bdbms – A Database Management System for Biological Data

Mohamed Y. Eltabakh   Mourad Ouzzani   Walid G. Aref
Department of Computer Science
Purdue University
West Lafayette, IN
{meltabak,mourad,aref}@cs.purdue.edu

ABSTRACT

Biologists are increasingly using databases for storing and managing their data. Biological databases typically consist of a mixture of raw data, metadata, sequences, annotations, and related data obtained from various sources. Current database technology lacks several functionalities that are needed by biological databases. In this paper, we introduce *bdbms*, an extensible prototype database management system for supporting biological data. bdbms extends the functionalities of current DBMSs with: (1) Annotation and provenance management including storage, indexing, manipulation, and querying of annotation and provenance as first class objects in bdbms, (2) Local dependency tracking to track the dependencies and derivations among data items, (3) Update authorization to support data curation via *content-based* authorization, in contrast to identity-based authorization, and (4) New access methods and their supporting operators that support pattern matching on various types of compressed biological data types. This paper presents the design of bdbms along with the techniques proposed to support these functionalities including an extension to SQL. We also outline some open issues in building bdbms.

1. INTRODUCTION

Biological databases are essential to biological experimentation and analysis. They are used at different stages of life science research to deposit raw data, store interpretations of experiments and results of analysis processes, and search for matching structures and sequences. As such, they represent the backbone of life sciences discoveries. However, current database technology has not kept pace with the proliferation and specific requirements of biological databases [25, 37]. In fact, the limited ability of database engines to furnish the needed functionalities to manage and process biological data properly has become a serious impediment to scientific progress.

This article is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/2.5/).
You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2007.

3rd Biennial Conference on Innovative Data Systems Research (CIDR) January 7-10, 2007, Asilomar, California, USA.

In many cases, biologists tend to store their data in flat files or spreadsheets mainly because current database systems lack several functionalities that are needed by biological databases, e.g., efficient support for sequences, annotations, and provenance. Once the data resides outside a database system, it loses effective and efficient manageability. Consequently, many of the advantages and functionalities that database systems offer are nullified and bypassed. It is thus important to break this inefficient and ineffective cycle by empowering database engines to operate directly on the data from within its natural habitat; the database system.

Biological databases evolve in an environment with rapidly changing experimental technologies and semantics of the information content and also in a social context that lacks absolute authority to verify correctness of information. Furthermore, because the only authority is the scientific community itself, biological databases often require some form of community-based curation. These characteristics make it difficult, even using good design strategies, to completely foresee the kinds of additional information (termed annotations) that, over time, may become necessary to attach to data in the database.

In this paper, we propose bdbms, an extensible prototype database engine for supporting and processing biological databases. While there are several functionalities of interest, we focus on the following key features: (1) Annotation and provenance management, (2) Local dependency tracking, (3) Update authorization, and (4) Non-traditional and novel access methods. bdbms will make fundamental advances in the use of biological databases through new native and transparent support mechanisms at the DBMS level.

Annotations and provenance data are treated as first-class objects inside bdbms. bdbms provides a framework that allows adding annotations/provenance at multiple granularities, i.e., table, tuple, column, and cell levels, archiving and restoring annotations, and querying the data based on the annotation/provenance values. In bdbms, we introduce an extension to SQL, termed Annotation-SQL, or *A-SQL* for short, to support the processing and querying of annotation and provenance information. A-SQL allows annotations and provenance data to be seamlessly propagated with query answers with minimal user programming.

In bdbms, we propose a systematic approach for tracking dependencies among database items. As a result, when a database item is modified, bdbms can track and mark any

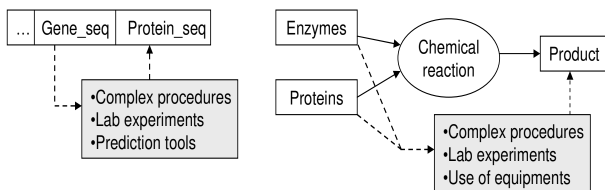

Figure 1: Local dependencies

other item that is affected by this modification and that needs to be re-verified. This feature is particularly desirable in biological databases because many dependencies cannot be computed using coded functions. For example (refer to Figure 1), protein sequences are derived from gene (DNA) sequences. If a gene sequence is modified, the corresponding protein sequence(s), derived calculated quantities (such as molecular weight), and annotations may become invalid. Similarly, we may store descriptions of chemical reactions, e.g., substrates, reaction parameters, and products. If any of the substrates in the reaction are modified, the products of the reaction are likely to require re-evaluation. However, since these dependencies are complex and involve lab experiments and external analysis, database systems cannot systematically re-compute the other affected items. Lack of system support to automatically track such dependencies raises significant concerns on the quality and the consistency of the data maintained in biological databases.

Authorizing database operations is also one of the features that we extended in bdbms. Current database systems support the GRANT/REVOKE access models that depend only on the identity of the user. In bdbms, we propose the concept of *content-based* authorization, i.e., the authorization is based not only on the identity of an updater but also on the content of the updated data. For example, lab members may have the authority to update a given data set. However, for credibility and reliability of the data, these updates have to be revised by the lab administrator. The lab administrator can then approve or disapprove the operations based on their contents. In the mean time, users may be allowed to view the data pending its approval/disapproval.

The other key feature in bdbms is to provide access methods for supporting various types of biological data. Our goal is to design and integrate non-traditional and novel access methods inside bdbms. For example, sequences and multi-dimensional data are very common in biological databases and hence there is a need to integrate new types of index structures such as SP-GiST [3, 4, 16, 22] and the SBC-tree [17] inside bdbms along with their supporting operators. SP-GiST is an extensible indexing framework for supporting multi-dimensional data while the SBC-tree is an index structure for indexing and querying compressed sequences without decompressing them.

The rest of the paper is organized as follows. In Section 2, we present the overall architecture of bdbms. In Sections 3–7, we present each of the bdbms key features. Section 8 overviews the related work, and Section 9 contains concluding remarks.

2. BDBMS SYSTEM ARCHITECTURE

The main components of bdbms are the annotation manager, the dependency manager, and the authorization manager. *A-SQL* is bdbms's extended SQL that supports annotation (Section 3) and authorization commands (Section 6). bdbms's *annotation manager* is responsible for handling the annotations in an *annotation storage* space (Section 3). The *dependency manager* is responsible for handling the dependencies and derivations among database items. These dependencies are stored in a *dependency storage* space (Section 5). The *authorization manager* handles *content-based* authorizations as well as the standard GRANT/REVOKE authorizations over the database (Section 6). *Index structures* are available in bdbms in support of the multidimensional and compressed data (Section 7).

3. ANNOTATION MANAGEMENT

Annotations are extra information linked to data items inside a database. They usually represent users' comments, experiences, related information that is not modeled by the database schema, or the provenance (lineage) of the data. Adding and retrieving annotations represent an important way of communication and interaction among database users. In biological databases, annotations are used extensively to allow users to have a better understanding of the data, e.g., how a piece of data is obtained, why some values are being added or modified, and which experiments or analyses are being performed to obtain a set of values. Annotations can be also used to track the provenance of the data, e.g., from which source a piece of data is obtained or which program is used to generate the data. Tracking the provenance of the data is very important in assessing the value and credibility of the data and in giving credit to the original data generators.

Users can annotate the data at multiple granularities, e.g., annotating an entire table, an entire column, a subset of the tuples, a few cells, or a combination of these.

Despite their importance, annotations are not systematically supported by most database systems. While annotation management has been addressed in previous works, e.g., [7, 8, 10, 35], most of the proposed techniques usually assume simple annotation schemes and focus mainly on annotation propagation, i.e., propagating annotations along with the query answers. Other aspects of annotations management, e.g., mechanisms for their insertion, archival, and indexing as well as more efficient annotation schemes such as multi-granularity schemes, have not been addressed.

In bdbms, we address several challenges and requirements of annotation management. We highlight these challenges and requirements through the following example. We consider two gene tables, `DB1_Gene` and `DB2_Gene` that have been obtained from two different databases (Refer to Figures 2 and 3 for illustration). Each table has a set of annotations attached to it. We assume a straightforward storage scheme for storing the annotations, e.g., the one used in [7], where each column in the database has an associated annotation column to store the annotations (Figure 3).

Adding annotations: Users want to annotate their data at various granularities in a transparent way. That is, how or where the annotations are stored should be transparent from end-users. However, current database systems do not provide a mechanism to facilitate annotating the data. For example, to add annotation *A2* over table `DB1_Gene` (Figure 2), the user has to know that the annotations are stored in the same user table in columns *Ann_GID*, *Ann_GName*, and *Ann_GSequence*. Then, the user issues an UPDATE

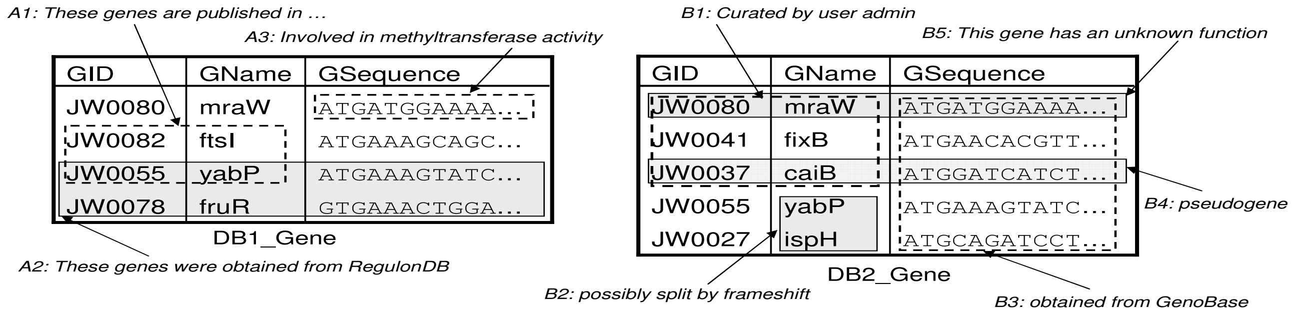

Figure 2: Annotating tables DB1_Gene and DB2_Gene

GID	Ann_GID	GName	Ann_GName	GSequence	Ann_GSequence
JW0080		mraW		ATGATGGAAAA...	*A3*
JW0082	*A1*	ftsI	*A1*	ATGAAAGCAGC...	
JW0055	*A1, A2*	yabP	*A1, A2*	ATGAAAGTATC...	*A2*
JW0078	*A2*	fruR	*A2*	GTGAAACTGGA...	*A2*

DB1_Gene

GID	Ann_GID	GName	Ann_GName	GSequence	Ann_GSequence
JW0080	*B1, B5*	mraW	*B1, B5*	ATGATGGAAAA...	*B3, B5*
JW0041	*B1*	fixB	*B1*	ATGAACACGTT...	*B3*
JW0037	*B1, B4*	caiB	*B1, B4*	ATGGATCATCT...	*B3, B4*
JW0055		yabP	*B2*	ATGAAAGTATC...	*B3*
JW0027		ispH	*B2*	ATGCAGATCCT...	*B3*

DB2_Gene

Figure 3: Simple annotation storage scheme: Every data column has a corresponding annotation column

statement to update these columns by adding *A2* to the desired annotation cells (Figure 3).

To support data annotation, the system has to provide new mechanisms for seamlessly adding annotations at various granularities. It is essential to provide new expressive commands as well as visualization tools that allow users to add their annotations graphically.

Storing annotations at multiple granularities: As in Figure 2, users may annotate a single cell, e.g., *A3*, few cells, e.g., *A1 and B2*, entire rows, e.g., *A2 and B4*, or entire columns, e.g., *B3*. Multi-granularity annotations motivate the need for efficient storage and indexing schemes. Otherwise, storing and processing the annotations can be very expensive. For example, annotations *A2* and *B3* are repeated in the annotation columns 6 and 5 times, respectively. The need for such efficient schemes is especially important in the context of provenance where a single provenance record can be attached to many tuples or even entire columns or tables.

Categorizing annotations: Although all annotations are metadata, they may have different importance, meaning, and creditability. For example, annotations that are added by a certain user or group of users can be more important than annotations added by the public or unknown users. Moreover, annotations that represent the lineage of the data have different purpose and importance from the annotations that represent users' comments. For example, annotations *A2* and *B3* represent the lineage of some data, and users may be interested only in these annotations. As will be discussed later in the paper, the different types of annotations will also have an impact on the storage mechanism adopted for each type. This diversity in annotations motivates the need for separating or categorizing the annotations. bdbms provides a mechanism that allows users to categorize their annotations at the storage, query processing, and annotation propagation levels.

Archiving annotations: Users may need to archive or delete annotations as they become obsolete, old, or simply invalid. Archived annotations should not be propagated to users along with query answers. For example, annotation *B5* in Figure 2 states that gene *JW0080* has an unknown function. But if the function of this gene becomes known and gets added to the database, then *B5* becomes invalid and users do not want to propagate this annotation along with query answers. Without providing a mechanism for archiving annotations, the archival operation may not be an easy task. For example, to archive annotation *B5*, the user needs to find out which tuples/cells in the database has *B5*, then the contents of each of these cells are parsed to archive then delete *B5*.

Propagating annotations: A key requirement in allowing annotation propagation is to simplify users' queries. This can be only achieved by providing database system support for annotation propagation; for example, by extending the query operators. Otherwise, users' queries may become complex and user-unfriendly. For example, consider a simple query that retrieves the genes that are common in `DB1_Gene` and `DB2_Gene` along with their annotations (Figure 3). To answer this query, the user has to write the following SELECT statements (a–c):

(a) $R_1(GID, GName, GSequence) =$
SELECT GID, GName, GSequence
FROM DB1_Gene
INTERSECT
SELECT GID, GName, GSequence
FROM DB2_Gene;

In Step (a), the user selects only the data columns from both gene tables, i.e., *GID, GName, GSequence*, and performs the intersection operation.

(b) $R_2(GID, GName, GSequence, Ann_GID, Ann_GName, Ann_GSequence) =$
SELECT R.GID, R.GName, R.GSequence,
G.Ann_GID, G.Ann_GName, G.Ann_GSequence
FROM R_1 R, DB1_Gene G

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

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

Figure 4: The A-SQL commands CREATE and DROP

WHERE R.GID = G.GID;

In Step (b), the user joins the output from Step (a) back with Table `DB1_Gene` in order to retrieve the annotations from this table. Notice that we cannot select the annotation columns in Step (a) because, since the annotation values in the annotation attributes may vary in the two tables, in this case the intersection operation would not return any tuples.

(*c*) $R_3(GID, GName, GSequence, Ann_GID,$
$Ann_GName, Ann_GSequence) =$
SELECT R.GID, R.GName, R.GSequence,
R.Ann_GID+G.Ann_GID,
R.Ann_GName+G.Ann_GName,
R.Ann_GSequence+G.Ann_GSequence
FROM R_2 R, DB2_Gene G
WHERE R.GID = G.GID;

In Step (c), a join is performed between R_2 and `DB2_Gene` to consolidate the annotations from `DB2_Gene` with R_2's annotations, where + is the annotation union operator.

The main reason for the complexity of querying and propagating the annotations is that users view annotations as metadata, whereas the DBMSs view annotations as normal data. For example, from a user's view point, the two tuples corresponding to genes *JW0080* and *JW0055* in Table *DB1_Gene* are identical to those in Table *DB2_Gene* (Figure 3). They only have different annotations. Whereas, from the database view point, these tuples are not identical because annotations are viewed as normal attribute data. As a result, users' queries may become complex in order to overcome the mismatch in interpreting the annotations.

In the following subsections, we introduce our initial investigations through bdbms to address the challenges and requirements highlighted above along with some preliminary results.

3.1 Storing and Indexing Annotations

bdbms allows a user relation to have multiple annotation tables attached to it. For example, table `DB1_Gene` may have an annotation table that stores the provenance information and another annotation table that stores users' comments. To create an annotation table over a given user relation, the A-SQL command *CREATE ANNOTATION TABLE* (Figure 4) is used. *CREATE ANNOTATION TABLE* allows users to design and categorize their annotations at the storage level. This categorization will also facilitate annotation propagation (discussed in Section 3.4), where users may request propagating a certain type of annotations. To drop an annotation table, the *DROP ANNOTATION TABLE* command (Figure 4) is used.

To efficiently store the annotations, we are investigating several storage and indexing schemes. One possible direction is to consider compact representation of annotations that would improve the system performance with respect to storage overhead, I/O cost to retrieve the annotations, and the query processing time. For example, instead of storing the annotations at the cell level, we may store some of the annotations at coarser granularities. For instance, the annotations over Table `DB2_Gene` (Figure 2) can be represented as rectangles attached to groups of contiguous cells as illustrated in Figure 5, where `DB2_Gene` is viewed as two-dimensional space, e.g., columns represent the X-axis and tuples represent the Y-axis. In this case, an annotation over any group of contiguous cells can be represented by a single annotation record. So, in general, an annotation over a subset of a table will map to multiple rectangular regions. Other annotation characteristics that may need to be taken into account include whether the annotation is linked to multiple data items in different tables or is linked to very few specific cells.

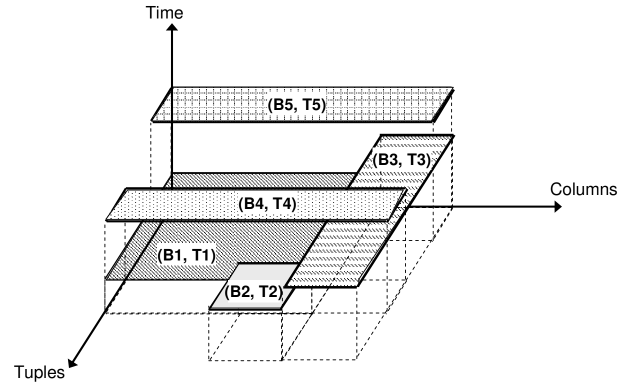

Figure 5: Compact storage for annotations

3.2 Adding Annotations at Multiple Granularities

To add annotations using A-SQL, we propose the *ADD ANNOTATION* command (Figure 6(a)). The *annotation_table_names* specifies to which annotation table(s) the added annotation will be stored. The *annotation_body* specifies the annotation value to be added. The output of the *SQL_statement* specifies the data to which the annotation is attached. Since annotations may contain important information that users want to query, we plan to support XML-formatted annotations. That is, *annotation_body* is an XML-formatted text. In this case, users can (semi-)structure their annotations and make use of XML querying capabilities over the annotations. The output of the *SQL_statement* can be at various granularities, e.g., entire tuples, columns, or group of cells. For example, to add annotation *B3* over the entire *GSequence* column in Table `DB2_Gene` (as illustrated in Figure 2), we execute the following ADD ANNOTATION command:

ADD ANNOTATION
TO DB2_Gene.GAnnotation
VALUE '< Annotation >
obtained from GenoBase
< /Annotation >'
ON (Select G.GSequence
From DB2_Gene G);

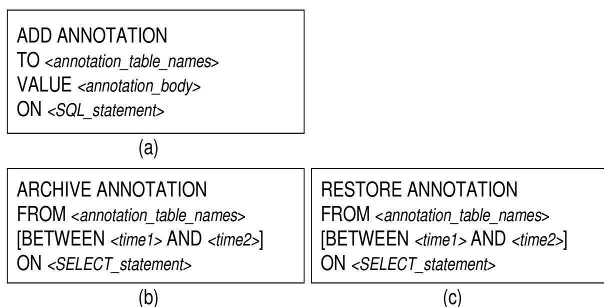

Figure 6: The A-SQL commands ADD, ARCHIVE, and RESTORE

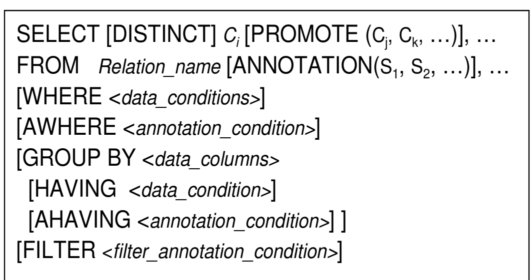

Figure 7: The A-SQL SELECT command

In this case, the annotation is attached to the entire *GSequence* column because no WHERE clause is specified. The annotation is stored in the annotation table *GAnnotation*. Notice that < *Annotation* > is the XML tag that encloses the annotation information.

Similarly, to annotate an entire tuple, e.g., annotation *B5*, we execute the following ADD ANNOTATION command:

```
ADD ANNOTATION
TO DB2_Gene.GAnnotation
VALUE '< Annotation >
            This gene has an unknown function
        < /Annotation >'
ON (Select G.*
      From DB2_Gene G
      WHERE GID = 'JW0080');
```

In this case, the annotation is attached to the entire tuples returned by the query since all the attributes in the table are selected.

To allow users to link annotations to database operations, i.e., INSERT, UPDATE, or DELETE, the *SQL_statement* will be an INSERT, UPDATE or DELETE statement. For example, instead of inserting a new tuple and then annotating it by issuing a separate ADD ANNOTATION command, users can insert and annotate the new tuple instantly by enclosing the insert statement inside the ADD ANNOTATION command. For the delete operation, the deleted tuples will be stored in separate log tables along with the annotation that specifies why these tuples have been deleted. Notice that the standard system recovery log cannot be used for this purpose as the users need the freedom to structure their annotation schemas the way they want, which system recovery logs do not support.

We plan to add a visualization tool to allow users to annotate their data in a transparent way. The visualization tool displays users' tables as grids or spreadsheets where users can select one or more cells to annotate. Oracle address the integration of database tables with Excel spreadsheets to make use of Excel visualization and analysis power [2]. In bdbms, we plan to add this integration feature to facilitate adding and visualizing annotations.

3.3 Archiving and Restoring Annotations

Archival of annotations allows users to isolate old or invalid annotations from recent and valuable ones. In bdbms, we support archival of annotations instead of permanently deleting them because biological data usually has a degree of uncertainty and old values may turn out to be the correct values. Archiving annotations gives users the flexibility to restore the annotations back if needed. Unlike other annotations, archived annotations are not propagated to users along with the query answers. However, if archived annotations are restored, then they will be propagated normally.

To archive and restore annotations, we introduce the *ARCHIVE ANNOTATION* (Figure 6(b)) and *RESTORE ANNOTATION* (Figure 6(c)) commands, respectively. The FROM clause specifies from which annotation table(s) the annotations will be archived/restored. The optional clause BETWEEN specifies a time range over which the annotations will be archived/restored. This time corresponds to the times-tamp assigned to each annotation when it is first added to the database. The output from the *SELECT_statement* specifies the data on which the annotations will be archived/restored. In addition, the output from the *SELECT_statement* can be at multiple granularities, as explained in the *ADD ANNOTATION* command.

3.4 Annotation Propagation and Annotation-based Querying

To support the propagation of annotations and querying of the data based on their annotations, we introduce the A-SQL command SELECT, given in Figure 7. A-SQL SELECT extends the standard SELECT by introducing new operators and extending the semantics of the standard operators. We introduce the new operators ANNOTATION, PROMOTE, AWHERE, AHAVING, and FILTER.

The ANNOTATION operator allows users to specify which annotation table(s) to consider in the query. Using the ANNOTATION operator, users can propagate their annotations transparently. That is, users do not have to know how or where annotations are stored. Instead, users only specify which annotations are of interest.

The PROMOTE operator allows users to copy annotations from one or more columns, possibly not in the projection list, to a projected column. For example, if column *GID* is projected from Table `DB1_Gene`, then Annotation *A3* will not be propagated unless the annotations over *GSequence* are copied to *GID*.

The AWHERE and AHAVING clauses are analogous to the standard WHERE and HAVING clauses except that the conditions of AWHERE and AHAVING are applied over the annotations. That is, AWHERE and AHAVING pass a tuple along with all its annotations only if the tuple's annotations satisfy the given AWHERE and AHAVING conditions. On the other hand, the FILTER clause passes all the data tuples of the input relation (keeps user's data intact) but it

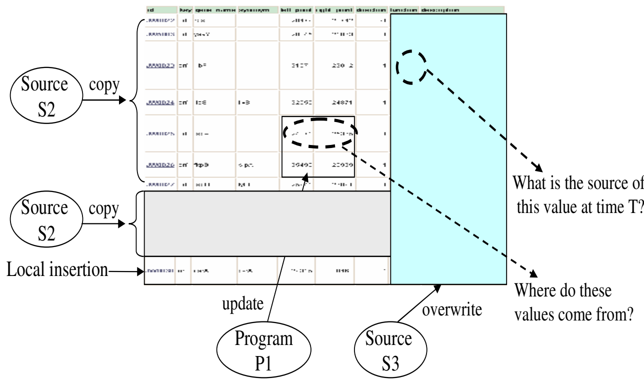

Figure 8: Data provenance at multiple granularities

filters the annotations attached to each tuple. That is, any annotation that does not satisfy *filter_annotation_condition* is dropped.

The standard operators, e.g., *projection*, *selection*, and *duplicate elimination*, are also extended to process the annotations attached to the tuples. For example, the *projection* operator selects some user attributes from the input relation and passes only the annotations attached to those attributes. For example, projecting column *GID* from Table `DB2_Gene` (Figure 2) results in reporting *GID* data along with annotations *B1, B4,* and *B5* only. The *selection* operators in WHERE and HAVING select tuples from the input relation based on conditions applied over the data values. The selected tuples are passed along with all their annotations. For example, selecting the gene with $GID = JW0080$ from Table `DB2_Gene` results in reporting the first tuple in `DB2_Gene` along with annotations *B1, B3,* and *B5*. Operators that group or combine multiple tuples into one tuple, e.g., *duplicate elimination*, *group by*, *union*, *intersect*, and *difference*, are also extended to handle the annotations attached to the tuples. These operators union the annotations over the grouped or combined tuples and attach them to the output tuple that represents the group.

While defining the above commands and operators is only the first step in supporting annotations and other features within bdbms, we need to define for each A-SQL operator its algebraic definition, cost estimate function, and algebraic properties that can be used by the query optimizer to generate efficient query plans.

4. PROVENANCE MANAGEMENT

Biologists commonly interact and exchange data with each other. Tracking the provenance (lineage) of data is very important in assessing the value and credibility of the data. Similar to annotations, data provenance can be attached to the database at multiple granularities, i.e., at the table, column, tuple levels, or any sub-groupings and subsets of the data. Also, biological data can be queried by its provenance. For example (refer to Figure 8), one table may contain data from multiple sources, e.g., *S1* and *S2*, or data that is locally inserted. Then, some values may be updated by a certain program, e.g., *P1*, and some columns may be overwritten by data from another source, e.g., *S3*. Then, users may be interested to know the source of some values at a certain moment in time.

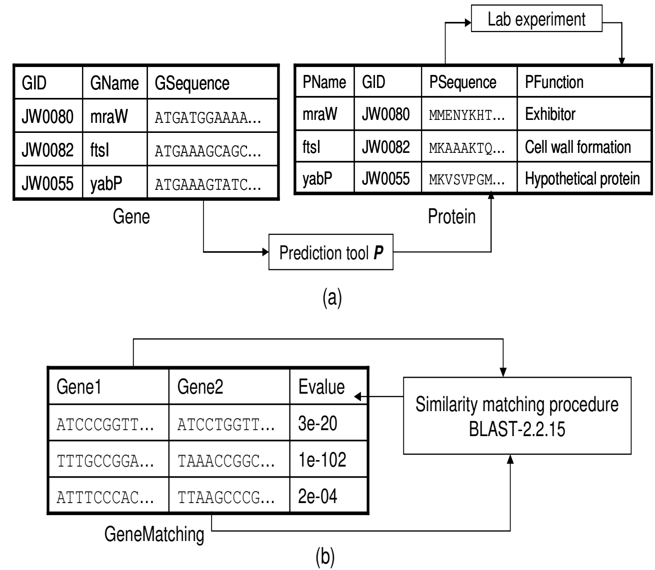

Figure 9: Local dependency tracking

In bdbms, we treat provenance data as a kind of annotations where all the requirements and functionalities discussed in Section 3 are also applicable to provenance data. However, provenance data has special requirements and characteristics that need to be addressed including:

- **Structure of provenance data:** Unlike annotations that can be free text, provenance data usually has well-defined structure. For example, the names of the source database and the source table draw their values from a list of pre-defined values. Supporting XML-formatted annotations can be beneficial in structuring provenance data. For example, provenance data can follow a predefined XML schema that needs to be stored and enforced by the database system.
- **Authorization over provenance data:** End-users are usually not allowed to insert or update the provenance data. Provenance data needs to be automatically inserted and maintained by the system. For example, integration tools that copy the data from one database to another can be the only tools that insert the provenance information. End-users can only retrieve or propagate this information. Therefore, we need to provide an access control mechanism over the provenance data (and annotations in general) to restrict the annotation operations, e.g., addition, archival, and propagation, to certain users or programs as required.

5. LOCAL DEPENDENCY TRACKING

Biological databases are full of dependencies and derivations among data items. In many cases, these dependencies and derivations cannot be automatically computed using coded functions, e.g., stored procedures or functions inside the database. Instead, they may involve prediction tools, lab experiments, or instruments to derive the data. Using integrity constraints and triggers to maintain the consistency of the data is limited to computable dependencies,

i.e., dependencies that can be computed via coded functions. However, non-computable dependencies cannot be directly handled using integrity constraints and triggers. In Figure 9, we give an example of the dependencies that can be found in biological databases. In Figure 9(a), protein sequences are derived from the gene sequences using a prediction tool *P*, whereas the function of the protein is derived from the protein sequence using lab experiments. If a gene sequence is modified, then all protein sequences that depend on that gene have to be marked as *outdated* until their values are re-verified. Moreover, the function of the outdated proteins has to be marked as *outdated* until their values are re-evaluated.

In Figure 9(b), we present another type of dependency where the value of the data in the database depends on the procedure or program that generated that data. For example, the values in the *Evalue* column (Figure 9(b)) depend on Procedure *BLAST-2.2.15*. If a newer version of *BLAST* is used or *BLAST* is replaced with another procedure, then we need to re-evaluate the values in the *Evalue* column. These values can be automatically evaluated if *BLAST* can be modeled as a database function. Otherwise, the values have to be marked as *Outdated*.

In bdbms, we propose to extend the concept of *Functional Dependencies* [5, 13] to *Procedural Dependencies*. In *Procedural Dependencies*, we not only track the dependency among the data, but also the type and characteristics of the dependency, e.g., the procedure on which the dependency is based, whether or not that procedure can be executed by the database, and whether or not that procedure is invertible. For example, we can model the dependencies in Figure 9 using the following rules.

Gene.GSequence —Prediction tool P (Executable, non-invertible)→ Protein.PSequence (1)

Protein.PSequence —Lab experiment (non-executable, non-invertible)→ Protein.PFunction (2)

GeneMatching.Gene1, GeneMatching.Gene2 —BLAST-2.2.15 (Executable, non-invertible)→ GeneMatching.Evalue (3)

Rule 1 specifies that Column *PSequence* in Table `Protein` depends on Column *GSequence* in Table `Gene` through the prediction tool *P* that is executable by the database and is non-invertible. Rule 2 specifies that column *PFunction* in Table `Protein` depends on Column *PSequence* through a lab experiment that is not executable by the database and is non-invertible. Rule 3 specifies that Column *Evalue* in Table `GeneMatching` depends on both columns *Gene1* and *Gene2* through Program *BLAST-2.2.15* that is executable by the database and is non-invertible. For example, from Rule 2, we infer that when Column *PSequence* changes, the database can only mark *PFucntion* as *Outdated*. In contrast, based on Rule 3, when either of the *Gene1* or *Gene2* columns or Procedure *BLAST-2.2.15* change, the database can automatically re-evaluate *Evalue*.

In addition, the notion of *Procedural Dependencies* allows us to reason about the dependency rules. For example, in addition to the closure of an attribute, we can compute the closure of a procedure, i.e., all data in the database that depend on a specific procedure. We can also derive new rules, for example, based on rules (1) and (2) above, we can derive the following rule:

Gene.GSequence —Prediction tool P, lab experiment (non-executable, non-invertible)→ Protein.PFunction (4)

Rule 4 specifies that Column *PFunction* in Table `Protein` depends on Column *GSequence* in Table `Gene` through a chain of two procedures, a perdition tool *P* and a lab experiment. This chain is non-executable by the database and is non-invertible. Notice that the chain is non-executable because at least one of the procedures, namely the lab experiment, is non-executable.

In bdbms, we address the following functionalities to track local dependencies:

- **Modeling dependencies:** We use *Procedural Dependencies* to allow users to model the dependencies among the database items as well as for bdbms to reason about these dependencies, for example, to detect conflicts and cycles among dependency rules, and to compute the closure of procedures.

- **Storing dependencies:** Dependencies among the data can be either at the schema level, i.e., the entity level, or at the instance level, i.e., the cell level. Schema-level dependencies can be modeled using foreign key constraints, e.g., protein sequences depend on gene sequences and they are linked by a foreign key. Instance-level dependencies are more complex to model because they are on a cell-by-cell basis. In this case, we can use dependency graphs to model such dependencies.

- **Tracking outdated data:** When the database is modified, bdbms uses the dependency graphs to figure out which items, termed the *outdated* items, may be affected by this modification. *Outdated* items need to be marked such that these items can be identified in any future reference. We propose to associate a bitmap with each table in the database. A cell in the bitmap is set to 1 if the corresponding cell in the data table is *outdated*, otherwise the bitmap cell is set to 0. For example, assume that the sequences corresponding to genes *JW0080* and *JW0082* (Figure 9(a)) are modified, then the bitmap associated with Table `Protein` will be as illustrated in Figure 10. Notice that the bits corresponding to *PSequence* are not set to 1 because *PSequence* is automatically updated by executing Procedure *P*. In contrast, *PFunction* cannot be automatically updated, therefore its corresponding bits are set to 1 to indicate that these values are outdated. To reduce the storage overhead of the maintained bitmaps, data compression techniques such as Run-Length-Encoding [23] can be used to effectively compress the bitmaps.

- **Reporting and annotating outdated data:** The main objective of tracking local dependencies is that the database should be able to report at all times the items that need to be verified or re-evaluated. Moreover, when a query executes over the database and

PName	GID	PSequence	PFunction
mraW	JW0080	MKENYKNM...	Exhibitor
ftsI	JW0082	MTATTKTQ...	Cell wall formation
yabP	JW0055	MKVSVPGM...	Hypothetical protein

Protein

PName	GID	PSeq.	PFun.
0	0	0	1
0	0	0	1
0	0	0	0

Protein-Bitmap

Figure 10: Use of bitmaps to mark outdated data

```
START CONTENT APPROVAL
ON <table_name>
[COLUMNS <column_names>]
APPROVED BY <user/group>
```

```
STOP CONTENT APPROVAL
ON <table_name>
[COLUMNS <column_names>]
```

Figure 11: Content-based approval

involves *outdated* items, the database should propagate with those items an annotation specifying that the query answer may not be correct. Detecting the *outdated* items at query execution time is a challenging problem as it requires retrieving and propagating the status of each item, i.e., whether it is outdated or not, in the query pipeline. A proposed solution is to consider the status of the database items as annotations attached to those items. These annotations will be automatically propagated along with the query answers as discussed in Section 3.

- **Validating outdated data:** bdbms will provide a mechanism for users to validate *outdated* items. An *outdated* item may or may not need to be modified to become valid. For example, a modification to a gene sequence may not affect the corresponding protein sequence. In this case, the protein sequence will be revalidated without modifying its value.

6. UPDATE AUTHORIZATION

Changes over the database may have important consequences, and hence, they should be subject to authorization and approval by authorized entities before these changes become permanent in the database. Update authorization (also termed *approval enforcement*) in current database management systems is based on GRANT/REVOKE access models [18, 24], where a user may be granted an authorization to update a certain table or attribute. Although widely acceptable, these authorization models are based only on the identity of the user not on the content of the data being inserted or updated. In biological databases, it is often the case that a data item can make it permanently to a database based on its value not on the user who entered that value. For example, a lab administrator may allow his/her lab members to perform insert and update operations over the database. However, for reliability, these operations have to be revised by the lab administrator. If the lab administrator is the only user who has the right to update the database, then this person may become a bottleneck in the process of populating the database.

In bdbms, we introduce an approval mechanism, termed *content-based approval*, that allows the database to systematically track the changes over the database. The proposed content-based approval mechanism works with, not in replacement to, existing GRANT/REVOKE mechanisms. The content-based approval mechanism maintains a log of all update operations, i.e., INSERT, UPDATE, and DELETE, that occur in the database. The database administrator can turn the content-based approval feature ON or OFF for a certain table or columns using a *Start Content Approval* and *End Content Approval* commands (Figure 11), respectively. The *table_name* value specifies the user table on which the update operations will be monitored. The optional clause COLUMNS specifies which column(s) in *table_name* to monitor. For example, we can monitor the update operations over only Column *GSequence* of Table `Gene` (Figure 9(a)). The APPROVED BY clause specifies the user or group of users who can approve or disapprove the update operations. If the content-based approval feature is turned ON over Table T, then bdbms stores all update operations over T in the log along with an automatically generated *inverse* statement that negates the effect of the original statement. More specifically, for INSERT, a DELETE statement will be generated, for DELETE, an INSERT statement will be generated, and for UPDATE, another UPDATE statement that restores the old values will be generated. The log stores also the user identifier who issued the update operation and the issuing time. The person in charge of the database, e.g., the lab administrator, can then view the maintained log and revise the updates that occurred in the database. If an operation is disapproved, then bdbms executes the inverse statement of that operation to remove its effect from the database. Executing the inverse statement may affect other elements in the database, e.g., elements that depend on the currently existing values. It is the functionality of the *Local Dependency Tracking* feature (Section 5) to track and invalidate these elements.

7. INDEXING AND QUERY PROCESSING

Biological databases warrant the use of non-traditional indexing mechanisms beyond B+-trees and hash tables. To enable biological algorithms to operate efficiently on the database, we propose integrating non-traditional indexing techniques inside bdbms. We focus on two fronts: (1) Supporting multidimensional datasets via multidimensional indexing techniques (suitable for protein 3D structures and surface shape matching), and (2) Supporting compressed datasets via novel external-memory indexes that work over the compressed data without decompressing it (suitable for indexing large sequences).

In bdbms, we focus on introducing non-traditional index structures for supporting biological data. For example, compressing the data inside the database is proven to improve the system performance, e.g., C-store [33]. It reduces significantly the size of the data, the number of I/O operations required to retrieve the data, and the buffer requirements. In bdbms, we investigate how we can store biological data in compressed form and yet be able to operate, e.g., index, search, and retrieve, on the compressed data without decompressing it.

7.1 Indexing Multi-dimensional Data

Space-partitioning trees are a family of access methods that index objects in a multi-dimensional space, e.g., protein 3D structures. In [3, 4, 16, 22], we introduce an exten-

Protein secondary structure:
LLLEEEEEEEHHHHHHHHHHHHHHHHHHHHHHEEEEEELLEEELHHHHHHHHHHLL
LLLLLLLLHHHHHHHHHHHHHHHHLLLLEEEEEEEHHHHHHHHHHHHEEEEEEEEEE
LLLLHHHHHHHLLLLHHHHHHHHHHHHHHEEEEEEEEEEHHHHHHHEEEEEEEEHH
HHHHHHHHEEEELEEEEEEEEEELLLEEEEEEEELLLLHHHHHHHHHHHHHHHEEEE
EELLEEEELLLLLLLLHHHHHHHHHHHHHHHHHHHHEEEELEEEEEEEEEELEEEEEL
LLLLLLLLEEEEELLLLLLEEEEEEEELEEEEEEEEELLLEEEEHHHHHHHHHHHHHHH
HHHEEEEELLLEEEEEEEEELLLHHHHHHHHHHHHHHHHHHHHLHHHHHHHHHHHH
EEEEELEEEEHHHHHHHHHHHHHHHHHEEEEEELLLLLEEEEEEELLLLEEEEEEEEE
EEEELEEEEEEEEEEEEEEHHHHHHHHHHHHHHLLLLLEEEEEEEEEEHHHHHHHEE
EEEEHHHHHHHHHHLLLLLLHHHHHHHHHHHEEEEEEEEEEEHHHHHHHHHHHHHL
LEEEEELLLLLLLLLLHHHHHHHHHHHHHHHHHHLLLEEEEEEEHHHHHHHHHLLLL
EEEEEEEEEEEEEEEEEELLLLEEELLHHHHHHHHHLLLLLLLLLLLHHHHHHHHHHHH
HHHHHHHHEEEEEEEEEEELEEEEHHHHHHHHHHHHLHHHHHHHHHHHHHHLLEE
EEEEEELLLLEEEEEEEEELLLLLEEEEELLLLLEEEEEEEEELLLEEEEEEEEELLLEEE
HHHHHHHHHHHHHLLLL

Sequence compression

RLE compressed form:
L3E7H22E6L2E3L1H10L10H16L4E7H12E10L4H7L4H14E10H7E8H10E4L1E10L3E8L
4H15E6L2E4L8H20E4L1E10L1E5L9E5L6E8L1E9L3E4H18E5L3E9L3H20L1H12E5L1E
4H17E6L5E7L4E13L1E14H14L5E10H7E6H10L6H11E11H13L2E5L10H18L3E7H9L4E
18L4E3L2H9L11H20E11L1E4H12L1H14L2E8L4E9L5E5L5E9L3E9L3E3H13L4

Indexing compressed sequences

SBC-tree Index

Figure 12: Indexing and querying RLE-compressed sequences

sible indexing framework, termed SP-GiST, that broadens the class of supported indexes to include disk-based versions of space-partitioning trees, e.g., disk-based trie variants, quadtree variants, and kd-trees. As an extensible indexing framework, SP-GiST allows developers to instantiate a variety of index structures in an efficient way through pluggable modules and without modifying the database engine. The SP-GiST framework is implemented inside PostgreSQL [34] and we use it in bdbms. Several index structures have been instantiated using SP-GiST, e.g., variants of the trie [11, 20], the kd-tree [6], the point quadtree [19], and the PMR quadtree [29]. We implemented several advanced search operations, e.g., k-nearest-neighbor search, regular expression match search, and substring searching. The experimental results in [16] demonstrate the performance potential of the class of space-partitioning tree indexes over the B+-tree and R-tree indexes, for the operations above. In addition to the performance gains and the advanced search functionalities provided by SP-GiST indexes, it is the ability to rapidly prototype these indexes inside bdbms that is most attractive.

A key challenge is to integrate SP-GiST indexes inside biological analysis algorithms such as protein structure alignment algorithms. Providing the index structures is the first step to improve the querying and processing capabilities of the analysis algorithms.

7.2 Indexing Compressed Data

Biological databases consist of large amounts of sequence data, e.g., genes, alleles, and protein primary and secondary structures. These sequences need to be stored, indexed, and searched efficiently. In bdbms, we propose to investigate new techniques for compressing biological sequences and operating over the compressed data without decompressing it. Sequence compression has been addressed recently in the C-Store database management system [33], where some operators, e.g., aggregation operators, can operate directly over the compressed data. Sequence compression is demonstrated to improve system performance as it reduces the size of the data significantly.

In bdbms, as a first step, we investigate the processing, e.g., indexing and querying, of Run-Length-Encoded (RLE) sequences. RLE [23] is a compression technique that replaces the consecutive repeats of a character C by one occurrence of C followed by C's frequency. One of the main challenges is how to operate on the compressed data without decompressing it. In [17], we proposed an index structure, termed the SBC-tree (<u>S</u>tring <u>B</u>-tree for <u>C</u>ompressed sequences), for indexing and searching RLE-compressed sequences of arbitrary length. In Figure 12, we illustrate how protein secondary structure sequences are stored in bdbms. We first compress the sequences using RLE, and then build an SBC-tree index over the compressed sequences. Queries over the sequences will use the index to retrieve the desired data without decompression. The SBC-tree is a two-level index structure based on the well-known String B-tree and a 3-sided range query structure. The SBC-tree supports substring as well as prefix matching, and range search operations over RLE-compressed sequences. The SBC-tree has an optimal external-memory space complexity as well as optimal search time for substring matching, prefix matching, and range search queries. More interestingly, SBC-tree has shown to be very practical to implement. The SBC-tree index is prototyped in PostgreSQL with an R-tree in place of the 3-sided structure. Preliminary performance results illustrate that using the SBC-tree to index RLE-compressed protein sequences achieves up to an order of magnitude reduction in storage, up to 30% reduction in I/Os for the insertion operations, and retains the optimal search performance achieved by the String B-tree over the uncompressed sequences.

In bdbms, we plan to address the following challenges regarding the processing of compressed data:

- **Full integration of the SBC-tree index:** To fully integrate the SBC-tree index inside bdbms we plan to address several query processing and optimization issues including: (1) supporting subsequence matching, and (2) providing accurate cost functions for estimating the cost of the index. Subsequence matching is an important operation over biological sequences as it is used in many algorithms such as sequence alignment algorithms. We plan to extend the supported operations of the SBC-tree index to include subsequence matching.

- **Processing various formats of compressed data:** Currently, bdbms supports indexing and querying RLE-compressed sequence data. RLE is effective in the case of sequences where characters have long repeats in tandem. Compression techniques like gzip and Burrows-Wheeler Transform (BWT) can be more effective in compressing the other kinds of data. Our plan is to investigate indexing and querying other formats of compressed data in addition to RLE-compressed sequences to efficiently support these data inside bdbms.

8. RELATED WORK

Periscope [30, 36] is an ongoing project that aims at defining a declarative query language for querying biological data. Periscope/SQ [36], a component of Periscope, introduces

new operators and data types that facilitate the processing and querying of sequence data. While the main focus of Periscope is on defining and supporting a new declarative query language, bdbms focuses on other functionalities that are required by biological databases, e.g., annotation and provenance management, local dependency tracking, update authorization, and non-traditional access methods.

Several annotation systems have been built to manage annotations over the web, e.g., [1, 26, 27, 28, 31, 32]. Biodas (Biological Distributed Annotation System) [1, 32] and Human Genome Browser [27] are specialized biological annotation systems to annotate genome sequences. They allow users to integrate genome annotation information from multiple web servers. Managing annotations and provenance in relational databases has been addressed in [7, 8, 10, 12, 21, 35]. In these techniques provenance data is pre-computed and stored inside the database as annotations. The main focus of these techniques is to propagate the annotations along with the query answer. Other aspects of annotation management, e.g., insertion, storage, and indexing, have not been addressed. Another approach for tracking provenance, termed the *lazy approach*, has been addressed in [9, 14, 15, 38], where provenance data is computed at query time. *Lazy approach* techniques require that the derivation steps of the data to be known and to be invertible such that the provenance information can be computed. In bdbms, we treat provenance data as a kind of annotations because the derivation of biological data is usually ad-hoc and does not necessarily follow certain functions or queries.

The access control and authorization process in current database systems is based on the GRANT/REVOKE model [18, 24]. Although widely acceptable, this model lacks being content-based, i.e., the authorization is based only on the identity of the user. In bdbms, we propose the *content-based* approval model that is based on the data as well as on the identity of the user.

9. CONCLUDING REMARKS

Two applications have been driving the bdbms project: building a database resource for the Escherichia coli (E. coli) model organism and a protein structure database project. Through these two projects, we realized the need for the functionalities that we address in bdbms, namely (1) Annotation and provenance management, (2) Local dependency tracking, (3) Update authorization, and (4) Non-traditional and novel access methods.

bdbms is currently being prototyped using PostgreSQL. In parallel work, we have extended relational algebra to operate on "annotated" relations. The A-SQL language and the content-based authorization model are currently under development in PostgreSQL. The SP-GIST and SBC-tree access methods are already integrated inside PostgreSQL. We are currently studying several optimizations, cost estimates, and complex operations over these indexes.

10. REFERENCES

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

[2] Exploiting the power of oracle using microsoft excel. *Oracle White Paper*, December 2004.

[3] W. G. Aref and I. F. Ilyas. An extensible index for spatial databases. In *Statistical and Scientific Database Management*, pages 49–58, 2001.

[4] W. G. Aref and I. F. Ilyas. Sp-gist: An extensible database index for supporting space partitioning trees. *Journal of Intelligent Information Systems*, 17(2-3):215–240, 2001.

[5] W. Armstrong. Dependency structures of database relationships. In *International Federation for Information Processing (IFIP)*, pages 580–583, 1974.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

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

[8] P. Buneman, A. P. Chapman, and J. Cheney. Provenance management in curated databases. In *ACM SIGMOD International Conference on Management of Data* , 2006.

[9] 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.

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

[11] W. A. Burkhard. Hashing and trie algorithms for partial match retrieval. *ACM Transactions Database Systems*, 1(2):175–187, 1976.

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

[13] E. Codd. A relational model for large shared data banks. In *Communications of the ACM 13:6*, pages 377–387, 1970.

[14] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *International Conference on Data Engineering*, pages 367–378, 2000.

[15] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *International Conference on Very Large Data Bases*, pages 471–480, 2001.

[16] M. Y. Eltabakh, R. H. Eltarras, and W. G. Aref. Space-partitioning trees in postgresql: Realization and performance. In *International Conference on Data Engineering*, pages 100–111, 2006.

[17] M. Y. Eltabakh, W.-K. Hon, R. Shah, W. G. Aref, and J. S. Vitter. The sbc-tree: An index for run-length compressed sequences. Technical Report CSD TR05-030, 2005.

[18] R. Fagin. On an authorization mechanism. *ACM Transactions on Database Systems (TODS)*, 3(3):310–319, 1978.

[19] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Information*, 4:1–9, 1974.

[20] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[21] F. Geerts, A. Kementsietsidis, and D. Milano. Mondrian: Annotating and querying databases through colors and blocks. In *International Conference*

on Data Engineering, page 82, 2006.

[22] T. M. Ghanem, R. Shah, M. F. Mokbel, W. G. Aref, and J. S. Vitter. Bulk operations for space-partitioning trees. In *International Conference on Data Engineering*, pages 29–40, 2004.

[23] S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12:399–401, 1966.

[24] P. P. Griffiths and B. W. Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems (TODS)*, 1(3):242–255, 1976.

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

[26] J. Kahan and R. S. M. Koivunen, E. Prud'Hommeaux. Annotea: An open rdf infrastructure for shared web annotations. *WWW10*, pages 623–632, 2001.

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

[28] D. LaLiberte and A. Braverman. A protocol for scalable group and public annotations. *WWW3*, pages 911–918, 1995.

[29] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *ACM SIGMOD International Conference on Management of Data*, pages 270–277, 1987.

[30] J. M. Patel. The role of declarative querying in bioinformatics. 7(1):89–92, 2003.

[31] M. A. Schickler, M. S. Mazer, and C. Brooks. Pan-browser support for annotations and other meta-information on theworld wide web. *WWW5*, pages 1063–1074, 1996.

[32] L. Stein, S. Eddy, and R. Dowell. Distributed sequence annotation system (das). *Washigton University, Technical Report WUCS-01-07*, 2001.

[33] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column oriented dbms. In *International Conference on Very Large Data Bases*, 2005.

[34] M. Stonebraker and G. Kemnitz. The postgres next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[35] W.-C. Tan. Containment of relational queries with annotation propagation. In *International Symposium on Database Programming Languages*, 2003.

[36] S. Tata, J. M. Patel, J. S. Friedman, and A. Swaroop. Declarative querying for biological sequences. In *International Conference on Data Engineering*, pages 87–96, 2006.

[37] T. Topaloglou. Biological data management: Research, practive and opportunities. In *International Conference on Very Large Data Bases*, pages 1233–1236, 2004.

[38] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *International Conference on Data Engineering*, pages 91–102, 1997.