

Supporting Real-world Activities in Database Management Systems

Mohamed El Tabakh
Walid Aref
Ahmed Elmagarmid
Yasin Silva
Mourad Ouzzani

CSD TR #09-001
March 2009

Supporting Real-world Activities in Database Management Systems

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

ABSTRACT

Databases are integral to many application domains in which the cycle of processing the data is complex and may involve real-world activities that are external to the database, e.g., wet-lab experiments, manual measurements, and collecting instrument readings. As a result, an update operation in the database may render dependent data items invalid or inconsistent until the real-world activities involved in deriving these items are re-performed and the output results are reflected back into the database. These real-world activities may take time to prepare for and to perform, and hence introduce inherently long time delays between the updates. The presence of these long delays between the updates along with the need for the intermediate results to be instantly available for querying makes integrating these real-world activities within the database systems a challenging task. In this paper, we address these challenges and propose techniques that reflect updates immediately into the database while keeping track of the dependent and *potentially invalid* data items until they are re-validated and reflecting their status in query results. The proposed system includes: (1) semantics and syntax for interfaces through which users can register real-world activities into the database system and express the dependencies of data items on these activities. (2) introducing new operator semantics that alert users when the returned query results contain potentially invalid or out-of-date data, and enable evaluating queries on either valid data only (no false positives), or both valid and potentially invalid data (including false positives), and (3) mechanisms for data invalidation and revalidation. The paper addresses several design issues and proposes directions for future work.

1. INTRODUCTION

In many application domains such as scientific experimentation in biology, chemistry, and physics, the cycle of processing the data and generating new results is complex and may involve sequences of human interactions and real-world

activities that are external to the database, e.g., wet-lab experiments, manual measurements, and collecting instrument readings. In traditional derived data, e.g., deriving age from the date-of-birth attribute, simple procedures internal to the database system can be coded and executed automatically to maintain the consistency of the data inside the database. In contrast, when the derivations among the data items depend on real-world activities, these derivations cannot be coded within the database. As a result, updating a database value will render all the dependent and derived values invalid until the real-world activity involved in the dependency, e.g., the wet-lab experiment, is re-executed and its output is propagated back and updated into the database. Because of this mandated delay in propagating the updates, parts of the underlying database remain inconsistent while it still needs to be available for querying. As a consequence, the quality of the returned query results becomes doubtful, e.g., the query result may contain potentially invalid data. Our focus in this paper is on supporting real-world activities inside the database systems and maintaining the consistency of the data that depend on these activities.

As a driving example, consider the scenario presented in Figure 1 from the biology domain. The gene binding site attribute (*BindingSite*) depends on the gene start position attribute (*StartPos*) through a lab experiment. Also, the gene function attribute (*GFunction*) depends on both the gene sequence (*GSeq*) and gene direction (*GDirection*) attributes through another lab experiment. Updating the start position of the 2nd and 4th genes (underlined values in *StartPos*) will invalidate the corresponding binding sites (the dotted cells) until the required lab experiment is re-conducted to find the new binding sites in the genome sequence. Similarly, updating the sequence of the 3rd and 5th genes (underlined values in *GSeq*) will invalidate the corresponding gene functions (the dotted cells) until they are re-evaluated. Consider now applying query *Q1* (Figure 1(b)) on the current instance of the database. Although the query result seems correct, it is missing crucial information that affects the quality of the answer as well as any decisions that are based on the result. For example, (1) the reported value ‘TCCA...’ in the result is potentially invalid because the start position on which this value depends is modified and the involved experiment is not yet re-conducted. (2) the reported values ‘JW0015, GTAA...’ are up-to-date, however, the tuple itself qualifies the query predicates based on the outdated value ‘F2’ in the third tuple. Hence, the tuple’s existence in the query answer is questionable, and (3) the tuple corresponding to gene ‘JW0019’ does not qualify for

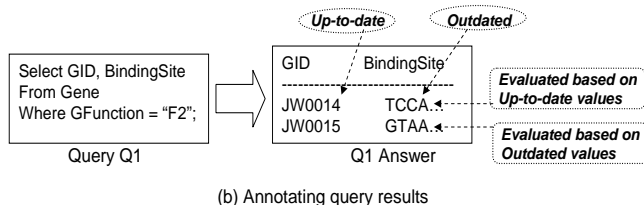
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB ’09, August 24-28, 2009, Lyon, France

Copyright 2009 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

| GID | StartPos | Binding site | GSeq | GDirection | GFunction |
|--------|----------|--------------|---------|------------|-----------|
| JW0013 | 5130 | AATG... | TGCT... | + | F1 |
| JW0014 | 10916 | TCCA... | GGTT... | + | F2 |
| JW0015 | 21112 | GTAA... | GGCT... | + | F2 |
| JW0018 | 31166 | CCGG... | CGTT... | - | F4 |
| JW0019 | 1905 | AAAT... | TGTG... | + | F5 |
| JW0012 | 17404 | GTAA... | TTCG | - | F7 |

(a) GENE table



(b) Annotating query results

Figure 1: Examples of data dependencies

the query, however, the function value of that gene is under re-evaluation, and it is possible that this gene will satisfy the query when its function is re-evaluated.

With the current state of the art, users may consider one of the following two options: (1) postpone the database updates until all dependent processes are executed and their outputs become consistent, then reflect these updates to the database at once (may involve unbounded long delays), or (2) reflect the updates immediately to the database and (temporarily) compromise the consistency of the derived data (and may lose track of the outdated data that needs re-evaluation). The former option is in fact compromising the consistency as well because users postpone reflecting valid updates to the database. Both options have serious problems as they delegate tracking the dependencies and maintaining the data consistency to the end-users instead of the database management system. In this paper, we propose a third, more appealing, option where users can reflect the updates immediately to the database, i.e., instant availability of the data, while the DBMS keeps track of the derived data by marking them as *potentially invalid* (*outdated*) and reflecting them in the queries' results until the external activities are re-executed and the outdated values are updated. Hence, the consistency of the data is not compromised.

The proposed system enables users to register real-world activities into the database and to define dependencies among data items based on these activities. We categorize dependencies into *computable* and *real-world* dependencies. Computable dependencies involve derivation functions that can be executed by the database system, e.g., user-defined functions created inside the database. On the other hand, real-world dependencies, involve real-world activities and human interactions that cannot be coded within the database, e.g., the wet-lab experiment that derives the gene function from its sequence (Figure 1(a)). The main contributions of the proposed system are summarized as follows:

- **Defining dependencies based on activities:** The proposed system enables users to register activities into the database system and to express the dependencies among the data items using these activities. Dependencies can be

column-based, i.e., every cell in the column follows the same dependency, or cell-based, i.e., different cells in a given column may follow different dependencies. Once the dependencies are defined in the database, they are automatically enforced by the DBMS.

- **Tracking outdated data:** A major benefit for defining real-world activities and dependencies inside the database is that the DBMS will keep track of the potentially outdated data items that are waiting on external activities to be performed. When a database value I (or an activity A) gets modified, all values that depend on I (or A) are marked as *outdated* to indicate that these values are potentially invalid and need re-evaluation. Outdated data can be reported to end-users for verification.

- **Introducing new query operators:** It is important to alert users of any potentially invalid data in the returned query results. Moreover, users may want to query only valid data and avoid building on any potentially invalid attribute values. In the proposed system, we introduce new query operators to: (a) reflect the status of the values in the query results as either *up-to-date* or *outdated*, (b) evaluate queries on only valid data and exclude any potentially invalid data (no false positives), or (c) evaluate queries on both valid and potentially invalid data (be conservative and include false positives).

- **Defining curation and manipulation mechanisms:** We propose systematic mechanisms for invalidating and re-validating the data inside the database. Moreover, we introduce new operators, termed *curation* operators, to manipulate user-defined dependencies among the database items. For example, the order in which the outdated data are revalidated is important because a database item I cannot be revalidated until all other items on which I depends on are up-to-date. The curation operators help identify the proper order of revalidation.

The rest of the paper is organized as follows. Section 2 overviews related work. Section 3 presents the needed definitions and axioms. Sections 4 and 5 introduce the new query operators as well as the invalidation and revalidation operators, respectively. In Section 6, we present several design issues. Extensions to the proposed model are discussed in 7. Section 8 contains concluding remarks and directions for future work.

2. RELATED WORK

The theory of functional dependencies (FDs), e.g., [11, 15], and its usage in ordinary databases are orthogonal to the problem highlighted in this paper. Functional dependencies provide systematic mechanisms for decomposing and normalizing a database schema to achieve a better database design. However, even with a good schema design and following the decomposition and normalization rules, the inconsistency problem of the data derived from real-world activities still exists. Several extensions to functional dependencies have been proposed to address other issues such as schema design in uncertain databases, e.g., [13], view updates and maintenance of materialized views, e.g., [12], and data cleaning and record linkage, e.g., [8]. However, none of this past work can be applied directly to the problem at hand. Other works have been proposed to support long-running transactions, e.g., [10], by loosening the ACID properties and avoiding locks, using optimistic concurrency control techniques, and using compensating transactions in the case of

failures. However, these techniques still expose invalid data for querying without notification mechanisms or special processing and hence do not solve the problems raised in this paper. Extensions to current DBMSs to process events have been proposed in [7, 16]. In these event processing systems, data tuples are viewed as streams of temporal events. The main functionalities include monitoring the coming events (tuples) and detecting the occurrences of pre-defined event sequences. Unlike these systems, the proposed model does not view the data tuples as events, however, there are external events, e.g., conducting a wet-lab experiment, that affect the status and consistency of the data.

Another related topic is in the area of *active databases*, e.g., [5, 6, 19, 20], where the DBMS supports mechanisms to respond automatically to events taking place either inside or outside the database. The outside events are ultimately mapped to operations that the database system can capture and respond to, e.g., insertion, deletion, update, calling a user-defined function, or a timed operation. The most common approach used by these systems is the *Event-Condition-Action* rules (ECA). Rule processing is integrated in many database systems, e.g., Postgres [14] and Starburst [17]. Unlike the *active database* model, in the proposed model, a change inside the database may trigger an execution (or re-execution) of a real-world activity. Until this activity is performed and its result is reflected back into the database, the system needs to keep track of the potentially outdated data items and reflect their status over query results. *Active databases* do not address these challenges.

Provenance management is a related research challenge that has been studied extensively in previous works to track the origin of the data and the process by which the data is generated, e.g., [1, 2, 3, 4, 18]. The two main approaches for representing provenance information are *inversion-based* and *annotation-based*. The inversion-based approach computes the provenance on-the-fly using inverted functions. This approach is not applicable to the addressed problem since we deal with external functions that cannot be executed by the database system in the first place. The annotation-based approach materializes the provenance information as annotations and enables users to both propagate the annotation along with the query results and query the data based on the annotation information. However, annotation-based techniques do not utilize the provenance information during data updates and hence they neither keep track of the outdated data that needs re-evaluation nor integrate the status of the data (up-to-date or outdated) in query processing. Therefore, current provenance management techniques do not address the challenges raised in this paper.

3. MODELING DEPENDENCIES AND ACTIVITIES

In this section, we present the needed definitions and axioms for activities and dependencies.

3.1 Definitions

Function (F): A function is a general term that refers to either a real-world activity or a user-defined function in the database. A function takes one or more input parameters and produces one output parameter. Each function F has a set of properties that specify (1) the function name, (2) the input and output types, and (3) the function type.

Similar to defining functions that are written in SQL, C, or Java languages, we extend the *Create Function* command to support a new type of functions called *real-world activity*. The DBMS interprets *real-world activity* functions as non-executable functions. Dependencies that involve this type of functions are interpreted as *real-world* dependencies. The following command defines a real-world activity:

```
Create Function <activity_name> (<input_types>)
Returns <output_type> As real-world activity;
```

For example, experiment *GeneFunExp* in Figure 1(a) is defined in the database using the following command:

```
Create Function GeneFunExp (text, char)
Returns text As real-world activity;
```

Dependency Instance (DI): A dependency instance is a dependency between a set of input parameters (database cells) and an output parameter (database cell) through a specific execution of a function. A dependency instance is defined as $DI = (F, SP, DP)$, where:

- **F:** The function involved in the dependency.
- **SP (Source Parameters):** An ordered set of database cells that are the input parameters to F .
- **DP (Destination Parameter):** A database cell that is the output parameter from F .

Throughout the paper, we focus on real-world dependencies. However, in order to properly maintain the consistency of the data, both real-world and computable dependencies need to be defined in the database. The reason is that when a database item I is invalidated (marked as outdated), all items that depend on I either through real-world or computable dependencies need to be invalidated as well. The key difference between the two dependency types is that when I is re-validated, items that depend on I through computable dependencies will be automatically validated whereas items that depend on I through real-world dependencies will remain invalid until the involved real-world activities are performed and the items are explicitly validated.

Analog to defining database constraints, we define the dependencies inside the database through extensions to both the *Create Table* and *Alter Table* commands. In the cases where the source and destination attributes of the dependency belong to the same table (or the source table(s) are already created in the database), the dependency can be defined as part of the *Create Table* command of the destination table. Otherwise, the dependency can be defined independently using the *Alter Table* command over the destination table of the dependency. To define new dependencies, the *Add Dependency* construct is added to the *Create Table* and *Alter Table* commands as follows:

The dependency is defined over table R that contains the destination attribute of the dependency $R.c_0$. The source tables, i.e., T_1, T_2, \dots , can be the same as table R or different. The optional *Where* clause contains predicates over the source and destination tables to specify (if needed) the exact table cells that are linked together. If the *Where* clause is omitted, then the source and destination tables must be the same. In this case the source and destination cells are assumed to belong to the same tuple. In the case of

```

Create Table <R>
(
  <columns_definitions>
  ....
  Add Dependency Using <func_name>
  Source <T1,c1[, T2,c2, ...]>
  Destination <R,c0>
  [Where <predicates>]
  [Invalidate Destination];
);

```

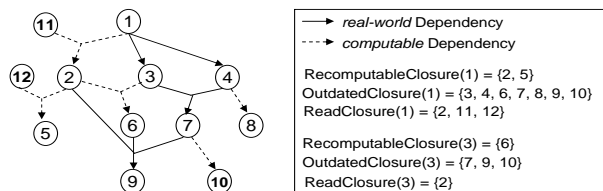


Figure 3: Examples of data closures

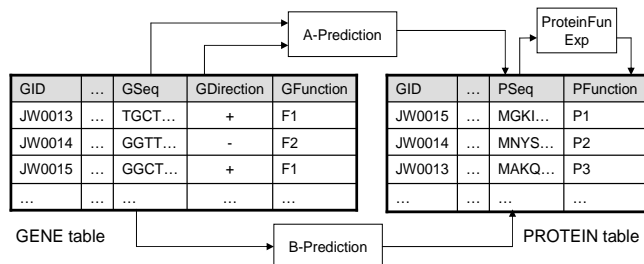


Figure 2: Dependencies across relations

Alter Table, the destination table may already contain data. Therefore, the optional clause *Invalidate Destination* is used to specify whether or not to invalidate the destination cells of the defined dependencies.

Example 1: Refer to the GENE table in Figure 1(a). The following extended *Create Table* command defines the dependency between the gene function and both the gene sequence and direction through experiment *GeneFuncExp*.

```

Create Table GENE(
  GSeq text,
  GDirction char,
  GFunction text,
  ...
  ADD Dependency Using GeneFuncExp
  Source GSeq, GDirection
  Destination GFunction);

```

Example 2: The example in Figure 2 shows dependencies that involve more than one table. Assume that the protein sequences corresponding to genes with function ‘F1’ are inferred using prediction tool *A-Prediction*, while the remaining proteins (corresponding to genes with function other than ‘F1’) are inferred using prediction tool *B-Prediction*. Notice that *A-Prediction* and *B-Prediction* have to be previously defined in the database using the *Create Function* command each with its own signature. The following command creates these dependencies.

```

Create Table Protein(
  GID text,
  PSeq text,
  GFunction text,
  ...
  ADD Dependency Using A-Prediction
  Source Gene.GSeq, Gene.GDirection
  Destination Protein.PSeq
  Where Protein.GID = Gene.GID
  And Gene.GFunction = 'F1',
  ADD Dependency Using B-Prediction
  Source Gene.GSeq

```

```

Destination Protein.PSeq
Where Protein.GID = Gene.GID
And Gene.GFunction != 'F1';

```

Dependencies can also be added after tables creation (and possibly after inserting data into the tables) using the *Alter Table* command. The *Add Dependency* construct may implicitly define one or more dependency instances. For example, the command in Example 1 implicitly defines one dependency instance for each tuple in table GENE. Whether or not each of these dependency instances will be created (materialized) inside the database is a design issue that we address in Section 6.

3.2 Axiomatization of Dependencies

In this section, we introduce axioms for user-defined dependencies.

3.2.1 The Cascading Property

Definition: Dependency instance DI_1 is cascaded by dependency instance DI_2 ($DI_1 \rightarrow DI_2$) if the destination parameter of DI_1 belongs to the source parameters of DI_2 ($DI_1.DP \subseteq DI_2.SP$).

The cascading property is used to form a composition of two or more dependencies where the output of the predecessor is the input to the successor. Based on this property, cycles are detected (Section 3.2.2) and closures of attributes are computed (Section 3.2.3).

3.2.2 The Cyclic Property

Definition: Dependency instances DI_1, DI_2, \dots, DI_n form a cycle if $DI_i \rightarrow DI_{i+1}$, for $1 \leq i < n$, and $DI_n \rightarrow DI_1$.

In the proposed model, cyclic dependency instances are not allowed. That is, there cannot be a sequence of derivations that leads to a cycle. If dependency DI_j is to be defined in the database and will form a cycle with existing dependencies, then the DBMS will reject DI_j .

Since cycles are not allowed in the database, the following property holds.

Property 1: The user-defined dependency instances create a set of acyclic dependency graphs (DAGs) among the database items.

3.2.3 The Closure of Attributes and Functions

For a given database cell C , we define *InputParameters*(C) to be the set of database cells that are read to re-compute C . If C is an output of an executable function F , then *InputParameters*(C) is the set of input parameters to F . Otherwise, *InputParameters*(C) is empty. Given a set of user-defined dependency instances, we define three types of closures for C as follows (See Figure 3 for an example):

RecomputableClosure(C): is the set of database cells $\{C'\}$ that are re-computed when C is modified. The set $\{C'\}$ can be reached from C using the cascading property of dependency instances that involve only *executable* functions.

OutdatedClosure(C): is the set of database cells $\{C'\}$ that are marked as *outdated* when C is modified. The set $\{C'\}$ can be reached from C using the cascading property of dependency instances that involve at least one *real-world* activity function.

ReadClosure(C): is the set of database cells that are read to re-compute the *RecomputableClosure(C)* when C is modified. The *ReadClosure(C)* is the union of the *InputParameters(C')*, $\forall C' \in \text{RecomputableClosure}(C)$.

The same types of closures are defined for functions. Function closures are useful in tracking the data items that need to be re-computed or invalidated when users, for example, upgrade or replace an existing function by another one or a newer version. Lemma 1 and Lemma 2 state the function closures in terms of the attributes closures for real-world and executable functions, respectively.

Lemma 1: For a real-world activity function F :

$$\begin{aligned} \text{RecomputableClosure}(F) &:= \phi, \\ \text{ReadClosure}(F) &:= \phi, \\ \text{OutdatedClosure}(F) &:= C \cup \text{OutdatedClosure}(C) \cup \\ &\quad \text{RecomputableClosure}(C), \forall \text{ output cell } C \text{ from } F. \end{aligned}$$

Lemma 2: For an executable function F :

$$\begin{aligned} \text{RecomputableClosure}(F) &:= C \cup \text{RecomputableClosure}(C), \\ &\quad \forall \text{ output cell } C \text{ from } F, \\ \text{ReadClosure}(F) &:= \cup \text{InputParameters}(C'), \\ &\quad \forall C' \in \text{RecomputableClosure}(F), \\ \text{OutdatedClosure}(F) &:= \text{OutdatedClosure}(C), \\ &\quad \forall \text{ output cell } C \text{ from } F. \end{aligned}$$

3.2.4 The Overriding Property

Definition: Dependency instance DI_1 is said to override dependency instance DI_2 ($DI_1 \rightsquigarrow DI_2$) if both instances have the same destination parameter ($DI_1.DP = DI_2.DP$) and DI_1 is defined after DI_2 .

Dependency instances capture the way the data are derived. As time passes, the derivation mechanisms among the data items may change, and hence the dependencies need to change. The proposed model allows newly defined dependencies to override existing dependencies to indicate a change in the derivation of the underlying data.

Example 3: Continuing with Example 1, assume that a biologist uses a new experiment, called *GeneFunExpNew*, to infer the function of genes *JW0013* and *JW0014* (The first two rows in table *GENE* presented in Figure 1(a)). Unlike the old experiment, the new experiment depends only on the gene sequence as input to infer the gene function. In this case, the derivations of these gene functions no longer follow the dependencies defined in Example 1. We need to define the *GeneFunExpNew* experiment in the database, and then create the new dependencies as follows:

```
Alter Table Gene
ADD Dependency Using GeneFunExpNew
Source GSeq
Destination GFunction
Where GID In ('JW0013', 'JW0014')
Invalidate Destination;
```

| GID | StartPos | Binding site | GSeq | GDirection | GFunction | Predicate 1 | Predicate 2 |
|--------|----------|--------------|---------|------------|-----------|------------------|--|
| | | | | | | GFunction = "F2" | GFunction = "F2" AND BindingSite="GTAA..." |
| JW0013 | 5130 | AATG... | TGCT... | + | F1 | F | F |
| JW0014 | 10916 | TCCA... | GGTT... | + | F2 | T | -ve |
| JW0015 | 21112 | GTAA... | GGCT... | + | F2 | +ve | +ve |
| JW0018 | 31166 | CCGG... | CGTT... | - | F4 | F | F |
| JW0019 | 1905 | AAAT... | TGTG... | + | F5 | -ve | F |
| JW0012 | 17404 | GTAA... | TTCG | - | F7 | F | F |

(a) GENE table

(b) Predicate Evaluation

Figure 4: Examples of predicate evaluation

Notice that the newly defined dependencies override the existing dependencies only for the two genes *JW0013* and *JW0014*. All the other gene functions still follow the dependencies defined in Example 1. The *Invalidate Destination* clause causes the gene functions of the specified genes to be marked as outdated until their new values are reflected into the database.

4. DATA QUERYING

In the proposed model, parts of the underlying database are marked as valid (up-to-date), while other parts are marked as potentially invalid (outdated). Therefore, it is important that the querying system reports back, as part of the query results, the status information of the values in the results, e.g., whether each value is up-to-date or outdated. Moreover, some users may prefer to avoid querying any suspicious (outdated) data until being revalidated even if these data satisfy the query, i.e., avoid getting false positive results. Other users may prefer conservative answers and prefer getting even suspicious data that have the potential to satisfy their queries, i.e., include false negative results. In this section, we introduce new operators to support these data querying capabilities. Conceptually, every table cell in the database has a status flag (0 = up-to-date, 1 = outdated) in addition to the cell value. That is, Relation R with n attributes is represented as: $R = \{r = \langle (C_1.value, C_1.status), \dots, (C_n.value, C_n.status) \rangle\}$.

4.1 Predicate Evaluation

The evaluation of a predicate over a tuple typically results in a boolean value True or False. With the status of each value in the database as *up-to-date* or *outdated*, we extend the predicate evaluation to return one of four possible values: True (T), False (F), Potentially false positive (+ve), and Potentially false negative (-ve) (See the example in Figure 4). The *True* value indicates that the tuple qualifies the predicates based on only *up-to-date* values, and hence, it is certainly part of the answer. The *False* value indicates that the tuple disqualifies the predicates based on only *up-to-date* values, and hence, it is certainly not part of the answer. The *Potentially false positive* value indicates that the tuple qualifies the predicates but based on *outdated* values, and hence, it is potentially false positive. The *Potentially false negative* value indicates that the tuple disqualifies the predicates based on *outdated* values, and hence, it is potentially false negative. Figure 4 illustrates two examples of evaluating predicates over the *GENE* table. Since the rules for evalu-

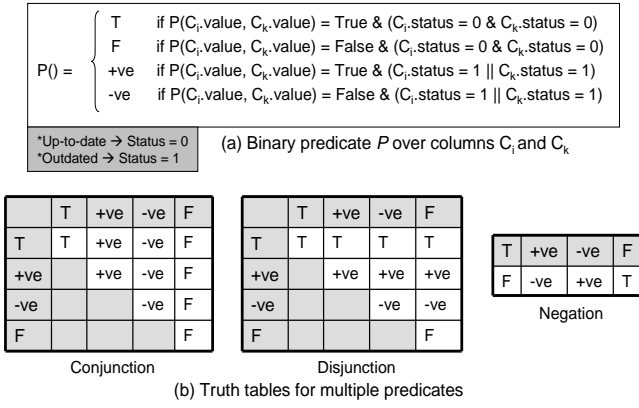


Figure 5: Predicate evaluation rules

ating unary predicates ($column \langle op \rangle constant$) are trivial, we present the rules for evaluating only binary predicates ($column \langle op \rangle column$) in Figures 5(a). The truth tables for evaluating multiple predicates with conjunction, disjunction, and negation are presented in Figure 5(b). For example, consider the second tuple in the **GENE** table against Predicate2 (Figure 4). The tuple satisfies the predicate ($GFunction = 'F2'$) with up-to-date value, and hence that part evaluates to T . However, the tuple disqualifies the predicate ($BindingSite = 'GTAA...'$) with an outdated value, and hence that part evaluates to $-ve$ in contrast to false F . The conjunction of these two sub-predicates evaluates the entire tuple to $-ve$ according to Figure 5(b).

4.2 New Query Operators

- **Selection:** With the extended semantics of predicate evaluation, the selection operator is extended to return the tuples of interest. Tuples that evaluate to T , $+ve$, or $-ve$ are of interest since they either satisfy or have the potential to satisfy the query. However, it is clear that returning these tuples altogether from one operator can be very misleading and hard to interpret. In the proposed model, we define three separate types of selection operators, namely, *True Selection* (σ_T), *False-positive Selection* (σ_+), and *False-negative Selection* (σ_-), that return tuples that evaluate to each of the values T , $+ve$, or $-ve$, respectively. The algebraic expressions of the selection operators are as follow.

True Selection ($\sigma_{T,P}$): Selects tuples that evaluate predicate P to T .
 $\sigma_{T,P}(R) = \{r = \langle (C_1.value, C_1.status), \dots, (C_n.value, C_n.status) \rangle, |P(r) = T\}$

False-positive Selection ($\sigma_{+,P}$): Selects tuples that evaluate predicate P to $+ve$.
 $\sigma_{+,P}(R) = \{r = \langle (C_1.value, C_1.status), \dots, (C_n.value, C_n.status) \rangle, |P(r) = +ve\}$

False-negative Selection ($\sigma_{-,P}$): Selects tuples that evaluate predicate P to $-ve$.
 $\sigma_{-,P}(R) = \{r = \langle (C_1.value, C_1.status), \dots, (C_n.value, C_n.status) \rangle, |P(r) = -ve\}$

Query plans may involve one or more types of selection operators to return the desired tuples. For example, referring

to Figure 4, the expression $\sigma_{T,P}(GENE) \cup \sigma_{+,P}(GENE)$, where predicate P is ($GFunction = 'F2'$), will return the second and third tuples from the **GENE** table. Notice that the standard selection operator (σ_P) is equivalent to the union of the two operators $\sigma_{T,P}$ and $\sigma_{+,P}$.

- **Inner Join:** The evaluation of a join predicate over a pair of tuples r and s results in one of four possible values, i.e., T , $+ve$, $-ve$, or F . Join predicates are binary predicates and hence they follow the evaluation rules presented in Figure 5(a). We define three types of join operators, namely, *True Join*, *False-positive Join*, and *False-negative Join*, that return tuples that evaluate to each of the values T , $+ve$, $-ve$, respectively. The algebraic expression of the *True Join* operator is as follows.

True Join ($R \bowtie_{T,P} S$): Returns the joined tuples r and s that evaluate predicate P to T .
 $R \bowtie_{T,P} S = \{z = \langle (r_1.value, r_1.status), \dots, (s_1.value, s_1.status), \dots, (s_m.value, s_m.status) \rangle \mid P(z) = T\}$

The algebraic expressions of the *False-positive* ($\bowtie_{+,P}$) and *False-negative* ($\bowtie_{-,P}$) join operators are similar to that of the *True* join operator with the exception of having the join predicate $P(z)$ evaluates to $+ve$ and $-ve$, respectively. Notice that the standard *inner join* operator (\bowtie_P) is equivalent to the union of the two operators $\bowtie_{T,P}$ and $\bowtie_{+,P}$.

- **Duplicate Elimination and Set Union:** The key difference between these extended operators and their standard counterparts is in identifying the identical tuples. In the extended semantics, two tuples are considered identical iff they share the same value and status for all their attributes. More formally, two tuples r and s are considered identical w.r.t columns c_1, c_2, \dots, c_n iff: $(r.c_i.value = s.c_i.value \ \& \ r.c_i.status = s.c_i.status) \ \forall \ i \in \{1, 2, \dots, n\}$.

- **Set Difference** ($R - S$): We define three types of set difference operators namely, *True Difference* ($-_T$), *False-positive Difference* ($-_+$), and *False-negative Difference* ($-_-$). The algebraic expressions of the set difference operators are as follow:

True Difference: Reports tuples in R that are guaranteed not to exist in S even after re-evaluating the outdated values in both relations, i.e., the unmatched depends on up-to-date values.
 $R -_T S = \{r \in R \mid \nexists s \in S \text{ where } r.c.value = s.c.value \ \forall \ c \text{ in which } r.c.status = s.c.status = 0\}$

False-positive Difference: Reports tuples in R that currently do not exist in S but may exist after re-evaluating the outdated values in both relations, i.e., the unmatched depends on outdated values that may match when re-evaluated.

$R -_+ S = \{r \in R \mid \exists s \in S \text{ where } r.c.value = s.c.value \ \forall \ c \text{ in which } r.c.status = s.c.status = 0, \text{ and } \exists c' \text{ where } r.c'.value \neq s.c'.value \text{ and } (s.c'.status = 1 \text{ or } r.c'.status = 1)\}$

False-negative Difference: Reports tuples in R that currently exist in S but may not exist after re-evaluating the outdated values in both relations, i.e., the matching

depends on outdated values that may not match when re-evaluated.

$R - S = \{r \in R \mid \exists s \in S \text{ where } r.c.value = s.c.value \forall c, \text{ and } \exists c' \text{ where } (s.c'.status = 1 \text{ or } r.c'.status = 1)\}$

Like the standard set difference operator, the extended set difference operators are not commutative.

- **Set Intersection ($R \cap S$):** We define three types of set intersection operators namely, *True Intersection* (\cap_T), *False-positive Intersection* (\cap_+), and *False-negative Intersection* (\cap_-). The algebraic expressions of the set intersection operators are as follow:

True Intersection: Reports tuples that are guaranteed to be in both relations independent of the re-evaluation of the outdated values, i.e., tuples that exist in both relations with all their values up-to-date.

$R \cap_T S = \{r \in R \mid \exists s \in S \text{ where } r.c.value = s.c.value \text{ and } r.c.status = s.c.status = 0 \forall c\}$

False-positive Intersection: Reports tuples in R that currently exist in both relations but may not exist in either of R or S when the outdated values are re-evaluation, i.e., the matching of the two tuples depend on outdated values.

$R \cap_+ S = \{r \in R \mid \exists s \in S \text{ where } r.c.value = s.c.value \forall c, \text{ and } \exists c' \text{ where } (r.c'.status = 1 \text{ or } s.c'.status = 1), \text{ and } r \notin R \cap_T S\}$

False-negative Intersection: Reports tuples in R that do not exist in S but may exist in S when the outdated values are re-evaluation, i.e., the unmatching depends on outdated values.

$R \cap_- S = \{r \in R \mid \exists s \in S \text{ where } r.c.value = s.c.value \forall c \text{ in which } r.c.status = s.c.status = 0, \text{ and } \exists c' \text{ where } r.c'.value \neq s.c'.value \text{ and } (r.c'.status = 1 \text{ or } s.c'.status = 1), \text{ and } r \notin R \cap_T S \text{ and } r \notin R \cap_+ S\}$

The extended *True Intersection* operator is commutative, i.e., $R \cap_T S = S \cap_T R$. In contrast, the *False-positive* and *False-negative* intersection operators are not commutative, i.e., $R \cap_+ S \neq S \cap_+ R$, and $R \cap_- S \neq S \cap_- R$.

- **Outer Join:** We consider the left outer join between relations R and S on join attributes $R.a$ and $S.b$ and join predicate P . We define three types of outer join operators namely, *True Outer Join*, *False-positive Outer Join*, and *False-negative Outer Join*. Each of the *True*, *False-positive*, and *False-negative* outer join operators reports the joined pairs of tuples that are in the corresponding inner join results plus additional sets O_T , O_+ , and O_- , respectively. O_T contains tuples r in R that do not join with any tuples in S and have $r.a$ up-to-date. O_+ contains tuples r in R that do not join with any tuples in S and have $r.a$ outdated. O_- contains tuples r in R that join with tuples in S and have $r.a$ outdated. More formally, O_T , O_+ , and O_- are defined as follows:

$O_T = \{< r, Null >, r \in R \mid r.a.status = 0 \text{ and } P(r.a.value, s.b.value) = False \forall s \in S\}$

$O_+ = \{< r, Null >, r \in R \mid r.a.status = 1 \text{ and } P(r.a.value, s.b.value) = False \forall s \in S\}$

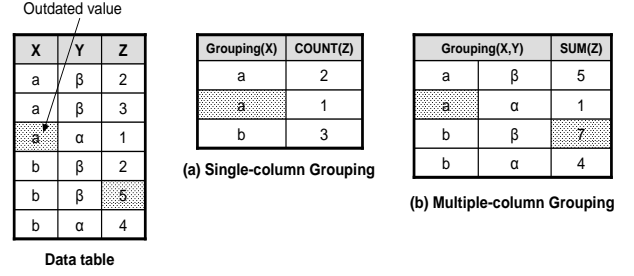


Figure 6: Grouping operators

$O_- = \{< r, Null >, r \in R \mid r.a.status = 1 \text{ and } \exists s \in S \text{ where } P(r.a.value, s.b.value) = True\}$

- **Grouping ($\gamma(R)$):** Tuples that are identical in all the grouping columns, according to the extended semantics of the identical tuples, are added to a single group. In Figure 6, we depict two examples of grouping operations. Notice that value ‘a’ in Figure 6(a) is represented by two groups; one group represents the up-to-date values while the other group represents the outdated values. Since the query results are annotated with the validity status information, i.e., whether values are up-to-date or outdated, users can estimate the effect of the outdated groups (if any) on the other groups. For example, the second reported group in Figure 6(a) may affect either of the first or third groups in the result if the outdated value is re-evaluated to value ‘a’ or ‘b’, respectively. In contrast, the second reported group in Figure 6(b) may only affect the fourth group in the result if the outdated value is re-evaluated to ‘b’. The other two groups will not be affected by the outdated group because they are guaranteed not to match in column Y.

- **Aggregation ($\eta(R)$):** An aggregation function, e.g., SUM, AVG, COUNT, aggregates a set of up-to-date and outdated values and returns a single value. The question is: What will the status of the returned value be? We categorize the aggregation functions into two categories, *value-insensitive* and *value-sensitive* aggregators. The *value-insensitive* aggregators such as COUNT return a value with an up-to-date status (status = 0). Whereas, the *value-sensitive* aggregators such as SUM, AVG, MIN, and MAX return a value with up-to-date status only if all the values in the group are up-to-date, otherwise the returned value will have an outdated status. The reason is that if a single value is suspicious then the outcome of the value-sensitive aggregator will be also suspicious. For example, consider the third and fourth groups in Figures 6 (a) and (b), respectively. The *COUNT()* aggregator returns an up-to-date value because the number of tuples in the group (three in our example) is independent of the outdated value 5. On the other hand, the *SUM()* aggregator returns an outdated value 7 because the sum of the values depends on the outdated value 5.

- **Projection (π):** The projection operator selects the projected attributes along with the status, *up-to-date* or *outdated*, of each database cell in these attributes.

$\pi_{C_1, \dots, C_x}(R) = \{r = < (C_1.value, C_1.status), \dots, (C_x.value, C_x.status) \}$

| | |
|---|-----|
| $\sigma_{?,P}(R) = \sigma_{?,P}(\sigma_P(R))$, where $? \in \{T, +\}$ | (1) |
| $\sigma_{?,P_1}(\sigma_{?,P_2}(R)) = \sigma_{?,P_2}(\sigma_{?,P_1}(R))$, where $?, ?' \in \{T, +, -\}$ | (2) |
| $\sigma_{?,P}(R \bowtie_P S) = R \bowtie_{?,P} S$, where $? \in \{T, +\}$ | (3) |
| $\sigma_{-,P}(R) = \sigma_{+,-P}(R)$ | (4) |
| $\sigma_{-,P_1 \wedge \dots \wedge P_i \wedge \dots \wedge P_k}(R) = \sigma_{-,P_1 \wedge \dots \wedge P_{i-1}}(\sigma_{T,P_i \wedge \dots \wedge P_k}(R))$, If C is zero-outdated column $\forall C \in \text{attributes}$ in P_j , $j \in [i \dots k]$ | (5) |
| $\sigma_{?,P}(R) = \phi$, where $? \in \{+, -\}$, If C is zero-outdated column $\forall C \in \text{attributes in } P$ | (6) |
| $\sigma_{T,P}(\sigma_P(R)) = \sigma_P(R)$ If C is zero-outdated column $\forall C \in \text{attributes in } P$ | (7) |

Table 1: Equivalence rules (Short list)

We extend the standard SQL *Select* statement to include the newly proposed operators. A comparison operator may be suffixed with ‘+’ or ‘-’ to indicate a false-positive or false-negative evaluation, respectively.

Example 4: Consider the following extended SQL *select* statement:

```
Select GSeq, PSeq
From GENE G, PROTEIN P
Where G.GID =+ P.GID And
And GFunction =- 'F2';
```

where the equality operators =, =+, and =- correspond to σ_T , σ_+ , and σ_- , respectively. The select statement is equivalent to the algebraic expression $\pi_{GSeq, PSeq}(\sigma_{-,GFunction='F2'}(GENE \bowtie_{+,G.GID=P.GID} PROTEIN))$.

4.3 Equivalences in the Extended Algebra

Equivalent expressions in the extended algebra are important for query optimization. The query optimizer typically maintains information about the data inside the database such as the number of tuples in a relation, the distribution of the data, and the distinct values in a given column. In the proposed model, the query optimizer maintains additional information about the status of the values in the database. For example, the optimizer keeps track of the number of outdated values in each column. Hence, the optimizer can decide whether a given column is *zero-outdated* (all the values are up-to-date), *all-outdated* (all the values are outdated), *sparse-outdated* (most of the values are up-to-date), or *dense-outdated* (most of the values are outdated). These statistics are useful in estimating the cost of the new operators and in choosing the optimum (near optimum) query plan.

In Table 1, we present a short list of equivalence rules in the extended algebra. The following examples demonstrate several optimization scenarios.

Example 5: The logical expression $\sigma_{T,GFunction='F2'}(GENE)$ selects tuples that evaluate the given predicate to True (2nd tuple in Figure 4). Based on Rule 1, this expression is equivalent to $\sigma_{T,GFunction='F2'}(\sigma_{GFunction='F2'}(GENE))$. The latter expression applies the standard selection operator over table GENE, which returns the 2nd and 3rd tuples, and then

applies the *True Selection* operator over these two tuples, which returns the 2nd tuple only. The cost of these two expressions can be significantly different depending on the storage scheme and the selectivity of the predicates. For example, assume that the status information is stored in a separate table, i.e., not inside the data tables, and the predicate selectivity is high (few tuples have the gene function ‘F2’). In this case, the latter expression can be much cheaper since the standard select will return few tuples (possibly using an index), and then the *True selection* operator is executed over the returned tuples by joining them with the status information and returning the tuples that evaluate the predicate to *True*. In contrast, if the *GFunction* column is a dense-outdated column, then the optimizer may prefer the former expression and executes the *True selection* operator directly by first retrieving the tuples having up-to-date values in the *GFunction* column (few tuples), and then joining them with the data table and returning the tuples that evaluate the predicate to *True*.

Example 6: The logical expression $\sigma_{-,BindingSite='TTAA...'} \text{ And } GDirection='- '(GENE)$ selects the tuples that evaluate the given predicate to *-ve* (4th tuple in Figure 4). One query plan is to execute the *potentially false-negative* selection operator either by directly scanning the GENE table (if the status information is stored inside the table) or by first joining the GENE table with the status information, and then scanning the produced table (if the status information is stored in a separate table). A more efficient plan can be generated using Rule 5 that transforms the expression to $\sigma_{-,BindingSite='TTAA...'}(\sigma_{T,GDirection='- '(GENE))$. Rule 5 makes use of the fact that zero-outdated columns in the predicate have to satisfy the conditions on them, otherwise the tuple cannot evaluate to *-ve*. Applying Rule 1 on the latter expression results in expression $\sigma_{-,BindingSite='TTAA...'}(\sigma_{T,GDirection='- '(GDirection='- '(GENE)))$. According to Rule 7, the *True Selection* operator can be dropped because it has the same predicate as the inner selection and the *GDirection* column is zero-outdated. We end up with expression $\sigma_{-,BindingSite='TTAA...'}(\sigma_{GDirection='- '(GENE))$. As a result, we can first apply the standard selection operator (possibly using an index) over the GENE table to select the tuples satisfying the predicate on *GDirection*, which are the 4th and 6th tuples in Figure 4. Then, we apply the *potentially false-negative* selection operator over these two tuples, which returns the 4th tuple only.

5. INVALIDATION AND REVALIDATION OPERATORS

When a database item is updated, all items that depend on it through real-world activities need to be temporarily invalidated, i.e., marked as *outdated*, until the involved activities are performed and the results are reflected back into the database. On the other hand, when an outdated item is re-evaluated, it needs to be revalidated, i.e., marked as *up-to-date*, as well as all items that depend on it through computable dependencies. The invalidation and revalidation operations are applied recursively over the dependent data items.

In Figure 7, we present three data invalidation and revalidation operators. The *Invalidate()* and *Validate()* operators

| | Invalidate(C) | Validate(C) | Update(C) |
|----------------------------------|---------------------------------|---|---|
| C.Status = 0 (Up-to-date) | C.Status = 1 Invalidate (C*) | ----- | C.Status = 0 - Update (C) - Invalidate (C ^S) |
| C.Status = 1 (Outdated) | ----- | IF (C*.status = 0) THEN - C.status = 0 - Validate (C ⁻) | IF (C*.status = 0) THEN - C.status = 0 - Update (C ⁻) |

$C^- \rightarrow \text{RecomputableClosure}(C)$ $C^S \rightarrow \text{OutdatedClosure}(C)$ $C^* \rightarrow C^- \cup C^S$
 $C^+ \rightarrow \text{InputParameters}(C)$

Figure 7: Data invalidation/revalidation operations

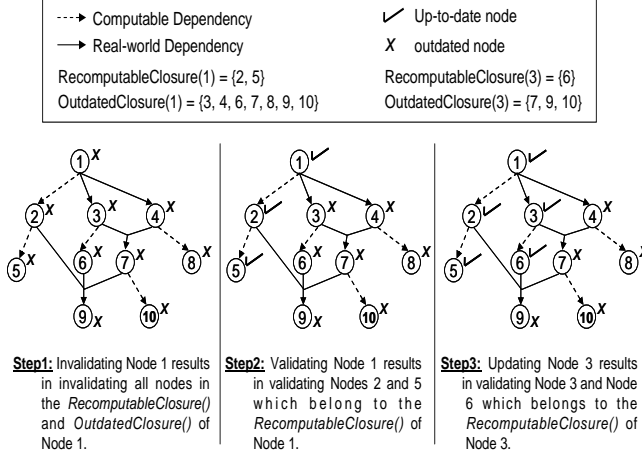


Figure 8: Example of data (in)validation

modify the status of the data without modifying the value, whereas the *Update()* operator modifies both the value and status of the data. The functionality of the operators is expressed using the *Closure()* properties given in Section 3.2.3. The *Invalidate(C)* operator invalidates *C* as well as all items that belong to both the *RecomputableClosure(C)* and *OutdatedClosure(C)*. Figure 8 depicts a set of database items (referred to as Nodes 1 - 10) along with a set of user-defined dependencies. Step 1 in Figure 8 gives an example of invalidating Node 1 in the dependency graph. The *Validate(C)* operator validates *C* only if the parents of *C* in the dependency graph, i.e., *InputParameters(C)* as defined in Section 3.2.3, are all valid. If *C* is validated, then the *Validate()* operator is recursively applied over the *RecomputableClosure(C)* set. Step 2 in Figure 8 gives an example of validating Node 1 in the dependency graph. The *Update(C)* operator changes the value of *C* and may change its status as well. If *C* is already up-to-date, then *C* remains up-to-date. The *RecomputableClosure(C)* is recursively recomputed and updated while the *OutdatedClosure(C)* is recursively invalidated. If *C* is already an invalid cell, then *C* is validated only if the parents of *C* are all valid, otherwise *C* remains invalid. The *RecomputableClosure(C)* is recursively recomputed and updated. Notice that since *C* was already invalid, then all items in the *OutdatedClosure(C)* set are also invalid and they will remain invalid until explicitly validated. Step 3 in Figure 8 gives an example of updating Node 3 in the dependency graph.

```

1 src_list = the set of all source cells involved in the dependency other than that in Ti.Ci;
2 dest_cell = the destination cell of the dependency (cell in column R.C)
3 IF ((dest_cell != null) and (Ci.old != Ci.new)) THEN
4   IF (Ci.Status.old = Ci.Status.new = "up-to-date") THEN
5     IF (status of all cells in src_list is "up-to-date") THEN
6       -- Insert a record into PendingActivity to log a request
7       -- for executing function F with src_list as input;
8     END IF
9     -- Update the status of dest_cell to "outdated";
10  ELSE IF (Ci.Status.old = "outdated" and Ci.Status.new = "up-to-date") THEN
11    IF (status of all cells in src_list is "up-to-date") THEN
12      -- Insert a record into PendingActivity to log a request
13      -- for executing function F using src_list as input;
14    END IF
15  END IF
16 ELSE IF ((dest_cell != null) and (Ci.Status.old != Ci.Status.new)) THEN
17   IF (Ci.Status.old = "up-to-date" and Ci.Status.new = "outdated") THEN
18     -- Delete from PendingActivity any request records for
19     -- executing function F with src_list as input;
20     -- Update the status of dest_cell to "outdated";
21   ELSE IF (Ci.Status.old = "outdated" and Ci.Status.new = "up-to-date") THEN
22     IF (status of all cells in src_list is "up-to-date") THEN
23       -- Insert a record into PendingActivity to log a request
24       -- for executing function F using src_list as input;
25     END IF
26   END IF
27 END IF

```

(a) Template for real-world activity functions

```

1 src_list = the set of all source cells involved in the dependency other than that in Ti.Ci;
2 dest_cell = the destination cell of the dependency (cell in column R.C)
3 IF ((dest_cell != null) and (Ci.old != Ci.new)) THEN
4   IF (Ci.Status.old = "outdated" and Ci.Status.new = "up-to-date") THEN
5     IF (status of all cells in src_list is "up-to-date") THEN
6       -- Update the status of dest_cell to "up-to-date";
7     END IF
8   END IF
9 ELSE IF ((dest_cell != null) and (Ci.Status.old != Ci.Status.new)) THEN
10  IF (Ci.Status.old = "up-to-date" and Ci.Status.new = "outdated") THEN
11    -- Update the status of dest_cell to "outdated";
12  ELSE IF (Ci.Status.old = "outdated" and Ci.Status.new = "up-to-date") THEN
13    IF (status of all cells in src_list is "up-to-date") THEN
14      -- Update the status of dest_cell to "up-to-date";
15    END IF
16  END IF
17 END IF

```

(b) Template for computable functions

Figure 9: Templates for *Add Dependency* construct referencing source columns $T_1.C_1, T_2.C_2, \dots, T_n.C_n$, destination column $R.C$, and function F .

6. DESIGN ISSUES

One of the DBMS tasks is to decide which dependencies to activate due to a modification in the database, i.e., when the value or status of a database cell is updated, the DBMS needs to trigger specific dependencies to update the value (or status) of other database cells. Moreover, the user-defined dependencies create dependency graphs among the database items. A key challenge is how to maintain and manipulate these dependency graphs. One approach is to materialize the dependency graphs inside the database. Another approach is to execute and track the dependencies using database triggers. The proposed system adopts the latter approach because of its scalability. In the following we discuss both alternatives and highlight their pros and cons.

6.1 Materialization of Dependencies

One approach for maintaining the user-defined dependen-

cies is to store these dependencies inside the database using tables separate from the data tables. That is, for every dependency instance, we store the source cells, the destination cell, and the function involved in the dependency in some relational schema. A database cell is uniquely identified by the triplet table name, column name, and tuple id. Since the *Add Dependency* construct may implicitly define multiple dependency instances at once (Refer to Section 3.1), we need to store each of these instances independently in the database. Then, when the value or status of a database cell is modified, we need to query the stored dependencies to find the dependent cells that need to be invalidated. Each query retrieves only the immediate descendants in the dependency graph. Therefore, we need to repeatedly query the dependencies until all descendants are reached. This repetitive querying corresponds to computing the *RecomputableClosure()* and *OutdatedClosure()* of the initially modified cell (Refer to Figure 7 for expressing the (in)validation operations in terms of the *Closure()* sets). The main disadvantages of this approach are that (1) the number and size of the stored dependencies can be significantly large especially because we store each individual dependency instance in the database, and (2) the repetitive querying over the stored dependencies can be expensive.

6.2 Realization of Dependencies using Triggers

In this approach, we utilize the fact that users express the dependencies among the data items using predicates. Therefore, by parsing the *Add Dependency* constructs, the DBMS can automatically generate code snippets over the source and destination tables to enforce the defined dependencies in the database. For each *Add Dependency* construct, one code snippet is generated in the *After Update* trigger over each of the source columns referenced in the dependency. The purpose of this code snippet is to propagate any modification in the value or status of the source cells to the status of the destination cell. Another code snippet is generated in the *Before Insert* trigger over the destination table of the dependency. The purpose of this code snippet is to check the status of the source cells, and if any is outdated, then the newly inserted destination cell will also be outdated. Recall that we establish the link between the source cells and the destination cell using the predicates supplied by the user in the *Add Dependency* construct. In this section, we focus on the *After Update* code snippets since they are more complex compared to the *Before Insert* code snippets.

In Figure 9, we present the *After Update* code templates. If the involved function is of type *real-world activity*, then the template in Figure 9(a) is used, otherwise the template in Figure 9(b) is used. These templates generate code snippets over each source column $T_i.C_i$ as part of the *After Update* trigger on table T_i . Since we are processing a specific tuple in T_i inside the trigger body, then the code snippet references a specific dependency instance in each execution.

Before discussing the details of the code templates, we need to present a storage scheme for the data model. Each cell in the database maintains, in addition to the cell value, a *status* flag which indicates whether the cell is up-to-date or outdated. Several storage schemes and optimizations are possible depending on whether the *status* information is stored inside the data tables or in separate tables. For simplicity, we assume that the *status* information is stored

inside the data tables. That is, every column C in table R will have an additional column C_Status that stores the *status* of C . The *status* attributes are system-maintained attributes that will be updated only by the database system. Initially, the *status* field of each cell in the database is set to zero, i.e., is up-to-date.

Consider the template in Figure 9(a). Line 1 retrieves all source cells involved in the dependency other than the current cell in column C_i . Line 2 retrieves the destination cell of the dependency. These two steps are performed by forming *select* statements according to the predicates supplied in the *Add Dependency* construct as well as the current value of C_i . If the value of C_i is updated while its status remains up-to-date and all other source cells in the dependency are also up-to-date, then a request record is inserted into the *PendingActivity* log (See Section 6.3) for performing the involved real-world activity (Line 6). If any source cell is outdated, then the request records is not created because the real-world function can be performed only when all its input parameters are up-to-date. Processing the request records in the *PendingActivity* log is discussed in Section 6.3. In either case, the destination cell of the dependency is marked *outdated* (Line 8). If the value of C_i is updated while its status remains outdated, then nothing needs to be done because the destination cell of the dependency should be already outdated. If the value of C_i is updated and its status is also updated from *outdated* to *up-to-date*, then a request record for performing the real-world activity is created only if all other source cells are up-to-date (Line 11). Notice here that the destination cell of the dependency should be already outdated because C_i was outdated. The destination cell remains outdated until the real-world activity is performed. The scenario in which the value of C_i is updated while the status is modified *up-to-date* to *outdated* is not a valid scenario. The reason is that the status attributes are system-maintained attributes, and hence there are scenarios for updating both the value and status that cannot occur.

The second part of the template (Lines 14-23) handles a change in the status of C_i without changing its value. If the status of C_i is modified from *up-to-date* to *outdated*, then any request record in *PendingActivity* for executing the involved function over C_i is deleted (Line 16). This is because the function can no longer be performed until C_i is revalidated. Moreover, the destination cell of the dependency is marked *outdated* (Line 17). If the status of C_i is modified from *outdated* to *up-to-date*, then a request record for performing the real-world activity is inserted into *PendingActivity* only if all other source cells are up-to-date (Line 20).

The template in Figure 9(b) is a simplified version of that in Figure 9(a). It is important to highlight that the template of the computable functions does not call (execute) the function involved in the dependency. We assume that the function call is handled by the database developer in the typical way. For example, if the value of a source cell in the dependency is updated, then we assume that there is a code written by the database developer that calls the involved function and updates the value of the destination cell of the dependency. The template only modifies the status of the destination cell according to the changes in the value or status of the source cells.

Example 7: Repeating the dependency from Example 2, where the protein sequences corresponding to genes with

function ‘F1’ are inferred using prediction tool *A-Prediction*. The *Add Dependency* construct is:

```

ADD Dependency Using A-Prediction
  Source Gene.GSeq, Gene.GDirection
  Destination Protein.PSeq
  Where Protein.GID = Gene.GID
  And Gene.GFunction = ‘F1’

```

The above construct creates a code snippets over each of the *Gene.GSeq* and *Gene.GDirection* columns. Considering the code snippet over the *GSeq* column (*GSeq* replaces C_i in the template in Figure 9(a)), the main lines to be replaced in the template are presented below.

```

Src_list = {New.GDirection:New.GDirection_Status} (Template Line 1)

Dest_cell = Select Pseq From PROTEIN (Template Line 2)
  Where Protein.GID = New.GID
  AND New.GFunction = ‘F1’;

Update statement = Update PROTEIN (Template Lines 8, 17)
  Set PSeq_Status = 1
  Where Protein.GID = New.GID
  AND New.GFunction = ‘F1’;

```

Since the *GDirection* and *GFunction* columns are in the same table as *Gseq*, their values are selected using the *NEW* variable inside the trigger. *Src_list* contains attribute *GDirection* and its status *GDirection_Status*. *Dest_cell* is set to the output of the depicted *select* statement. The predicates of the *select* statement are the same as the predicates supplied in the construct with the exception of replacing references to table *GENE* with the *NEW* variable to refer to the tuple at hand. The same concept applies when we update the status of the destination cell. Notice that if the tuple at hand corresponds to a gene with function different from ‘F1’, then the destination cell will be empty and the code snippet will be skipped. This is an expected behavior because in this case the tuple is irrelevant to the dependency.

The advantages of realizing dependencies using triggers are that (1) we do not need to explicitly store the user-defined dependency instances inside the database, (2) the recursive search to reach all dependent data items, i.e., computing the closures, is implemented as update propagation using the triggering mechanism, and (3) the code snippets are generated per source column in the *Add Dependency* construct independent of the number of dependency instances that are implicitly defined by the construct, i.e., the number of tuples that satisfy the construct predicates. One scalability issue is that in some application domains, the number of automatically generated triggers over a single table in the database can be very large depending on the number of user-defined dependencies. Several advanced activation and indexing techniques for firing triggers have been proposed to address this scalability issue and to efficiently select which triggers to fire. Examples of these techniques include the indexing and memory-caching mechanisms proposed in [9] and the marking scheme adopted in the POSTGRES rule system [14]. One of the disadvantages of realizing dependencies using triggers is that manipulating the individual dependency instances is no longer straightforward. For example, overriding one dependency instance by a new instance due to a change in the data derivations (Refer to Section 3.2.4) means that the existing code snippet that corresponds to the old dependency needs to be disabled over specific data

| LogId | Timestamp | FunctionName | PendingUpdate |
|-------|-----------|--------------|---|
| 1 | t_i | A-Prediction | Update PROTEIN Set PSeq = A-Prediction('TTCT...'; '4') PSeq_Status = 0 Where PROTEIN.GID = (Select P.GID From PROTEIN P, GENE G Where P.GID = G.GID AND G.GFunction = ‘F1’ AND G.GID = ‘JW0013’); |
| ... | ... | ... | ... |

Replaced with the function result at execution time

Figure 10: Structure of the *PendingActivity* log

items. In Section 7.1, we present simple extensions to the proposed model to overcome this challenge.

6.3 Logging and Resuming Pending Activities

When a function F of type *real-world activity* has all its source parameters up-to-date but its destination parameter is outdated, then a request record is created for F and inserted into *PendingActivity* (Refer to Lines 6, 11, and 20 in Figure 9(a)). If any of the source parameters of F is outdated, then F is no longer ready for execution and its record is deleted from *PendingActivity* (Refer to Line 16 in Figure 9(a)). The structure of a request record is illustrated in Figure 10. *PendingUpdate* is an *update* statement that will be executed when the involved real-world activity is performed and its result is supplied back to the system. Continuing with the dependency defined in Example 7, if the sequence of gene ‘JW0013’ is updated to be ‘TTCT...’, then the code snippet over column *GENE.GSeq* will create the record shown in Figure 10. The *PendingUpdate* statement updates the destination table of the dependency by setting the destination column to the result of the function involved in the dependency. Moreover, the status of the destination column will become up-to-date. The function call contains the arguments that should be passed to the real-world activity when performed. This function call will be replaced with the function result at execution time.

When the real-world activity involved in the dependency is performed and its output result is available, the result is passed to the database system using the following command:

```

Resume Function <func_name>
  Value <func_output>
  References <logId>;

```

where *logId* references the *LogId* column in *PendingActivity* (1st column in Figure 10). This command executes the update statement in the *PendingUpdate* column (4th column in Figure 10) after replacing the function call with the function output value <func_output>. Then, the request record is deleted from *PendingActivity*. The execution of the update statement may cause more records to be inserted into the *PendingActivity* log.

There are considerations that need to be taken into account when processing the *PendingActivity* log. Since real-world activities may take long time to perform, a request record may be deleted from *PendingActivity* while its function is being conducted externally. The record may be deleted either because one of the source parameters of the involved function is outdated (Line 16 in Figure 9(a)) or because multiple updates took place over the source parameters and hence the request record of the most recent update operation overwrites (deletes) the previous one (The overwriting operation is implicitly performed in Line 6 in

Figure 9(a)). In either case, when the *Resume Function* command is executed, it will not find the request record referenced by *logId*. This behavior is acceptable because in both cases it is too late to pass the output result of the pending execution to the database.

7. MODEL EXTENSIONS

7.1 Dependencies Overriding

New dependency instances may override existing ones due to changes in the derivations of the underlying data (Refer to Section 3.2.4 for an example). In the case of materializing the dependency graphs inside the database (Section 6.1), overriding one dependency instance, say DI_1 , by another instance, say DI_2 , is a matter of deleting DI_1 and inserting DI_2 . In contrast, the overriding mechanism in the case of realizing the dependencies using triggers (Section 6.2) is not straightforward because dependencies are enforced by code snippets generated automatically inside the database. We need a mechanism by which we can disable the execution of the code snippet corresponding to the overridden dependency over specific destination cells. Using the *Alter Table* command to disable a trigger is not applicable because we need to disable the execution over specific destination cells.

The proposed system enables or disables the code snippet execution by explicitly adding predicates in the trigger body that depend on version numbers assigned automatically to the user-defined dependencies. More specifically, we extend the *Add Dependency* construct, the data model, and the code snippet templates in the following way. Each *Add Dependency* construct is automatically assigned a version number that is always increasing, e.g., a sequence number. This version number defines precedence among the dependencies, i.e., a dependency with a larger version number (defined more recently) overrides other dependencies that share the same destination parameter and have lower version numbers. The data model is extended such that each database cell maintains, in addition to the cell value and status, a version number that corresponds to the most recent dependency that has this cell as the destination parameter. In the storage scheme, this extension maps to an additional column $C_Version$ that resides in the data tables for each user column C . The last extension concerns the code snippet templates presented in Figure 9. Line 2 in both templates retrieves, in addition to the destination cell, the version number attached to the destination cell. Then, the remaining segment of each template (Lines 3-23 in Figure 9(a) or Lines 3-17 in Figure 9(b)) is encapsulated inside the following *IF* condition:

```

IF (dest_cell_version = v_n) THEN
    ...
END IF

```

where v_n is the version number assigned to the *Add Dependency* construct. The *IF* statement above simply disables the execution of the code snippet if the version of the destination cell does not match the version number of the dependency that generated the code snippet. If the version numbers match, then the dependency that generated the code snippet is not overridden by more recent dependencies with respect to the given destination cell, and hence the code snippet will be executed. The same *IF* condition will encapsulate the *Before Insert* code snippets that are generated over the destination table of the dependency. This

| Add Dependency Construct | Depend_ version | Function_ Name | Dest_ Table | Dest_ Column | Dest-to-Src Query |
|--|-----------------|----------------|-------------|--------------|--|
| Add Dependency Using A-Prediction Source Gene.GSeq, Gene.GDirection Destination Protein.PSeq Where Protein.GID = Gene.GID And Gene.GFunction = 'F1'; | v_n | A-Prediction | Protein | PSeq | Select GSeq, GSeq_Status, GDirection, GDirection_Status From Protein, Gene Where Protein.GID = Gene.GID And Gene.GFunction = 'F1'; |
| ... | ... | ... | ... | ... | ... |

Replaced with an instance value at execution time

Figure 11: Structure of *BackwardTraversal* table

ensures that for a given destination cell, the correct source cells will be retrieved.

7.2 Curation Operators

The *PendingActivity* log presented in Section 6.3 enables users to systematically revalidate the data inside the database. However, it does not directly support rich operations over the outdated data. Examples of these operations include: (1) Given a query result that contains both up-to-date and outdated data, what are the outdated values in the result that can be instantly revalidated (outdated values that do not depend on any other outdated values)? and (2) Given an outdated database cell that is of interest to be revalidated, what are the steps to follow in order to revalidate the given cell (the sequence of the outdated ancestors of the given cell to be revalidated first)? In this section, we propose two curation operators that support the functionalities described above.

- ***OutdatedRoots()* [*OutdatedRoots(R)*]:** The *OutdatedRoots()* operator reports the set of outdated database cells that do not have any outdated ancestors in the dependency graph, i.e., database cells that can be instantly revalidated. The *OutdatedRoots(R)* operator takes a relation R as an input and reports the outdated roots in R . Relation R can be an output of SPJ select statement. The *OutdatedRoots()* operator is implemented by a direct extension to the *PendingActivity* log. Since *PendingActivity* stores records only for functions whose source parameters are all valid, then the target database cells of the *PendingUpdate* statements are the outdated roots (one outdated root for each update statement). To explicitly keep track of the outdated roots, we extend the structure of *PendingActivity* to store the unique identifier of each outdated root by adding three additional attributes that store the table name, column name, and tuple id of each outdated-root cell. These attributes are filled automatically when a new tuple is inserted into *PendingActivity*. The *OutdatedRoots(R)* operator is implemented by searching the *PendingActivity* log for each outdated cell in R . If a cell is found, then it is reported as an outdated root and the corresponding record identifies the function that should be performed in order to revalidate the cell. The only limitation on R is that we need to keep track of the tuple ids and column names of the cells in R , which is feasible since R is a product of a simple SPJ select statement.

- ***PreValidation(c)* [*PreValidation(R)*]:** The *PreValidation(c)* operator takes a database cell c as an input and reports the minimal set of outdated database cells that need to be validated before validating c . The reported set consists of the outdated ancestors of c in the

dependency graph. The $PreValidation(R)$ operator takes a relation R as input and reports the $PreValidation(c)$ set for each outdated cell c in R . The $PreValidation$ operator requires a backward traversal in the dependency graph, i.e., given a destination parameter of a dependency, retrieve the source parameters. In order to support the backward traversal, we generate for each $Add\ Dependency$ construct, a query that retrieves the source parameters of the dependency given its destination parameter. When an $Add\ Dependency$ construct is executed, the DBMS parses the construct and creates a record in the $BackwardTraversal$ table presented in Figure 11. For example, the record in Figure 11 corresponds to the dependency defined in Example 7. The queries stored in the $Dest-to-Src$ column (Last column in Figure 11) reference the destination table of the dependency. At execution time, these references are replaced by specific values from a given tuple instance in the destination table.

The procedure for executing $PreValidation(c)$, where c is defined by table T , column C , and tuple r , is as follows. We search the $BackwardTraversal$ table for a dependency with $version$ number that matches the version number attached to c , i.e., the value of $C_Version$ of tuple r . The input cell c is the destination parameter of the returned dependency D . The $Dest-to-Src$ query of D is executed after replacing any columns that belong to T with the corresponding value from r . If the status of any of the returned source parameters is outdated, then the outdated source parameters are stacked along with the function name referenced in D . Then, the search is repeated over each of the outdated cells. The search stops when no more outdated source parameters are reached. Retrieving the stacked records in a LIFO order will report the outdated ancestors of c in a proper revalidation order along with the function names that should be externally performed to revalidate the outdated values.

8. CONCLUSION AND FUTURE WORK

In this paper, we proposed constructs to support real-world activities in database systems while maintaining the consistency of the data that depend on these activities. The proposed system enables users to register real-world activities into the database system and to express the dependencies (induced by these real-world activities) among the data items. The system keeps track of potentially-invalid data items that await the completion of real-world activities to propagate and update their output results back into the database. We introduced new query operator semantics that alert users when the users' query results contain potentially-invalid or out-of-date data (due to pending real-world activities). The new operators enable query execution either on valid data only, or on both valid and potentially-invalid data. We proposed new mechanisms for data invalidation, revalidation, and curation that enable users to manipulate and operate on potentially-invalid (outdated) data. As part of the future work, we plan to investigate further the query optimization and costing issues highlighted in Section 4.3. We plan to study the tradeoffs between storage overhead and execution times during update and query times while maintaining and activating the user-defined dependencies. The optimization issues include determining which statistics to collect, estimating the cost of the new operators, and picking the optimal (near optimal) query plans.

9. REFERENCES

- [1] D. Bhagwat, L. Chiticariu, W. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.
- [2] P. Buneman, A. P. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD*, 2006.
- [3] 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.
- [4] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB*, pages 471–480, 2001.
- [5] U. Dayal. Active database management systems. *SIGMOD Rec.*, 18(3):150–169, 1989.
- [6] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. *SIGMOD Rec.*, 19(2):204–214, 1990.
- [7] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. In *In CIDR*, pages 412–422, 2007.
- [8] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2):1–48, 2008.
- [9] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *ICDE*, pages 266–275, 1999.
- [10] C. L. L. Bocchi and G. Zavattaro. A calculus for long-running transactions. *Lecture Notes in Computer Science*, 2884:124–138, 2003.
- [11] D. Maier. Theory of relational databases. In *Computer Science Press*, 1983.
- [12] M. K. Mohania, P. R. Krishna, K. V. P. Kumar, K. Karlapalem, and M. W. Vincent. Functional dependency driven auxiliary relation selection for materialized views maintenance. In *COMAD*, pages 37–45, 2005.
- [13] A. D. Sarma, J. Ullman, and J. Widom. Functional dependencies for uncertain relations. Technical Report Technical Report, Stanford University, 2007.
- [14] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of POSTGRES. *TKDE*, 2(1):125–142, 1990.
- [15] J. Ullman. Principles of database and knowledge-base systems. volume 1, 1988.
- [16] W. White, M. Riedewald, J. Gehrke, and A. Demers. What is "next" in event processing? In *PODS*, pages 263–272, 2007.
- [17] J. Widom. The starburst rule system: Language design, implementation, and applications. *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15:1–4, 1992.
- [18] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. *CIDR*, pages 262–276, 2005.
- [19] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.
- [20] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. *SIGMOD Rec.*, 19(2):259–270, 1990.