

Supporting Real-world Activities in Database Management Systems

Mohamed Y. Eltabakh ^{#1}, Walid G. Aref ^{#2}, Ahmed K. Elmagarmid ^{#3}, Yasin N. Silva ^{#4}, Mourad Ouzzani ^{*5}

[#]Computer Science Department, Purdue University
West Lafayette, IN, USA

¹meltabak@cs.purdue.edu

²aref@cs.purdue.edu

³ake@cs.purdue.edu

⁴ysilva@cs.purdue.edu

^{*}Cyber Center, Purdue University
West Lafayette, IN, USA

⁵mourad@cs.purdue.edu

Abstract— The cycle of processing the data in many application domains is complex and may involve real-world activities that are external to the database, e.g., wet-lab experiments, instrument readings, and manual measurements. These real-world activities may take long time to prepare for and to perform, and hence introduce inherently long time delays between the updates in the database. The presence of these long delays between the updates, along with the need for the intermediate results to be instantly available, makes supporting real-world activities in the database engine a challenging task. In this paper, we address these challenges through a system that enables users to reflect their updates immediately into the database while keeping track of the dependent and *potentially invalid* data items until they are re-validated. The proposed system includes: (1) semantics and syntax for interfaces through which users can express the dependencies among data items, (2) new operators to alert users when the returned query results contain potentially invalid or out-of-date data, and to enable evaluating queries on either valid data only, or both valid and potentially invalid data, and (3) mechanisms for data invalidation and revalidation. The proposed system is being realized via extensions to PostgreSQL.

I. INTRODUCTION

In many application domains, e.g., scientific experimentation, the cycle of processing the data to generate new results is complex and may involve sequences of real-world activities that cannot be coded within the database system, e.g., wet-lab experiments, instrument readings, and manual measurements. As illustrated in Figure 1, it is typical in scientific applications to have the following scenario: (1) scientists retrieve values from the database system, (2) perform one or more real-world activities that depend on the retrieved values, and then (3) store the output results from the performed activities back into the database. Current database technologies do not capture the dependencies among the data items that involve real-world activities. Therefore, updating a database value will render all the dependent and derived values invalid until the external activities involved in the dependency, e.g., the wet-lab experiment, are re-executed and the output results are reflected back into the database. Because of this mandated unbounded delay in propagating the updates, parts of the underlying

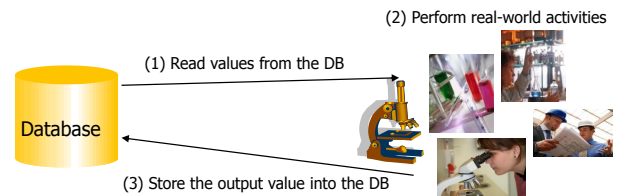


Fig. 1. Processing cycle involving real-world activities

database may remain inconsistent while it still needs to be available for querying.

With current database technologies, scientists may consider one of the following two options: (1) postpone the database updates until all dependent processes are executed and their outputs become consistent, and then reflect these updates into the database at once (may involve unbounded long delays), or (2) reflect the updates immediately into the database and *temporarily* compromise the consistency of the derived data (in this case, users may lose track of the outdated data that need re-evaluation). 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 scientists can reflect the updates immediately into the database 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 manual activities are re-executed and the outdated values are updated.

The theory of functional dependencies (FDs) [4], [8] and its usage in ordinary databases address an orthogonal problem to the problem highlighted in this paper. That is, even with a good schema design of the database and following the decomposition and normalization rules, the inconsistency problem of the derived data still exists. Several extensions to functional dependencies have been proposed to address other issues, e.g., [7], [5]. 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., [6], 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 and hence do not address the problems raised in this paper. Other systems such as active databases, e.g., [10], and uncertain and fuzzy databases, e.g., [3], [9], do not focus on modeling the dependencies that involve real-world activities among the data items, and hence they neither keep track of the temporarily invalid data items nor alert users when their query results involve out-of-date values.

In this paper, we propose a system [1], [2] that enable users to register real-world activities in the database and to create dependencies among the data items. We categorize dependencies into *computable* and *real-world* dependencies. Computable dependencies involve derivation functions that can be executed by the database system, e.g., stored functions in the database, whereas real-world dependencies involve real-world activities that require human intervention, e.g., wet-lab experiment, and hence cannot be coded within the database. The proposed system extends the functionalities of current DBMSs to include:

- **Defining dependencies based on activities:** Functional Dependencies (FDs) capture the dependencies at the conceptual level, i.e., at the schema-design level. In contrast, the proposed system allows users to define real-world activities inside the database and to express the dependencies among the data items using these activities.
- **Tracking outdated data:** To maintain the consistency of the data, the proposed system keeps track of the outdated data items in the database by marking them as *outdated* to indicate that these values are potentially invalid and need re-evaluation. Outdated data can be reported to end-users for verification.
- **Extended querying mechanisms:** We propose extended querying mechanisms that include functionalities such as alerting users of any potentially invalid data items in the returned query results, and evaluating queries on either valid data only and avoid building on any potentially invalid attribute values, or both valid and potentially invalid data.
- **Manipulation and curation mechanisms:** We propose data manipulation mechanisms for invalidating and re-validating the data. We also propose curation mechanisms for recommending the proper execution order of the real-world activities to revalidate the data.

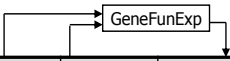
The rest of the paper is organized as follows. In Section II, we formalize the proposed system and present the needed definitions. In Sections III, we introduce the data manipulation and curation operators. In Section IV, we present the extended querying mechanism. The conclusion and implementation remarks are presented in Section V.

II. MODELING OF DEPENDENCIES AND FUNCTIONS

In this section, we present our model and introduce the needed definitions for functions and dependencies.

A. Definitions

Function (F): A function is a general term that refers either to a stored executable function or to a real-world activity. A function takes one or more input parameters and produces one output parameter. Each function has a set of properties that



GID	StartPos	GSeq	GDirection	GFunction
JW0013	5130	TGCT...	+	F1
JW0014	10916	GGTT...	+	F2
JW0015	21112	GGCT...	+	F2
JW0018	31166	CGTT...	-	F4
JW0019	1905	TGTG...	+	F5
JW0012	17404	TTCG	-	F7

GENE table

Fig. 2. Example of real-world dependencies

specify (1) whether or not F is executable, (2) the data types of the inputs, (3) the data type of the output, and (4) the code (the database function name) if F is executable.

For example, referring to Figure 2, the values in the gene function attribute are inferred from both the gene's sequence and direction using a wet-lab experiment *GeneFunExp*. This experiment can be defined in the database as a function with the properties *Not Executable*, $\{text, char\}$, *text*, and *Null*.

It is common that the values stored in one database attribute are produced using a number of possible functions. For example, the values in the gene function attribute can be generated using a group of different experiment types. We capture the fact that any of these related functions can be used to generate the output values using the notion of a *Function Family*.

Function Family (FF): A function family is a set of functions that can be used alternatively to produce the output parameter. All functions in a function family have the same properties.

To specify constraints on the actual dependencies, we introduce the concept of *dependency schemas*. Dependency schemas allow the DBMS to control the permitted set of dependencies that users can define. It is analogous to table schemas that control the structure of the actual inserted tuples.

Dependency Schema (DS): A dependency schema defines the structure of the permitted dependencies among a set of database attributes. A dependency schema is defined as $DS = (S, D, FF, DS-Properties)$, where:

- **S (Source Domain):** An ordered set of database attributes that represent the source domain from which the input parameters will be drawn. The attributes are uniquely identified by the table and column names.
- **D (Destination Range):** A database attribute that represents the destination range from which the output parameter will be drawn. The attribute is uniquely identified by the table and column names.
- **FF:** A function family that represents the set of possible functions that can be used to map from S to D .
- **DS-Properties:** A set of properties of the dependency schema that includes (1) *Overlapping*, and (2) *Cyclic*. *Overlapping* indicates whether or not the destination range can be covered by other dependency schemas. *Cyclic* indicates whether or not this dependency schema can be part of a cycle.

For example, referring to the GENE table in Figure 2, a dependency schema $DS_1 = (S=[GSeq; GDirection], D=[GFunction], FF=\{GeneFunExp1; GeneFunExp2\}, DS-$

Properties={No overlapping; No cycles}) states that attribute *GFunction* depends on the pair of attributes *GSeq* and *GDirection* using one of two possible experiments *GeneFunExp1* or *GeneFunExp2*. Recall that these two experiments have to be previously defined in the database as functions. The “*No overlapping*” property indicates that there is no other means by which *GFunction* can be derived or inferred. As a result, the DBMS will reject any other dependency schema that contains *GFunction* in its destination range. The “*No cycles*” property indicates that DS_1 cannot be part of a cycle. As a result, the DBMS will reject any other dependency schema that forms a cycle with DS_1 , e.g., schema that has *GFunction* in the source domain and *GSeq* in the destination range.

Dependency Instance (DI): A *dependency instance* is an actual 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 $DI = (DS, F, Exec-Properties, Q_S, Q_D)$, where:

- **DS:** The dependency schema from which DI is inherited.
- **F:** The function involved in the dependency such that $F \in DS.FF$.
- **Exec-Properties:** A set of execution properties of *F* that captures the start time, end time, surrounding environment parameters, and runtime setup.
- **Q_S:** An ordered set of predicates over the tables defined in the source domain of DS ($DS.S$). The number of predicates in Q_S matches the number of columns in $DS.S$ where each predicate determines the exact table cell in the corresponding column that will be an input parameter to *F*.
- **Q_D:** A set of predicates over the table defined in the destination range of DS ($DS.D$).

For example, dependency instance $DI_1 = (DS = DS_1, F = GeneFunExp1, Exec-Properties = \{\text{start_time} = t1; \text{end_time} = t2\}, Q_S = [\{GID = JW0015\}; \{GID = JW0015\}], Q_D = \{GID = JW0015\})$ inherits from the dependency schema DS_1 . The first and second predicates in Q_S specify a database cell in each of the *GSeq* and *GDirection* columns (the column names are specified in DS_1 schema) to be the inputs to function *GeneFunExp1* which are ‘GGCT..’ and ‘+’ in the 3rd tuple, respectively (Refer to Figure 2). The predicate in Q_D specifies the database cell in the *GFunction* column (‘F2’ in the 3rd tuple) to be the output from this execution of *GeneFunExp1*. The DBMS enforces automatically the dependency instance once defined. For example, if the sequence of gene ‘JW0015’ is modified (the bold underlined value in Figure 2), the DBMS marks automatically the corresponding gene function as *outdated* (the dotted table cell) until experiment *GeneFunExp1* is re-conducted. In the example above, all the predicates are the same because the columns are in the same table and both the input and output parameters belong to the same tuple. In general, the source and destination columns may belong to different tables and hence the predicates in the dependency instances will be different.

We summarize the process of defining the dependencies in the database as follows: (1) define the *functions* and *function families* that will be referenced in the database, (2) define the *dependency schemas* that capture the attribute-level dependen-

cies and act as skeletons for the dependency instances, and (3) define the *dependency instances* that capture the cell-level dependencies, i.e., the exact input values, the output value, and the execution properties of the involved function.

III. MANIPULATION AND CURATION OPERATORS

Users may invalidate (mark as outdated) or revalidate (mark as up-to-date) the data inside the database. The invalidation and revalidation operations are either explicit, e.g., explicitly invalidating a suspicious piece of data, or implicit, e.g., modifying an input parameter of a real-world activity will implicitly invalidate all dependent data items until the real-world activity is re-executed and its output result is reflected back into the database. The invalidation and revalidation operations apply also for functions. For example, a user may invalidate a faulty experiment run, i.e., invalidates a specific execution of an experiment, which in turn invalidates all data items that depend on that specific experiment run. In all cases, the DBMS keeps track of the cascading among the defined dependencies and propagates the invalidation and revalidation operations accordingly. In this section, we introduce three data manipulation operations over a given database cell *c*.

Invalidate(c): invalidates *c* by marking it as outdated. The DBMS recursively invalidates all data items that depend on *c* either through computable or real-world dependencies. In the case where *c* is already invalid, the *Invalidate(c)* procedure does nothing since all dependent data items are already invalid.

Validate(c): validates *c* by marking it as up-to-date. The target database cell *c* is validated only if all source parameters from which *c* is derived are up-to-date, otherwise the validation is rejected. If *c* is validated successfully, then all data items that are derived from *c* through computable dependencies are recursively validated. In contrast, data items that are derived from *c* through real-world dependencies will remain invalid until the corresponding real-world activities are performed and their output results are updated back into the database.

Update(c): modifies the value of *c* and may change its status as well. In the case where *c* is up-to-date, the modification in *c*’s value results in invalidating all database items *c*’ that are derived from *c* through real-world dependencies, i.e., calling *Invalidate(c)* recursively. Moreover, all database items *c*’ that are derived from *c* through computable dependencies are re-computed and recursively updated, i.e., calling *Update(c)* recursively. If *c* is already outdated, then *update(c)* will validate *c* only if all source parameters from which *c* is derived are up-to-date, otherwise the value of *c* will be modified without changing *c*’s status. Independent of whether *c* will be validated or not, all database items that are derived from *c* through real-world dependencies will not be automatically revalidated since they are waiting on real-world activities to be performed.

In addition to the data manipulation operations, there is a need for data *curation operations* to manage the dependency graphs formed from the user-defined dependencies. These dependency graphs involve both computable and real-world dependencies as illustrated in Figure 3. For example, data item ‘7’ is derived from both data items ‘3’ and ‘4’ through a real-world activity, whereas ‘6’ is derived from ‘3’ through a computable function, e.g., stored function inside the database. In the proposed system, a database item can be an output

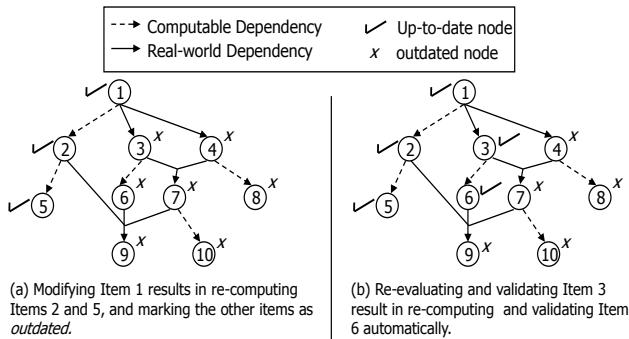


Fig. 3. DAGs generated from the user-defined dependencies

parameter to at most one dependency instance, i.e., a specific value in the database cannot be computed or inferred by two different means. Based on this property, the formed dependency graphs are in fact directed acyclic graphs (DAGs). The order of re-validating the data items inside the database is important. For example, if two tandem real-world activities A and B , where A is feeding its output to B , are invalidated, then B cannot be validated until A is validated. Curation operators support operations such as: (1) Reporting the database items that are ready to be validated, i.e., database items that do not depend on other invalid data items, (2) Reporting a plan (order) by which users can re-validate all the potentially invalid data items in the database, and (3) Reporting the database items that need to be validated before validating a given database item I . We introduce the following curation operators:

- **OutdatedRoots()**: Returns the set of outdated database items that are ready to be validated, e.g., the outdated roots in Figures 3(a) and (b) are the sets $\{‘3’, ‘4’\}$, and $\{‘4’\}$, respectively.

- **PreValidation(I)**: Returns an ordered set (based on the validation order) of outdated database items that need to be validated by the user before validating database item I , e.g., $PreValidation(‘9’)$ in Figure 3(a) returns the ordered set $\{‘3’, ‘4’, ‘7’\}$. Notice that the curation operators need to take into account the type of the involved dependencies, i.e., whether a dependency is computable or real-world. For example, data item ‘6’ is not reported to users as a pre-required validation of data item ‘9’ because ‘6’ will be validated automatically once ‘3’ is validated as illustrated in Figure 3(b).

- **PostValidation(I)**: Returns the set of outdated database items that will be ready for validation once data item I is validated, e.g., $PostValidation(‘4’)$ in Figures 3(a) and (b) returns the sets $\{‘8’\}$, and $\{‘7’, ‘8’\}$, respectively.

IV. QUERYING MECHANISMS

In the proposed model, some values of the underlying database are marked as valid (up-to-date), while the other values are marked as potentially invalid (outdated). Therefore, it is important for the querying mechanism to report back, as part of the query results, the status information of the values in the results. We extend the querying mechanisms to include the following functionalities:

- **Annotating query results with status information**: Query operators are extended to propagate automatically the status information of the data items appearing in the query results, i.e., each value in the query result has an attached flag

indicating whether the value is up-to-date or outdated. For example, referring to Figure 2, the following select statement, "Select GFunction From GENE", returns value ‘F2’ that corresponds to gene ‘JW0015’ marked as outdated while all the other function values marked as up-to-date.

- **Querying only up-to-date data**: We define new query operators that allow users to execute their queries over only the up-to-date data values and exclude the outdated values even if they satisfy the query predicates. For example, the following select statement, "Select * From GENE Where GFunction =@ ‘F2’", returns the information corresponding only to gene ‘JW0014’ (2nd tuple in Figure 2). The ‘=@’ operator is a newly defined equality operator that returns *True* only if the matching value in the database is up-to-date, and *False* otherwise.

- **Querying outdated data**: Some users may prefer conservative query answers that include tuples having the potential to satisfy the query predicates. For example, the following select statement, "Select * From GENE Where GFunction =- ‘F1’", returns the information corresponding to genes ‘JW0013’ and ‘JW0015’ even if the latter gene does not currently satisfy the query. The reason is that the function of gene ‘JW0015’ is under re-evaluation and it may satisfy the query predicate once corrected. The ‘=-’ operator is a newly defined equality operator that return *True* if the value in the database qualifies a given predicate or if the value does not qualify but it is under re-evaluation (outdated).

V. CONCLUSION

Integrating real-world activities into the database engine is a challenging task especially because it affects the consistency of the data. In this paper, we addressed several of these challenges including: (1) Enabling users to define real-world activities in the database and to express dependencies among the data items using these activities, (2) Keeping track of any temporarily outdated data values and reflecting their status over the query results, (3) Proposing new data manipulation and curation mechanisms to support the invalidation and re-validation of data, and (4) Proposing extended querying mechanisms that enable evaluating queries on either up-to-date data only or both up-to-date and outdated data.

REFERENCES

- [1] M. Eltabakh, M. Ouzzani, and W. Aref. bdbms: A database management system for biological data. In *CIDR*, pages 196–206, 2007.
- [2] M. Eltabakh, M. Ouzzani, W. Aref, A. Elmagarmid, Y. Laura-Silva, M. Arshad, D. Salt, and I. Baxter. Managing biological data using bdbms. In *ICDE*, pages 1600–1603, 2008.
- [3] J. Galindo, A. Urrutia, and M. Piattini. Fuzzy databases: Modeling, design, and implementation. *Idea Group Publishing*, 2006.
- [4] D. Maier. Theory of relational databases. In *Comp. Sci. Press*, 1983.
- [5] 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.
- [6] H. Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, 1987.
- [7] A. D. Sarma, J. Ullman, and J. Widom. Functional dependencies for uncertain relations. Technical Report Technical Report, Stanford University, 2007.
- [8] J. Ullman. Principles of database and knowledge-base systems. In *Comp. Sci. Press*, volume 1, 1988.
- [9] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. *CIDR*, pages 262–276, 2005.
- [10] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996.