML elements $< User >, < Timestamp >$, $< ArchiveFlag >$, and $< Columns >$, where element *User* contains the name of the user inserting the annotation, *Timestamp* contains the annotation insertion time, *ArchiveFlag* indicates whether the annotation is archived or not, initially set to FALSE, and *Columns* lists the columns to which the annotation is attached. These elements are accessible to end-users, e.g., users can query the annotations based on the *User* or *Timestamp* values.

If using the graphical tool to add the annotations, the tool would track the column names and tuple identifiers of the displayed data. Then, when the users selects or highlights certain cells to annotate, these cells will be mapped to the corresponding column names and tuple identifiers. Then, the tool generates an *ADD ANNOTATION* command, where the *select_statement* parameter will select the specified column names and tuple identifiers. The command will then be passed to the database engine to be executed according to the procedure above, except that Step 3 will be skipped since the tuple identifiers are already part of the *select_statement*.

3.4 Archiving and Restoring Annotations

Archival of annotations allows users to isolate outdated, invalid, or worthless annotations from recent and valuable ones. The archival operation changes the status of an annotation from active to inactive. Inactive annotations will not be propagated with the query results. Archival is useful in many scenarios, for example, when the data associated to the annotation is modified or when the the lab administrator decides that certain annotations are no longer correct or relevant.

We propose the *ARCHIVE ANNOTATION* command (Figure 4(b)) to archive annotations. The annotations will be archived only from the annotation tables specified in *annotation_table_names*. The cells over which the annotations will be archived are specified through the *select_statement* parameter. If the optional clause *BETWEEN* is specified, only the annotations created between *time1* and *time2* will be archived.

Restoring annotations is the inverse operation to archiving annotations. We propose the *RESTORE ANNOTATION* command (Figure 4(c)) to restore archived annotations. *RESTORE ANNOTATION* has the same clauses as *ARCHIVE ANNOTATION*. Restored annotations will be propagated again to end-users along with query answers.

The procedures for executing the *ARCHIVE/RESTORE ANNOTATION* commands are similar to that of *ADD ANNOTATION*, where the *select_statement* is first executed to identify the query results, and then the annotations attached to the identified cells are marked as inactive or active.

In case the users' data gets updated, the annotations attached to the updated cells may or may not remain valid based on the annotations' semantics. Moreover, after a sequence of updates, it may become confusing which annotations are attached to which data values. In the proposed mechanis, we do not automatically archive the existing annotations when there is an update on the data since they may still be valid for the new values. Instead, we provide a mechanism that allows users to track which annotations are added to which values. When a cell that already has an annotation added to it gets updated, the system automatically generates an annotation specifying that the cell value has been updated by user *user_name* at timestamp *update_time* with old value *old_value*. These annotations will be stored in a separate annotation table (maintained by the system), and will be propagated automatically to end-users whenever a query involves annotation propagation. As a result, users can track which annotations are added to which values. If users want to actually archive certain undesired annotations, they can manually issue an ARCHIVE command. This approach works well for databases that do not involve frequent updates. In our future work, we plan to study other alternatives and techniques that suits databases with frequent updates.

3.5 Data- and Annotation-based Querying

To propagate annotations along with query answers or to query data based on annotation values, we introduce an extended SELECT statement (Figure 6) that operates over annotated relations. We extended the semanitcs of the standard operators and introduced the new operators PROMOTE, ANNOTATION, AWHERE, AHAVING, and FILTER that allow users to perform operations and apply conditions over the annotations.

The main motivation behind extending the semantics of existing operators and introducing new operators is that annotations are metadata and should receive special processing in the query pipeline, otherwise users' queries will become very complex. For example, assume identical tuples $T1$, $T2$, ..., Tn with different annotations in the query pipeline. Existing operators that group tuples, e.g., DISTINCT, UNION, GROUP BY, INTERSECT, will produce all $T1$, $T2$, ..., Tn tuples because they cannot detect that these tuples are identical. Thus many duplicates are produced in the output and these operators will lose their functionalities. The other way of processing is to apply these operators before attaching the annotations to the tuples. In this case, the operators will produce one copy of the identical tuples. However, the other identical tuples will be lost and tracking them to retrieve

```
SELECT [DISTINCT] Ci [PROMOTE (Cj, Ck, ...)], ...
FROM  Relation_name [ANNOTATION(S1, S2, ...)], ...
[WHERE <data_conditions>]
[AWHERE <annotation_condition>]
[GROUP BY <data_columns>
  [HAVING <data_condition>]
  [AHAVING <annotation_condition>] ]
[FILTER <filter_annotation_condition>]
[ORDER BY <data_columns>]

[UNION | UNION ALL | INTERSECT | EXCEPT]
```

Figure 6: The Extended SELECT

their annotations is very complex. Another scenario that motivates the need for new operators is the following query: retrieve each tuple T along with all its annotations only if T has an annotation inserted by user "Admin". The standard join operator will join T with its annotation tuples. Adding the condition User = "Admin" will result in generating T only with the annotations inserted by Admin. The other annotations will be dropped. To retrieve the other annotations on T, more join operations are needed. In contrast, the proposed extended query operators allow the database engine to natively and efficiently support the processing and retrieval of annotations.

The semantics of the new clauses are as follows. PROMOTE, which may follow a column's name in the projection list, copies annotations from some columns, possibly not in the projection list, to a specific column. As a result, annotations over non-projected attributes can be kept and propagated in the query pipeline. ANNOTATION, which may follow a relation's name in the FROM clause, specifies which annotation tables to consider in the query, i.e., which annotations to propagate. For example, a user may want to only propagate the annotations stored in annotation table *AR.Provenance*. The selection operator AWHERE selects tuples based on conditions applied to the tuples' annotations, i.e., if the annotations attached to the tuple satisfy *annotation_condition*, then the tuple is selected along with its annotations. AHAVING is analogous to AWHERE except that the former is applied after the grouping is performed. Unlike AWHERE and AHAVING which may drop entire user tuples, the FILTER operator passes all tuples of the input relation after filtering the annotations attached to each tuple based on the filtering condition(s). For example, users may want to propagate only the annotations added in the last month with each tuple. In this case, FILTER will filter out any annotation with a timestamp before the last month.

Since annotations are XML documents, then the new operators, AWHERE, AHAVING, FILTER, and PROMOTE operate on XML data. The AWHERE, AHAVING, and FILTER operators apply XPath boolean expressions over a given annotation document. The PROMOTE operator concatenates two annotation documents into one document.

Example: Consider the following query $Q1$:

Q1: SELECT GeneID, GeneName [PROMOTE(Sequence)]
FROM GENE[ANNOTATION(Glab)]

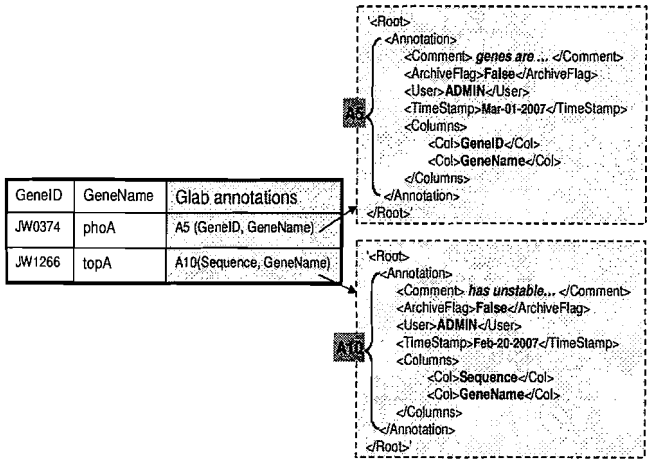


Figure 7: Results from query Q1

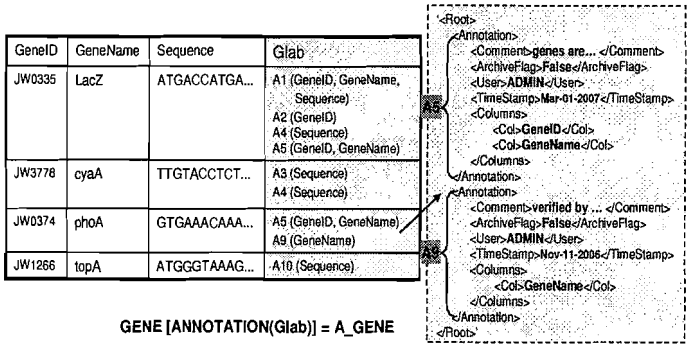


Figure 8: Annotated relation in the query pipeline (Single annotation document attached to each tuple)

WHERE GeneID IN ('JW3778', 'JW0374', 'JW1266')
AWHERE xpath_bool(Glab,'Root/Annotation/User ="ADMIN"')
FILTER xpath_bool(Glab,'Root/Annotation/TimeStamp >"Jan-01-2007"');

The query selects the GeneID and GeneName from table GENE (Figure 1(a)) where (1) GeneID equals 'JW3778', 'JW0374', or 'JW1266', and (2) the gene entry has annotations inserted by user 'ADMIN' in Glab. For each output tuple, report only the annotations inserted after 'Jan-01-2007' and copy the annotations on column Sequence to column GeneName.

The output of $Q1$ is given in Figure 7. The WHERE clause passes only the three tuples corresponding to genes JW3778, JW0374, and JW1266. the AWHERE clause passes only the two tuples corresponding to genes JW0374 and JW1266 since they have annotations entered by 'ADMIN'. The FILTER clause drops annotation $A9$ from tuple JW0374 because $A9$ is entered before 'Jan-01-2007'. The PROMOTE clause copies annotation $A10$ from column Sequence to column GeneName. Finally, the projection selects columns GeneID and GeneName along with their annotations $A5$ and $A10$.

4. QUERY REWRITE AND EXECUTION

A user relation R is transformed to an annotated relation AR by the ANNOTATION operator in the FROM clause. The structure of AR in the query pipeline is independent of the underlying annotations' storage scheme. This separation between the underlying annotations' storage scheme and the query processor is achieved through the ANNOTATION operator, i.e., the implementation of the ANNOTATION operator varies based on the underlying annotations' scheme to generate the same structure in the query pipeline.

The general structure of AR in the query pipeline is: $(\{user\_attributes\}, \{annotation\_attributes\})$, where *user_attributes* is a set of user attributes and *annotation_attributes* is a set of attributes that hold the annotations over each tuple of AR. *annotation_attributes* are special attributes and hence receive special processing in the query pipeline. The various query operators identify the annotation attributes by their data type, i.e., *XMLText*. *XMLText* is a new data type that we added to PostgreSQL to store the annotations.

In AR, an annotation attribute is added for each *Off-table* annotation table specified in the ANNOTATION clause. For example, $GENE[ANNOTATION(Glab)]$ produces table A_GENE with an additional annotation attributes *Glab* (See Figure 8) that holds the annotations from Glab annotation table. A_GENE table corresponds to the conceptual structure given in Figure 1(a). The annotations over each tuple are concatenated and stored in a single XML document as illustrated in the figure. For example, the tuple corresponding to gene *JW0374* has two annotations attached to it, $A5$ that is attached to columns GeneID and GeneName and $A9$ that is attached to column GeneName.

The semantics of the standard query operators are modified to support the processing of *annotation_attributes* as follows (the algebraic definition of the operators is presented in Appendix A):

DISTINCT, UNION, INTERSECT, and EXCEPT: These operators do not take *annotation_attributes* into account while comparing the columns to identify the identical tuples. That is, tuples $T1$ and $T2$ are considered identicals if the *user_attributes* are identicals. A generated tuple T will have the same *annotation_attributes* as the input tuples. Each annotation attribute A is the union of A's annotations over T's identical copies.

Projection: Projects the *annotation_attributes* in addition to the *user_attributes* in the projection list. The annotations in each annotation attribute are filtered to drop off the annotations over the non-projected user attributes.

Cartesian product: Cartesian product joins two annotated relations AR and AS and produces another annotated relation AT with attributes $(\{AR\_attributes\}, \{AS\_attributes\})$. Hence, the number of annotation attributes in AT is the sum of the number of the annotation attributes in AR and AS.

The new clauses ANNOTATION, AWHERE, AHAVING, FILTER, and PROMOTE, are re-written using the existing operators as well as database stored functions according to the following rules:

ANNOTATION: The re-writing rule for clause $R[ANNOTATION(A1, A2, ..., An)]$ depends on which storage scheme is used.

– **In-table:** *Project from R the annotation attributes A1, A2, ..., An.* Initially, R has all the annotation attributes.

– **Off-table:** *Left join R with the annotation tables A1, A2, ..., An. Project the user attributes from R and the AnnotationBody column from each of the annotation tables. Group the tuples based on the user attributes of R and apply a union aggregation operator over the annotation attributes.* Notice that each annotation table *Ai* adds an annotation attribute with name *Ai* to the projection list.

– **Hybrid:** *Apply the In-table rule, Off-table rule, or both as required.*

AWHERE: *Move the AWHERE conditions to the WHERE clause (create a new WHERE clause if needed).* The AWHERE conditions are applied over the *annotation attributes.* Since the *annotation attributes* are materialized using the ANNOTATION operator, then AWHERE conditions can be added as regular WHERE conditions.

AHAVING: *Move the AHAVING conditions to the HAVING clause (create a new HAVING clause if needed).* AHAVING is analogous to AWHERE.

FILTER: *FILTER is implemented as a database scalar function FILTER_ANNOTATION() that is applied to the annotation attributes in the projection list.* FILTER_ANNOTATION() takes an annotation attribute and *filter_annotation_conditions* as arguments. Each annotation that does not satisfy *filter_annotation_conditions* is removed.

PROMOTE: *PROMOTE is implemented as a database scalar function PROMOTE_ANNOTATION() that is applied to the annotation attributes in the projection list.* PROMOTE_ANNOTATION() takes an annotation attribute and a list of source and destination user attributes as arguments. An annotation over any of the source attributes is copied to the destination attribute.

Example: Consider re-writing query $Q1$ in the case of the *In-table* storage scheme:

```
SELECT GeneID, GeneName, PROMOTE_ANNOTATION(
  FILTER_ANNOTATION(Glab, 'Root/Annotation/TimeStamp
  >"Jan-01-2007"'), 'Sequence', 'GeneName') AS Glab
FROM GENE
WHERE GeneID IN ('JW3778', 'JW0374', 'JW1266')
AND xpath_bool(Glab,'Root/Annotation/User ="ADMIN"');
```

In the case of an *Off-table* storage scheme, the re-writing is the same except in the FROM clause, which will be:

```
FROM (SELECT GeneID, GeneName, Sequence,
          AnnConcat(AnnotationBody) AS Glab
    FROM GENE Left Join Glab
    ON ((GENE.OID,0), (GENE.OID,MaxCol)) @@ Glab.TupleCol
    GROUP BY GeneID, GeneName, Sequence) AS GENE
```

Parameter	Definition
C	Number of coarse-granularity annotations
Z_C	Average size of a coarse-granularity annotation
N_C	Average number of tuples of a single coarse-granularity annotation (In-table scheme)
N'_C	Average number of tuples of a single coarse-granularity annotation (Off-table scheme), $N'_C \leq N_C$
F	Number of fine-granularity annotations
Z_F	Average size of a fine-granularity annotation
N_F	Average number of tuples of a single fine-granularity annotation (In-table scheme)
N'_F	Average number of tuples of a single fine-granularity annotation (Off-table scheme), $N'_F \leq N_F$
M	Memory block size
R	User relation

Figure 9: Storage analysis parameters

Variable	Value	Definition
$B(C_{oft})$	$(Z_C{*}C{*}N'_C)/M$	Number of blocks of the coarse-granularity annotations (Off-table scheme)
$B(C_{in})$	$(Z_C{*}C{*}N_C)/M$	Number of blocks of the coarse-granularity annotations (In-table scheme)
$B(F_{oft})$	$(Z_F{*}F{*}N'_F)/M$	Number of blocks of the fine-granularity annotations (Off-table scheme)
$B(F_{in})$	$(Z_F{*}F{*}N_F)/M$	Number of blocks of the fine-granularity annotations (In-table scheme)

Figure 10: Storage analysis

We left-join table GENE with annotation table Glab such that each data tuple (represented by the line segment *((GENE.OID,0), (GENE.OID,MaxCol))* in the two dimensional space) joins with all its annotations (represented by the rectangles in Glab.TupleCol) where @@ is the intersection operator between a line segment and a rectangle. Then we concatenate all the annotations of each tuple using the aggregate function *AnnConcat.*

5. PERFORMANCE ANALYSIS

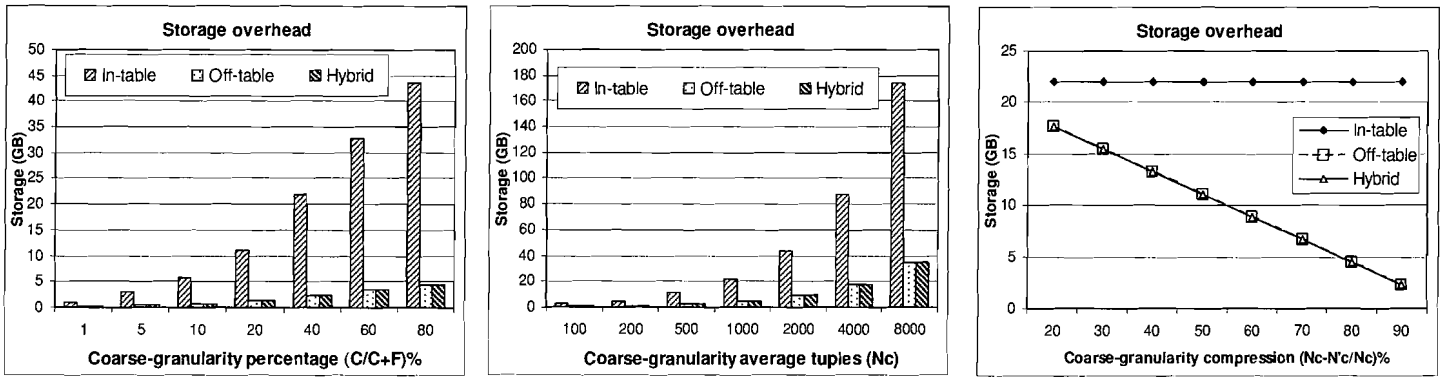
We focus our performance analysis on studying and comparing the storage overhead of the three storage schemes and the I/O cost introduced by the new annotation-specific operators. The set of parameters that affect our analysis are listed in 9. Parameters C and F represent the number of coarse- and fine-granularity annotations, respectively, i.e., $C + F$ is the total number of annotations in the database. The average size of a single coarse- and fine-granularity annotation is represented by Z_C and Z_F, respectively. Although we set Z_C and Z_F to the same value in the experiments, we use two different parameters to provide a more accurate analytical analysis. Another important parameter is the average number of tuples associated with a single annotation, which is represented by N_C and N_F for a coarse- and fine-granularity annotation, respectively. The last parameter that affects the comparison among the various storage schemes is how much an annotation can be compressed in the *Off-table* scheme, i.e., the average number of rectangles associated with a single annotation. This parameter is represented by N'_C and N'_F for a coarse- and fine-granularity annotation, respectively. For example, $N_C - N'_C/N_C$ represents the average compression of a single coarse-granularity annotation.

In our analysis, instead of randomly generating annotations and then measuring the storage and I/O costs, we provide

Scheme	ANNOTATION operator maps to	Size of input relation(s)	Algorithm	I/O cost
In-table	Scan	$B(R) + B(C_{in}) + B(F_{in})$	Table scan	$B(R) + B(C_{in}) + B(F_{in})$
Off-table (R join C_{off})	Join	$B(R)$, $B(C_{off})$	Block-based nested loop join	$B(R) + B(C_{off})$
			Nested loop with index on C_{off}	$B(R) + [T(R) * C*N_C/T(R)] = B(R) + C*N_C$
* *For (R join F_{off}), replace C with F*	Group By	$[B(R) * C*N_C/T(R)] + B(C_{in})$	In-memory processing (Input relation in order)	0
			Two-pass algorithm (input relation not ordered)	$3 * ([B(R) * C*N_C/T(R)] + B(C_{in}))$
Hybrid (F_{in})	Scan	$B(R) + B(F_{in})$	Table scan	$B(R) + B(F_{in})$
Hybrid (R join C_{off})	Join	$B(R) + B(F_{in})$, $B(C_{off})$	Block-based nested loop join	$B(R) + B(F_{in}) + B(C_{off})$
			Nested loop with index on C_{off}	$(B(R)+B(F_{in})) + [T(R) * C*N_C/T(R)] = B(R) + B(F_{in}) + C*N_C$
	Group By	$[(B(R)+B(F_{in})) * C*N_C/T(R)] + B(C_{in})$	In-memory processing (Input relation in order)	0
			Two-pass algorithm (input relation not ordered)	$3 * ([(B(R)+B(F_{in})) * C*N_C/T(R)] + B(C_{in}))$

* T(R): Number of tuples in R

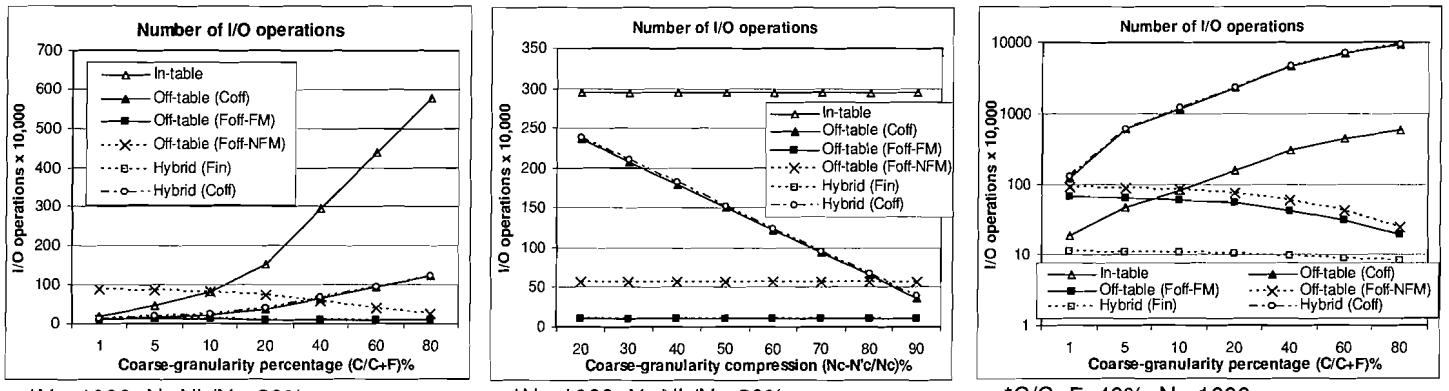
Figure 11: I/O cost analysis



*N_c=1000, N_c-N'_c/N_c=80% *C/C+F=40%, N_c-N'_c/N_c=80% *C/C+F=40%, N_c=1000

(a) (b) (c)

Figure 12: Storage overhead of various schemes



*N_c=1000, N_c-N'_c/N_c=80% *N_c=1000, N_c-N'_c/N_c=80% *C/C+F=40%, N_c=1000

(a) I/O cost: Block-based joins (b) Compression of coarse-granularity annotations (c) I/O cost: Index-based joins

Figure 13: I/O cost of the ANNOTATION operator

an analytical model for the storage and I/O costs, and then vary the values of the analysis parameters to illustrate their effect on the performance. We present the analytical models in Figures 10 and 11. Figure 10 gives the storage requirements (in memory blocks) of the coarse- and fine-granularity annotations when using the *In-table* and *Off-table* schemes. The overall storage overhead of the *In-table*, *Off-table*, and *Hybrid* schemes is then computed as: $B(C_{in}) + B(F_{in})$, $B(C_{off}) + B(F_{off})$, and $B(C_{off}) + B(F_{in})$, respectively. Notice that for the *Hybrid* scheme, the coarse-granularity annotations are stored *Off-table* while the fine-granularity annotations are stored *In-table*.

In the I/O cost analysis, the ANNOTATION operator is the only operator that involves I/O operations. The AWHERE, AHAVING, FILTER, and PROMOTE operators do not involve additional I/O operations as they operate on a tuple-by-tuple basis as tuples pass in the pipeline. Figure 11 illustrates the I/O cost model of the ANNOTATION operator under the three storage schemes along with other conditions including the use of indexes and whether the annotation table fits into memory or not.

In the *In-table* scheme, the annotations, i.e., C_{in} and F_{in}, are stored in the user relation. Therefore, the size of the user relation is the sum of $B(R)$, $B(C_{in})$, and $B(F_{in})$. In this scheme, the ANNOTATION operator involves only a scan over the user table (I/O cost $= B(R) + B(C_{in}) + B(F_{in})$).

In the *Off-table* scheme, we join the user relation R with the annotation table, i.e., C_{off} or F_{off}, to retrieve the annotations. We consider two methods for performing the join operation: (1) block-based nested loops join, and (2) nested loops join with an index on the annotation table. In the latter method, we build an R-tree index over the rectangles to which the annotations are attached. The I/O costs of these two methods are given in Figure 11. Following the join operation, a group-by operation is performed to group each user tuple with all its annotations into a single tuple. The I/O cost of the group-by operation depends on whether the input tuples are ordered. For example, if the join is performed using an index over the annotation table, then each user tuple will join with all its annotations at once. Then, the group-by operator will group these tuples into one tuple using main memory processing, i.e., no I/O operations are involved. The same case applies if the join is performed using block-based nested loops join where the annotation table can fit entirely into memory. If the annotation table is large and cannot fit into memory, then input to the group-by operator is not ordered and a two-pass algorithm is needed to sort and group the input tuples, which will require I/O operations as in Figure 11.

In the *Hybrid* scheme, the fine-granularity annotations are stored in the user table, i.e., F_{in}. Therefore, the size of the user table is the sum of $B(R)$ and $B(F_{in})$. In this scheme, retrieving the fine-granularity annotations involves only a scan operation over the user table with I/O cost $B(R) + B(F_{in})$. Whereas retrieving the coarse-granularity annotations (i.e., C_{off}) involves join and group-by operations. The cost of these two operations is similar to that of the *Off-table* scheme with the difference of having the user table of size $B(R) + B(F_{in})$ instead of $B(R)$ (See Figure 11).

We study the effect of various parameters on the storage and I/O requirements as follows. Our database consists of a Swiss-Prot [7] protein table that stores 200,000 tuples and occupies 600MB of disk space approximately. We define three annotation datasets D1, D2, and D3, that correspond to annotating 25%, 50%, and 100% of the protein tuples, respectively. That is, the number of annotations $(C + F)$ in D1, D2, and D3 is 50K, 100K, and 200K annotations, respectively. In each dataset, we vary the following parameters: (1) the percentage of the coarse-granularity annotations to the total annotations, i.e., $C/(C + F)$, (2) the average number of tuples associated with a single coarse-granularity annotation, i.e., N_C, and (3) the compression of the coarse-granularity annotations, i.e., $N_C - N'_C/N_C$. The variation in the remaining parameters is usually very small, therefore we fixed their values. The size of an annotation (Z_C or Z_F) is usually few hundred bytes, so we fix it to 500 bytes. Also, according to our definition, a fine-granularity annotations is attached to very few tuples, and hence we fix this value (N_F) to 5. Since N_F is very small, then the probability that two tuples be adjacent and merged togather in one rectangle is very low. Hence, we set N'_F to the same value as N_F, i.e., no compression occurs for the fine-granularity annotations. The result that will be presented in the following figures is the average over the three datasets D1, D2, and D3.

In Figure 12, we present the effect of varying the parameters' values on the storage overhead. The figure illustrates that the *In-table* scheme involves the highest storage overhead due to storing each coarse-granularity annotation with every tuple it is attached to. The overheads of the *Off-table* and *Hybrid* schemes are the same because storing the fine-granularity annotation in the user table or in a separate table does not make a difference in this case. Notice that changing the compression percentage of the coarse-granularity annotation $N_C - N'_C/N_C$ does not affect the *In-table* scheme because the coarse-granularity annotations are stored in the user table.

In Figure 13(a), we illustrate the I/O cost of the various schemes where join operations are performed using the block-based nested loops join. We consider two cases for the off-table fine-granularity annotations F_{off} since their number and size can be large: (1) F_{off}*-FM*: F_{off} can fit entirely into memory, and (2) F_{off}*-NFM*: F_{off} cannot fit entirely into memory. For example, the curve labeled with $F_{off}-$FM corresponds to the I/O cost of retrieving the fine-granularity annotations from a separate annotation table that can fit into memory. The figure illustrates that if the annotation table can fit entirely into memory, then the *Off-table* scheme involves the least I/O cost and the *Hybrid* scheme involves a slightly higher cost (See C_{off} and F_{off}*-FM* in Figure 13(a)). If the annotation table cannot fit into memory, then the I/O cost of the *Off-table* scheme increases significantly because the group-by operator will perform an I/O-intensive two-pass algorithm to sort and group the tuples (See F_{off}*-NFM* in Figure 13(a)). In this case, the *Hybrid* scheme outperforms the *Off-table* scheme because the fine-granularity annotations are stored in the user relation. The figure illustrates also that the I/O cost of the *In-table* scheme is very high and that the cost increases dramatically with the increase in C. The reason is that the size of the user relation is very large (See Figure 12), and hence a scan over the table

is I/O-intensive.

In Figure 13(b), we vary the compression percentage of the coarse-granularity annotations by varying N'_C. The change in the compression percentage affects only the retrieval of C_{off} in the *Off-table* and *Hybrid* schemes. The figure illustrates that even with low compression percentage, e.g., 30% or 40%, the *Off-table* and *Hybrid* schemes still outperform the *In-table* scheme.

In Figure 13(c), we study the use of an index, namely an R-tree index on the annotation table, in the join operations. The advantage of the index is that independently from whether the annotation table fits into memory, the group-by operator can group the input tuples using in-memory processing, i.e., without performing I/O operations. Despite this advantage, performing the join operations using the index is I/O intensive and dominates the I/O saving of the group-by operator. The cost of the *Off-table* scheme and the *Hybrid* scheme over C_{off} increase dramatically when the join operation uses the R-tree index. The reason is that, in this case, the R-tree index is a non-clustered index and hence the I/O cost becomes a function of the number of joined tuples instead of the number of blocks of the joined relations. Figure 13(b) indicates that building an index over the annotation tables is worthless and that the use of a block-based nested loops join performs much better.

In conclusion, our performance analysis illustrates clearly that the *Off-table* and *Hybrid* schemes achieve significant storage saving over the *In-table* scheme due to the compression of the coarse-granularity annotations. With respect to the I/O cost, the *Off-table* scheme performs the best if the annotation tables can fit into memory. Otherwise, the *Hybrid* scheme performs the best.

6. CONCLUSION

We presented a framework for supporting annotations as first-class objects in the database. We provided a declarative and expressive mechanism to operate over the annotations, e.g., adding, archiving, restoring, and propagating annotations, as well as querying the data based on the annotation values. We presented an extension to SQL, along with new operators and extended semantics for the standard relational operators to support the processing and querying of annotations. We introduced three annotation storage schemes to accommodate different granularities and sizes of the annotations. The performance analysis illustrates that effective compression of coarse-granularity annotations leads to a significant saving in the storage requirements and in the I/O cost of the queries.

APPENDIX

A. EXTENDED QUERY ALGEBRA

We provide the algebraic definitions of the extended query operators over annotated relations.

Standard relation (R): $R = \{r_i =< C_1, C_2, ..., C_m >\}$ is defined as a set of tuples r_i with attributes C_1, C_2, ..., C_m.

Annotated relation (AR): $AR = \{ar_i =< (C_1, AC_1), (C_2, AC_2), ..., (C_m, AC_m) >\}$ is the annotated version of R. AR is defined as a set of tuples ar_i with attributes C_1, C_2, ..., C_m. Each attribute C_i has a list of annotations AC_i attached to it. This definition is conceptual, i.e., annotations at various granularities are broken down into the cell level and attached to each cell in the table.

ANNOTATION: The ANNOTATION operator $\Upsilon(R, S_1, S_2, ..., S_n)$ specifies which annotations to propagate with a relation R from the set of annotation tables that have been created for that relation R. The output of the ANNOTATION operator is the annotated relation $AR = \{ar_i =< (C_1, AC_1), (C_2, AC_2), ..., (C_m, AC_m) >\}$, where AC_i is the list of annotations over C_i in $S_1, S_2, ..., S_n$.

Projection (Figure 14(a)): the projection $\pi'_{C_1, C_2, ..., C_x}(AR)$ over annotated relation AR selects a set of attributes C_1, C_2, ..., C_x, along with their annotations AC_1, AC_2, ..., AC_x. Other attributes and annotations are dropped.

Annotation copying (PROMOTE) (Figure 14(b)): copies the annotations of some attributes of AR, e.g., C_2, C_3, ..., C_x, to a another attribute of AR, e.g., C_1.

$$\beta(AR, C_1, \{C_2, C_3, ..., C_x\}) = \{ar_i =< (C_1, AC_1 + AC_2 + AC_3 + ... + AC_x), (C_2, AC_2), (C_3, AC_3)..., (C_x, AC_x), ..., (C_m, AC_m) >\}$$

where '+' is the annotation union operator.

Data-based selection (WHERE, HAVING) (Figure 14(c)): selects tuples from AR based on conditions P applied over attributes $C_1, C_2, ..., C_m$.

$$\sigma'_P(AR) = \{ar_i =< (C_1, AC_1), (C_2, AC_2), ..., (C_m, AC_m) > | P(C_1, C_2, ..., C_m) = true\}$$

Annotation-based selection (AWHERE, AHAVING)(Figure 14(d)): selects tuples from AR based on conditions P applied over the annotation lists $AC_1, AC_2, ..., AC_m$.

$$\theta_P(AR) = \{ar_i =< (C_1, AC_1), (C_2, AC_2), ..., (C_m, AC_m) > | P(AC_1 + AC_2 + ... + AC_m) = true\}$$

Annotation filtering (FILTER)(Figure 14(e)): selects all tuples of AR after filtering the annotations of each tuple based on conditions P applied over the annotation lists $AC_1, AC_2, ..., AC_m$. Annotations that satisfy P are the only annotations to pass.

$$\zeta_P(AR) = \{ar_i =< (C_1, F_P(AC_1)), (C_2, F_P(AC_2)), ..., (C_m, F_P(AC_m)) >\}$$

$F_P(AC_k)$ selects the annotations that satisfy P from the AC_k list.

Duplicate elimination (DISTINCT): reports one tuple per group of identical tuples of AR. The matching of tuples is based only on attributes C_1, C_2, ..., C_m, i.e., annotations are not considered while matching tuples. The annotations of the resulting tuple represent the union of the annotations of that tuple's identical copies.

$$\delta'(AR) = \{ar_i =< (C_1, AC'_1), (C_2, AC'_2), ..., (C_m, AC'_m) > | r_i =< C_1, C_2, ..., C_m > \in \delta(R), AC'_k = +(AC_k) \forall j : r_j = r_i\}$$

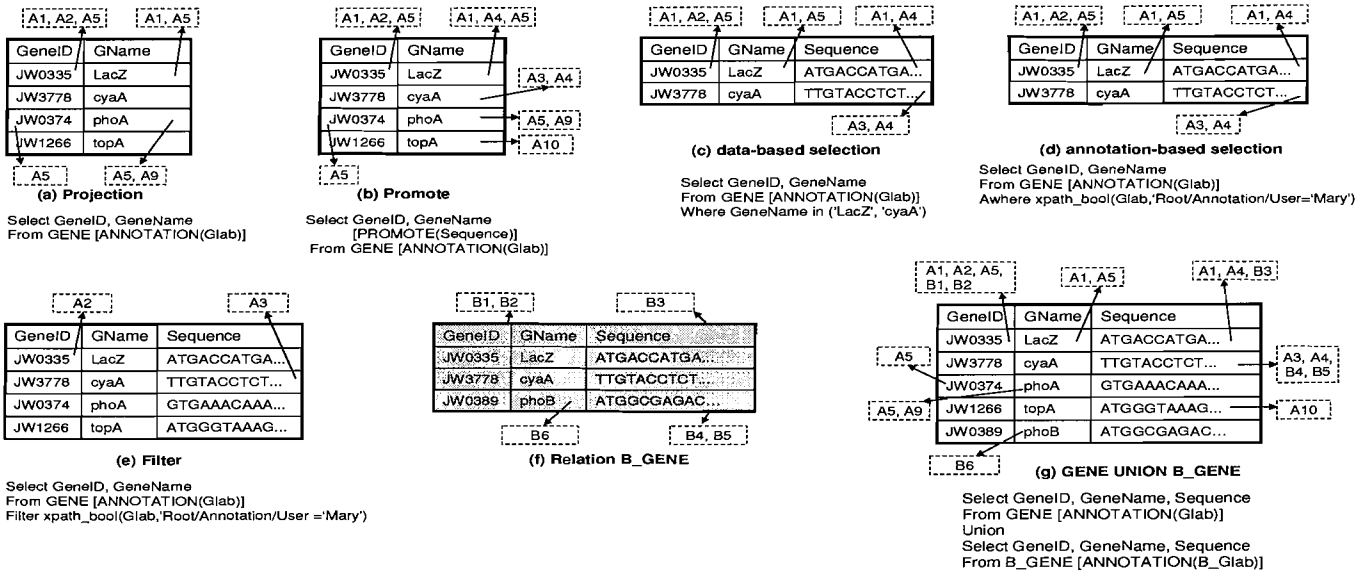


Figure 14: Extended SQL Operators

We illustrate the duplicate elimination example along with the union operator.

Set union (UNION)(Figure 14(g)): the union of AR and AS produces one copy of each tuple that appears in AR or AS or both. Identical copies of the same tuple are eliminated. The matching of tuples is based only on attributes C_1, C_2, ..., C_m and not on the associated annotations. The annotations of a resulted tuple represent the union of the annotations of the tuple's identical copies.

$$AR \sqcup' AS = \{at_i =< (C_1, AC'_1), (C_2, AC'_2), ..., (C_m, AC'_m) > | t_i =< C_1, C_2, ..., C_m > \in (R \sqcup S), AC'_k = +(AC_k) \forall j : r_j = t_i, s_j = t_i\}$$

The other set operations, e.g., INTERSECT and set difference (EXCEPT) have similar semantics. For example, the intersection between A_GENE and AA_GENE produces the first two rows of the relation illustrated in Figure 14(g). The difference between A_GENE and AA_GENE produces the third and fourth rows of the relation illustrated in Figure 14(g).

Cartesian product ($AR \times AS$): produces another annotated relation AT, where the annotations attached to a resulted tuple in AT are the annotations from both joined tuples in AR and AS.

$$AR \times' AS = \{at_i =< (C_1, AC_1), ..., (C_m, AC_m), (S_1, AS_1), ..., (S_n, AS_n) > | t_i =< C_1, ..., C_m, S_1, ..., S_n > \in (R \times S)\}$$

B. 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] P. Buneman, A. P. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGOMD*, 2006.

[4] P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. *Lecture Notes in Computer Science*, 1973:316–333, 2001.

[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] T. U. Consortium. The Universal Protein Resource (UniProt). *Nucl. Acids Res.*, 35(suppl_1):D193–197, 2007.

[8] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *ICDE*, pages 367–378, 2000.

[9] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB*, pages 471–480, 2001.

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

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

[12] 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.

[13] 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.

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

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

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

[17] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. *CIDR*, pages 262–276, 2005.

[18] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, pages 91–102, 1997.