HandsOn DB: Managing Data Dependencies involving Human Actions

Mohamed Eltabakh\textsuperscript{1}, Walid Aref\textsuperscript{2}, Ahmed Elmagarmid\textsuperscript{3}, Mourad Ouzzani\textsuperscript{3}

\textsuperscript{1} Worcester Polytechnic Institute, MA, USA, meltabakh@cs.wpi.edu
\textsuperscript{2} Purdue University, IN, USA, aref@cs.wpi.edu
\textsuperscript{3} Qatar Computing Research Institute, Qatar, \{aelmagarmid, mouzzani\}@qf.org.qa

Abstract—Consider two values, \( x \) and \( y \), in the database, where \( y = F(x) \). To maintain the consistency of the data, whenever \( x \) changes, \( F \) needs to be executed to re-compute \( y \) and update its value in the database. This is straightforward in the case where \( F \) can be executed by the DBMS, e.g., SQL or C function. In this paper, we address the more challenging case where \( F \) is a human action, e.g., conducting a wet-lab experiment, taking manual measurements, or collecting instrument readings. In this case, when \( x \) changes, \( y \) remains invalid (inconsistent with the current value of \( x \)) until the human action involved in the derivation is performed and its output result is reflected into the database. Many application domains, e.g., scientific applications in biology, chemistry, and physics, contain multiple such derivations and dependencies that involve human actions. In this paper, we propose HandsOn DB, a prototype database engine for managing dependencies that involve human actions while maintaining the consistency of the derived data. HandsOn DB includes the following features: (1) semantics and syntax for interfaces through which users can register human activities into the database and express the dependencies among the data items on these activities, (2) mechanisms for invalidating and revalidating the derived data, and (3) new operator semantics that alert users when the returned query results contain potentially invalid data, and enable evaluating queries on either valid data only, or both valid and potentially invalid data. Performance results are presented that study the overheads associated with these features and demonstrate the feasibility and practicality in realizing HandsOn DB.

Index Terms—Scientific databases, data dependency and consistency, human actions, query processing.

1 INTRODUCTION

In many application domains such as scientific experimentation in biology, chemistry, and physics, the derivations among the data items are complex and may involve sequences of human actions, e.g., conducting a wet-lab experiment, taking manual measurements, and collecting instrument readings. In traditional derived data that are stored inside the database, 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. In contrast, when the derivations among the data items involve human actions, these derivations cannot be coded within the database. Hence, updating a database value may render all dependent and derived values invalid until the required human actions are performed and their output results are updated back in the database.

Typical databases may contain multiple dependencies which may cascade and interleave with other dependencies that involve executable functions, e.g., SQL and C functions. Hence, a complex dependency graph is created among the database items. Since human actions may take long time to prepare for and perform, parts of the underlying database may remain inconsistent for long periods of time while the data still need to be made available for querying. Our focus in this paper is on managing dependencies that involve human actions or more generally, real-world activities, inside the database engine while maintaining the consistency of the derived data under update and query operations.

Motivating Examples: Figure 1 illustrates an example, from the biology domain, of a pipeline collecting different pieces of information about genes/proteins and storing them in the database. As depicted in the figure, the initial sequence files stored in the database will be used as input to a set of procedures involving human actions in order to discover more information, e.g., the protein family, gene function, and the location of SNP (Single-Nucleotide Polymorphism). If the underlying sequence data is modified due to correction of sequencing errors or an improved assembly, then the corresponding output data from the procedures become potentially invalid and need to be re-verified. Another example of chemical reactions is illustrated in Figure 2 where chemists may store in the database descriptions of chemical reactions, e.g., substrates, reaction parameters, instruments settings, and products. Clearly, these chemical reactions require human intervention. If, for example, any of the substrates in the reaction are modified, then the products of the reaction may change as well, and hence they become invalid until the reaction is re-executed or the chemist verifies the old value.

The presence of potentially-invalid values in the database directly affects the correctness of the queries’ answers as well as any decisions based on the results. For example, continuing with the biology pipeline (Figure 1) assume the database instance shown in Figure 3 where
the sequence of gene JW0014 has been updated, and hence the dependent values become invalid (marked as black table cells). Although the reported result from query Q1 seems correct, it is missing crucial information, e.g., the reported value “14457” is potentially invalid and needs to be verified, and the first tuple in the answer matched the query predicate (PredictedFunction = “F2”) based on an invalid value F2, and hence its presence in the output is questionable.

Candidate Solutions with Current Technologies:

While there are several existing technologies that may be used to solve our problem, all have limitations as compared to HandsOn DB. We discuss below three of these possible solutions.

The first possible approach is to store metadata information along with the data values, e.g., a version number or timestamp, to track whether values are up-to-date or outdated. However, this approach has several limitations including: (1) Without explicitly modeling the dependencies inside the DBMS, the maintenance of the auxiliary metadata is delegated to end-users which is both an overwhelming task and error prone. (2) Integrating the metadata information inside the query engine is problematic because users’ queries have to incorporate the metadata information in their evaluation. For example, dependencies may span multiple tables, e.g., values in table R depend on values in tables S_i and S_j, and hence a query on only table R needs to be extended to also check tables S_i and S_j to decide whether R’s values are up-to-date or outdated, which will make even simple queries very complex. (3) All curation operations, e.g., why certain values are invalid and how to re-validate them, and which external activities need to be performed and using which parameters, will be manually performed without any system support.

Another possible approach is to delay the updates to the database until all derived values are computed, i.e., keeping the partial results outside the database. Although feasible in some scenarios involving simple dependencies, it has several limitations as it does not scale with complex dependencies, and storing the data outside the database is not a preferred solution w.r.t. recovery, instant availability of results, and ensuring the consistency of the new values with other database values.

A third approach is in the context of workflow management, e.g., [27], [28], [29], where the external activities, dependencies, and the database updates are modeled as processes of a workflow. While attractive, this approach has two major limitations: (1) Since the dependencies are modeled at the workflow management level (i.e., outside the DB), directly querying the database without consulting the workflow system will not reflect such dependencies, and hence may reveal potentially-invalid values to end-users without notification, and (2) The advanced optimizations and features offered by HandsOn DB, such as extended querying capabilities and curation operators, will no longer be applicable because the external dependencies are not coherently integrated inside the database system. Supporting such optimizations and features at the workflow level will be much harder.

Contribution: The above limitations motivate the need for a more systemic mechanism and an end-to-end solution that enables scientists to focus on running their experiments and uploading the results, instead of tracking the dependencies among the data items and verifying their consistency. In this paper, we propose HandsOn DB, a prototype database engine for managing dependencies that involve real-world activities while maintaining the consistency of the derived data. HandsOn DB addresses all of the above limitations as it enables users to reflect the updates immediately into the database, i.e., instant availability of the data, while the DBMS keeps track of the derived data by marking them as potentially invalid (aka outdated) and reflecting their status in the query results, i.e., the consistency of the data is not compromised. HandsOn DB introduces extended query operators for evaluating users’ queries on either valid data only, thus avoiding relying on any potentially invalid values (no false-positive tuples), or both valid and potentially invalid data (include false-positive tuples). HandsOn DB is a component in bdms [1], [2], our proposed database system for scientific data management. We first highlighted the main research challenges involved in manag-
ing complex real-world dependencies in [3]. In this paper, we propose novel solutions to these challenges.

The contributions of this paper are as follows:

- Proposing new SQL syntax and its corresponding semantics to register real-world activities into the database and to express the dependencies among data items using these activities.
- Introducing an extended relational algebra and new semantics for query operators to alert users of any potentially-invalid data in the query results, and to enable querying either valid data only or both valid and potentially-invalid data.
- Proposing new data manipulation and curation operations for invalidating and revalidating the data, and for keeping track of the real-world activities that need to be performed in order to revalidate the data.
- Experimentally evaluation the proposed features of HandsOn DB and demonstrating the system’s practicality.

The rest of the paper is organized as follows. Section 2 overviews the related work. Section 3 presents the needed definitions. Sections 4 and 5 introduce the data manipulation operations as well as the new query operators, respectively. In Section 6, we present several design issues. The performance analysis is presented in Section 7. Section 8 contains concluding remarks.

## 2 RELATED WORK

The theory of functional dependencies (FDs) in DBMSs, e.g., [5], [6], [7], is used to model dependencies among data items, infer keys, and systematically normalize database schemas to prevent several inconsistency problems, e.g., redundancy, and update and delete anomalies. However, FDs cannot solve the inconsistency problem raised in this paper mainly because the dependencies that involve real-world activities cannot be modeled or coded inside the database (E.g., using triggers or user-defined functions) regardless of how well the schema is designed or normalized. The existence of such external activities that cannot be handled by DBMSs triggered the research in long-running transactions, e.g., [8], [9], where a database transaction may involve external activities, e.g., getting a manager’s signature to complete a purchase transaction. Systems for long-running transactions took the approach of loosening the ACID properties, using optimistic concurrency control techniques, and using compensating transactions in the case of failures. The key objective of these systems is to keep track of the derived data that are already modified by the transaction to roll them back if needed (compensating transactions). However, long-running transactions do not keep track of the derived data that are still awaiting to be updated and are currently inconsistent, and hence they delegate this inconsistency issue to the end-user without any system support. More importantly, the currently inconsistent values are subject to querying—possibly for long periods of time—without any special query processing or notification mechanisms.

Active databases [10], [11], [12] provide mechanisms (through triggers) to respond automatically to events taking place either inside or outside the database. The outside events are ultimately mapped to operations that the DBMS can capture, e.g., insertion, deletion, or calling of a user-defined function. Unlike active databases, in HandsOn DB, a change inside the database, e.g., updating the gene sequence in Figure 3, may trigger the execution of a real-world activity outside the database to update the derived data. Until this activity is performed, the system needs to keep track of all potentially invalid data items, reflect their status over query results, and provide mechanisms for re-validating these invalid items. Active databases do not address these challenges.

Multi-version systems and databases, e.g., [13], [14], [15], maintain the old and new values of updated data. However, like snapshot databases (which is the focus of this paper), multi-version databases model only the computable dependencies among the data items. And hence, the provided isolation level and consistency degree are based on the traditional notion of transactions. For example, if value \( v_j \) depends on value \( v_i \) through an external activity, then a transaction updating \( v_i \) to \( v_i' \) would create a newer version of the database where both \( v_i' \) and \( v_j \) are viewed as consistent and up-to-date values. Although this is correct with the traditional notion of transactions, it is semantically incorrect because the external dependency is not taken into account. Extending the proposed techniques in the context multi-version databases is left as future work, and in this case, the history of dependencies changing over time can be also maintained.

Some systems such as checkout/checkin systems [16] consider querying an old consistent version of the data while updating an off-line version until all required changes to all dependent data items have been performed. Then, the off-line version is released as the newer consistent version. The drawbacks of this approach include violating the need for making users’ updates available as early as possible, hiding possible data corrections for unbounded long delays, and resolving any consistency issues outside the DBMS, i.e., data conflicts are resolved at the checkin time using version-control systems outside the DBMS. HandsOn DB resolves all these issues within the database system.

Probabilistic and fuzzy database systems (PDBMSs), e.g., [17], [18], [19], [20], overlap with HandsOn DB assuming that data invalidation introduces uncertainty to the data, e.g., potentially invalid values can be viewed as unknown values. However, the focus of the two systems is different. In HandsOn DB a change in the state of a database value, i.e., being valid or invalid, is triggered by database operations, while in PDBMSs the uncertainty is inherent to the data and is given as external input. These uncertainties do not change over time unless users
manually modify them. Therefore, PDBMSs do not address several challenges, that are the core of HandsOn DB, such as modeling the dependencies among the data items, keeping track of when and why a data item becomes uncertain (invalid), and keeping track of how to revalidate a data item to become certain (valid).

Provenance management, e.g., [20], [21], [22], [23], follows two main approaches; inversion-based and annotation-based. Inversion-based techniques are not applicable to the problem at hand since we deal with external activities that cannot be executed by the DBMS in the first place. Annotation-based techniques [21], [24], [25], [26] lack the ability of modeling and integrating real-world activities inside the database system, and hence the dependency graph involving these external activities cannot be constructed. Annotations, therefore, can neither maintain the consistency of the derived data items (computable or non-computable) nor keep track of pending activities that need to be performed to revalidate the data. Provenance has been also studied extensively in the context of scientific workflows, e.g., [27], [28], [29], [30] where workflow systems are instrumented to capture and store the provenance information. In these systems, a database can be a single component within a bigger workflow. However, as discussed in Section 1, modeling the external dependencies at the workflow management level has several limitations and does not actually guarantee the desired consistency level. Nevertheless, HandsOn DB can still be used in conjunction with workflow systems to achieve stronger consistency of the data.

3 Modeling Activities & Dependencies

In this section, we present the formal definitions of activities and dependencies, and show how to model them inside the database. We assume a relational database model, and we use the term "database cell" to refer to an attribute value in a single tuple. The value itself can be primitive, e.g., integer or string, or complex, e.g., arrays or bit maps.

**Definition (Real-world Activity):** A real-world activity (RWA, for short) is an activity that requires human intervention, and hence cannot be executed by the DBMS. RWA takes one or more input parameters and produces one or more output parameters.

Since real-world activities are very close in definition to functions, we define the concept of a real-world activity function that maps a real-world activity to a nondeterministic function inside the database. RWA functions are nondeterministic since RWAs such as lab experiments may not generate the same exact output given the same input parameters.

**Definition (Real-world Activity Function):** A real-world activity function RWA-F is a nondeterministic function inside the database that represents a real-world activity. RWA-F is of type ‘real-world activity’ and has a signature that specifies the function name and the input and output types. RWA-F has no associated code.

Similar to defining SQL, C, or Java functions inside the database, we extend the SQL Create Function command to define real-world activity functions as follows:

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

Once real-world activity functions are defined in the database, users can create dependencies among the data items using these functions. Users can also create dependencies among the data items using executable functions, e.g., SQL or C functions. Each dependency defines the function name involved in the dependency, the input parameters to the function (the order of the inputs matters only if the function is executable), and the output parameters from the function.

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

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

If F is of type real-world activity, then DI is called real-world dependency, otherwise, DI is called computable dependency. The dependency of DP on SP is complete in the sense that each database cell in DP depends on all database cells in SP.

Dependency instances are conceptually defined at the cell level, i.e., they capture the dependencies between the database table cells. Such fine-granular level of expressing the dependencies may sometimes involve high overhead as reported in [35]. In HandsOn DB, we introduce a higher level of abstraction using the Add Dependency construct, by which users may define dependencies over one cell, multiple cells, or even entire columns at once. Dependencies are created in (or dropped from) the database using the Add Dependency (or Drop Dependency) constructs that are augmented to the SQL Create Table and Alter Table commands as follows:

```
Create Table <R>
{
   <columns definitions>
}

Add Dependency <dependency_id>
Using <func_name>
Source <T1.c1[, T2.c2, ...]> 
Destination <R.c1[, R.c2, ...]> 
[Where <predicates>]
);

Alter Table <R>
Add Dependency [dependency_id]
Using <func_name>
Source <T1.c1[, T2.c2, ...]> 
Destination <R.c1[, R.c2, ...]> 
[Where <predicates>]
[Invalidate Destination] ;

Alter Table <R>
Drop Dependency <dependency_id>
[Invalidate Destination] ;
```

A new dependency is defined over Table R that contains the destination attributes R.c1, R.c2, ..., of the dependency. The dependency_id is a unique id that is either defined by the user or generated automatically by the system. If the dependency applies to multiple destination tables, then it is defined over each of these tables with different dependency_id. The optional Where clause contains join and selection predicates over the source and destination tables to specify the exact table.
cells that are linked together. Examples 1 and 2 below illustrate defining dependencies over single and multiple tables.

**Example 1: Single-table dependency**

Create Table Gene(
  GID text primary key,
  GSeq text,
  GDirection char,
  GFunctExp text,
  ...
)

ADD Dependency Using GeneFunExp
Source GSeq, GDirection
Destination GFunctExp;

Description: Each gene function is inferred from the corresponding gene’s sequence and direction using GeneFunExp.

**Example 2: Cross-table dependency**

Create Table Protein(
  PID text primary key,
  GID text references Gene(GID),
  PSeq text,
  PFunction 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';

Description: For proteins whose gene functions = 'F1', the protein sequence is inferred from the corresponding gene’s sequence and direction using A-Prediction. Otherwise, the protein sequence is inferred from only the gene’s sequence using B-Prediction.

In Example 1, the Where clause is omitted which indicates that the source and destination table cells belong to the same tuple. In this example, the dependency definition applies to all tuples in table Gene. In Example 2, the dependencies are defined between the two tables, Gene and Protein. In this case, the Where clause contains a join between the two tables. Join predicates are mandatory for cross-table dependencies and are restricted to only equality joins between a foreign key in the destination table, e.g., Protein.GID, and the primary key in the source table, e.g., Gene.GID. This restriction ensures that each destination table cell is attached to unique source table cells. Notice that the predicates of the two Add Dependency constructs in Example 2 are not disjoint, i.e., one tuple may have the gene function equals "F1" and the start position greater than 10000. In this case, the destination table cell of that tuple, i.e., the protein sequence, follows the definition of the second dependency (the most recent one) as will be explained using the overriding property in Section 4.

4 DATA MANIPULATION OPERATIONS

In this section, we present the data manipulation operations in HandelsOn DB and define how they affect the value and status of the database cells. These operations represent the interfaces through which the dependency graph—created from the user-defined real-world and computable dependencies—is manipulated. Conceptually, each table cell has a status (0 = up-to-date, 1 = outdated) in addition to the cell value. That is, Relation R having n attributes is represented as: R = {r =< (C1.value, C1.status), ..., (Cn.value, Cn.status) >}

We define five data manipulation operations, namely, insert(t), delete(t), update(c), invalidate(c), and validate(c), where t is a tuple and c is a database cell. In Figure 4, we demonstrate a sequence of cumulative operations over two sample tables T and S. Tuples in S reference the tuples in T using the foreign and primary keys S.T.fk, and T.T.pk, respectively. The dependencies among the two tables are depicted in Figure 4(a), where the dashed and solid lines represent computable and real-world dependencies, respectively. The semantics of each dependency are presented in Figure 4(b). Figure 4(c) gives the state of the database after performing each operation. The black-marked table cells represent the outdated values while the red-marked ones represent modified or newly inserted values.

Throughout this section, we use the following shorthand notations for a given database cell c. The database cells from which c is derived are called InputParameters(c). The database cells that are derived from c through real-world and computable dependencies are called RealworldOutputs(c), and ComputableOutputs(c), respectively.

- **invalidate(c):** invalidates c and all database cells that depend on c recursively, i.e., database cells in ComputableOutputs(c) and RealworldOutputs(c). For example, when Operation 1 in Figure 4 invalidates the database cell (t4,r1) because of the existence of the real-world activity F2, the invalidation propagates recursively to the dependent database cells (t5,r1), (s1,r1), and (s3,r1).
- **validate(c):** validates c only if InputParameters(c) are all up-to-date. If c is validated, then the validate procedure is called recursively on the database cells in ComputableOutputs(c). The database cells in RealworldOutputs(c) are not validated automatically because they are waiting on real-world activities to be performed. Referring to Operation 2 in Figure 4, when Activity F2 is externally performed and its result (value 13) is updated in (t4, r1), the dependent table cell (t5,r1) is re-computed and validated automatically through the computable dependency involving F3. However, the validation does not propagate to (s1, r1) because F4 is a real-world activity.
- **update(c):** updates the value of c as well as the values of ComputableOutputs(c). If c is currently invalid, then c is validated only if InputParameters(c) are all valid. Otherwise, the status of c remains unchanged. Moreover, since the value of c is modified, the database cells in RealworldOutputs(c) are invalidated. For example, Operation 1 in Figure 4 re-computes and modifies the value of (t1, r1) from ‘9’ to ‘3’ while maintaining its status valid. In contrast, the value in (t4, r1) is invalidated through the real-world dependency involving F3. Similarly, Operation 3 results in re-computing the value in (s3, r1) to be ‘90’ instead of ‘150’ through F5. However, this update does not validate (s3,r1) because one of its input parameters, i.e., (s1,r1), is still outdated.
- **delete(t):** If the values in t are neither source nor destination parameters to any dependency, then t is deleted without any further processing. The same rule applies if the values in t are only destination parameters to some existing dependencies. However, if the values in t are source parameters to some existing dependencies, then either the deletion of t is rejected, or t is deleted and the destination parameters of these dependencies are invalidated. The default behavior in the system is to reject the deletion of t. This behavior can be altered at the table level using the following command:

```
Alter Table <tableName> Add Constraint <constName> On Delete Propagate Invalidation;
```
Fig. 4. Examples of database operations under a set of user-defined dependencies.

- **insert(t)**: One of the restrictions on the Add Dependency construct is that the destination table has to contain foreign keys that reference the primary keys in the source tables (Refer to Section 3). This restriction ensures the uniqueness of the source table cells for a given destination table cell. It also simplifies the insertion procedure since it ensures that at the insertion time of a tuple in the database, we need to address two issues: (1) whether or not DI will form a cycle with other existing dependencies, and (2) whether or not DI will override other existing dependencies.

A cycle among a set of user-defined dependencies indicates that the derivations among the destination and source parameters of these dependencies may loop indefinitely. Cycles are not allowed in HandsOn DB according to the following definitions— Recall that DI.DP and DI.SP correspond to the destination and source parameters of dependency DI, respectively.

**Definition (Cycle Formation):** Dependency instances \( DI_1, DI_2, \ldots, DI_n \) form a cycle if \( DI_1.DP \cap DI_{i+1}.SP \neq \phi \), for \( 1 \leq i < n \), and \( DI_1.DP \cap DI_2.SP \neq \phi \).

**Property (Cycle-Free Dependency Graph):** A newly defined dependency instance \( DI_1 \) is rejected by the system if there exists dependency instances \( DI_j, DI_k, \ldots, DI_n \mid DI_1 \cup DI_2 \cup \ldots \cup DI_n \) form a cycle.

The algorithm for detecting and preventing cycles among the dependencies is discussed in Section 6.1.

Since the derivations among the data items may change over time, the user-defined dependencies may change over time as well, i.e., new dependencies may override other existing dependencies. For example, if a database...
cell \(c\) is a destination parameter to dependency \(DI_j\) and a new dependency \(DI_i\) is defined where \(c\) is its destination parameter, then \(DI_i\) overrides \(DI_j\) w.r.t. \(c\). Now \(c\) is being derived according to \(DI_i\) instead of \(DI_j\). Dependencies \(DI_i\) and \(DI_j\) may or may not be of the same type (real-world or computable). The overriding property is defined as follows:

**Definition (Dependency Overriding)** Dependency instance \(DI_i\) is said to override dependency instance \(DI_j\) w.r.t destination parameters \(DP' \neq \emptyset\) if \(DI_i.DP \cap DI_j.DP = DP'\) and \(DI_i\) is defined (chronologically) after \(DI_j\).

The dependency overriding mechanism creates flexibility in the system by allowing derivations of values to change over time while making sure that any database cell \(c\) can be a destination parameter to at most one dependency instance at any given time. Hence, there is exactly one way to derive or infer \(c\) (if any). This guarantees the deterministic behavior of HandsOn DB, otherwise the system cannot determine which function derives a given value.

- **Adding/Dropping dependencies**: A new dependency is added to the database using either the extended `Create Table` or `Alter Table` commands (see Section 3). If an added (or dropped) dependency is defined using the `Alter Table` command including the optional `Invalidate Destination` clause, then the destination table cells that satisfy the dependency predicates are invalidated. Otherwise, the destination table cells remain valid. Newly added dependencies may override existing ones. For example, Operation 7 in Figure 4 defines a new dependency indicating that the values in column \(S.s3\) are new derived from the RWA-F \(F6\) instead of the computable dependency involving \(F5\). The overriding mechanism is described in detail in Section 6.1.

## 5 Extended Querying Mechanism

Initially, all values in HandsOn DB have a valid status. However, as users perform database updates, parts of the underlying database will become potentially invalid (under re-evaluation). Thus, it is crucial for end-users to get alerted when their queries touch or depend on potentially invalid values. In this section, we introduce extended semantics for the query operators to annotate the query results with the status information and to enable evaluating queries on either valid data only (avoid getting false positive tuples), or both valid and potentially invalid data (include false positive tuples).

### 5.1 Predicate Evaluation

A predicate evaluation over a tuple typically results in a boolean value True or False. With the status of each value in the database being up-to-date or outdated, we extend the predicate evaluation to return one of four possible values (4-valued logic): True \((T)\), False \((F)\), Potentially false positive \((+ve)\), and Potentially false negative \((-ve)\). The value True indicates that the tuple qualifies the predicate based on only up-to-date values, and hence, the tuple is certainly part of the answer (e.g., the 2\textsuperscript{nd} tuple against Predicate 1 in Figure 5). The value False indicates that the tuple disqualifies the predicate based on only up-to-date values, and hence, the tuple is certainly not part of the answer (e.g., the 1\textsuperscript{st} tuple against Predicate 1 in Figure 5). The value Potentially false positive indicates that the tuple qualifies the predicate but based on outdated values, and hence, the tuple is potentially a false positive (e.g., the 3\textsuperscript{rd} tuple against Predicate 1 in Figure 5). The value Potentially false negative indicates that the tuple disqualifies the predicate based on outdated values, and hence, the tuple is potentially a false negative (e.g., the 5\textsuperscript{th} tuple against Predicate 1 in Figure 5). Although it seems easier to set the potentially-invalid values to Null and use the 3-valued logic supported by most DBMSs, i.e., True, False, and IS NULL, the proposed 4-valued logic has several advantages: (1) Columns in the database may be defined as `Not NULL` and hence the Null value may not be allowed in the first place. (2) Null values are known to be problematic to work with as they mandate the use of special functions, e.g., IS NULL or NVL, when comparing values to avoid nondeterministic evaluation. Nevertheless the complexity of manipulating Null values by query operators such as joins and grouping. (3) Under the 3-valued logic, the query operators still need to be extended to differentiate between the user-inserted Null values and the system-generated Null values, otherwise, simple operation like reporting the potentially-invalid values in a given column cannot be performed.

In Figure 6(a), we present the rules for evaluating binary predicates (Column \(<op>\) Column)— rules for evaluating unary predicates (Column \(<op>\) constant) are trivial. As an example, a predicate `P` over Columns \(c_i\) and \(c_j\) evaluates to `+ve` if the values of \(c_i\) and \(c_j\) satisfy `P` while at least one of the two values is outdated. The truth tables for evaluating multiple predicates are presented in Figure 6(b).

### 5.2 Query Operators

In this section, we present extended semantics for the query operators. Our goal is that the output produced from the query operators should be both semantically meaningful and easily interpretable by end-users. We extended the selection and join operators with three different semantics to enable retrieving tuples that evaluate the predicate(s) to either True, +ve, or -ve. The semantics of the other operators, e.g., duplicate elimination and set operators, have been also extended to take both the status...
and the value of the database items into account while comparing them. The following notations are used in the rest of this section: R and S are relation names, r and s are individual tuples in R and S, respectively, and ci is a column name. When ambiguous, we use r.ci to refer to Column ci in Tuple r.

- **Selection**: Tuples that evaluate the selection predicate to T, +ve, or -ve are of interest since they either satisfy or have the potential to satisfy the query. However, returning these tuples altogether from one operator can be very misleading and hard to interpret. In HandsOn DB, we define three types of selection operators, namely, True Selection (σT), False-positive Selection (σ+), and False-negative Selection (σ−), that return tuples that evaluate to each of T, +ve, or -ve, respectively. The algebraic expressions of the selection operators are as follows.

**True Selection** (σT,r): Selects tuples that evaluate Predicate P to T.

\[ σT,r(R) = \{ r \mid r ∈ R \land P(r) = T \} \]

**False-positive Selection** (σ+,r): Selects tuples that evaluate Predicate P to +ve.

\[ σ+,r(R) = \{ r \mid r ∈ R \land P(r) = +ve \} \]

**False-negative Selection** (σ−,r): Selects tuples that evaluate Predicate P to -ve.

\[ σ−,r(R) = \{ r \mid r ∈ R \land P(r) = -ve \} \]

- **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 6(a). Similar to the selection operator, 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 ⋈T,P,S): Returns the joined tuples r and s that evaluate predicate P to T.

**Fig. 6. Predicate evaluation rules.**
otherwise the returned value will have an outdated status. For example, the group corresponding to the pair \((b, \beta)\) in Figure 7(b) has \(\text{sum}(Z)\) equals 7 with outdated status since the sum depends on the outdated value 5.

We extend the 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:** Consider the following extended query:

\[
\pi \text{GSeq, PSeq} \text{From GENE G, PROTEIN P}
\text{Where G.GID} = + \text{P.GID And GFunction} = -'F2';
\]

where =+, and =- correspond to false-positive and false-negative equality operators, respectively. The query is equivalent to the algebraic expression:

\[
\pi \text{GSeq, PSeq}(\sigma_{-\text{GFunction}='F2'}(\text{GENE} \times_{+,\text{G.GID}=\text{P.GID}} \text{PROTEIN})).
\]

### 5.3 Curation Operators

**HandsOn DB** provides a set of curation operators that help users managing and tracing the dependencies among the data. The query and curation operators are complementary to each other where the former operators allow users to seamlessly query the data, the latter operators allow users to track why certain tuples/values are invalid and how to validate them. In this section, we present three of these curation operators, i.e., dependency tracking (DTrack), hierarchical dependency tracking (HDTrack), and dependency roots (DRoots). All the operators execute over the output from an SQL select statement as depicted below:

\[
\text{[DTrack | HDTrack | DRoots] (}
\text{Select *}
\text{From <table name> }
\text{Where <predicates> )};
\]

For each output tuple \(t\) from the select statement, the DTrack operator reports for each attribute value \(v\) in \(t\), the status of \(v\), the dependency id to which \(v\) is a destination, and the source table cells on which \(v\) depends. DTrack starts by retrieving the \(c_{\text{dependency_id}}\) values stored in \(t\) and then based on the dependency definitions, it executes a reverse query to retrieve the source table cells. The HDTrack operator executes in a similar way as DTrack except that it reports the complete hierarchy of dependencies until it reaches values with no further dependencies, i.e., HDTrack recursively executes DTrack until no further dependencies can be found. Both DTrack and HDTrack help users to trace the dependencies upward in order to re-validate certain values. DRoots, on the other hand, reports all invalid values within the selected tuples that depend only on all up-to-date values—and hence they are the ones to start re-validating the (roots). To find the roots in a given set of tuples, DRoots scans the Pending Activity table (Refer to Section 6.2) for records with pending status and no compensating counterparts. The destination table cells in these records are the set of roots in the database which will then be filtered based on the user’s selection.

### 6 Design Issues

Since the dependency graph may grow massively, materializing and storing it inside the database may involve unnecessary and substantial overhead. Therefore, HandsOn DB utilizes the powerful triggering mechanism in database systems by dynamically generating triggers that enforce the user-defined dependencies and propagate the (in)validation operations accordingly (Section 6.1). The maintenance of the real-world activities pending execution is presented in Section 6.2.

**Storage Scheme:** For each dependency, we store in catalog tables the dependency definition, the names of the source and destination tables and columns, the type of the dependency as either computable or real-world, and a unique identifier (dependency_id) that is assigned to the dependency at the creation time (See the Add Dependency construct in Section 3). For each table cell \(c_i\) in addition to \(c_i\)’s value, we keep two additional system-maintained fields \(c_{\text{status}}\) and \(c_{\text{dependency_id}}\), where \(c_{\text{status}}\) indicates whether \(c\) is up-to-date (\(\text{status} = 0\)) or outdated (\(\text{status} = 1\)), and \(c_{\text{dependency_id}}\) stores the id of the most recent dependency to which \(c\) is a destination parameter (if dependency_id is null, then \(c\) does not depend on other values in the database). When a new dependency having dependency_id, say \(v_i\), is added to the database, all destination table cells belonging to this dependency will have their dependency_id field updated to \(v_i\).

### 6.1 Realization of Dependencies using Triggers

When a new dependency is added to the database, the system extracts the predicates from the Add Dependency construct and automatically generates triggers that enforce the dependency. The processing of a given Add Dependency construct involves four steps: (1) detecting whether or not the new dependency forms a cycle with the already existing dependencies, (2) assigning a unique id \(v_i\) to the dependency, (3) generating a set of triggers over the source and destination tables to enforce the dependency, and (4) overriding other existing dependencies by modifying the dependency_id field of the destination parameters of the new dependency to \(v_i\). Regarding the scalability of the triggering mechanism, we experimented with few thousands of triggers in PostgreSQL over 5 to 10 tables and did not observe any bottleneck due to the number of triggers. Yet, one simple approach to even scale better is to merge multiple triggers together — This is straightforward since triggers are automatically generated by the system. Another approach is to consider more advanced indexing techniques for triggers as proposed in [36].

**Formation of cycles:** Testing the formation of a cycle among the user-defined dependencies can be very expensive if performed at the cell level. HandsOn DB, therefore, detects cycles in two phases; filter (column_level test) and refine (cell_level test). The intuition is that if no dependency is found between columns \(T.c_i\) and \(T.c_j\), then there is no need to check the values in these columns (the filter phase). If a dependency is found between \(T.c_i\) and
In principle, given a new dependency definition, **Hand**(a)sOn DB creates automatically several triggers to enforce the dependency and to propagate the status to dependent values. In Figures 8(a) and (b), we present the **After Update** code templates for real-world, and computable dependencies, respectively. Consider the template in Figure 8(a). The trigger fires when an update occurs to a tuple, say \( t_i \), in Table \( T_i \). Column \( c_i \) is the source column in the dependency. Lines 1 and 2 retrieve the source and destination table cells involved in the dependency. If the dependency_id value of the destination table cell does not match with the id value assigned to the dependency when defined \((c_i)\), then the trigger terminates because the destination cell is no longer a destination parameter to this dependency (Lines 3-5). Lines 7-11 handle the case when the value of \( c_i \) is updated while its status remains the same. If the status of \( c_i \) as well as all other source parameters of the dependency are up-to-date, then a request record is inserted into the **PendingActivity** table (See Section 6.2) to indicate that the involved real-world activity should be performed based on the new value of \( c_i \) (Line 9). In Line 11, the destination table cell of the dependency is marked outdated. The **PendingActivity** table is maintained and populated automatically to help users identify which real-world activities are ready for execution as will be presented in Section 6.2. Lines 12-15 handle the case when the value is updated and the status is modified from outdated to up-to-date. In this case, a request record is inserted into **PendingActivity** only if all other source cells are up-to-date (Line 14). Notice that dest_cell remains outdated until the real-world activity is performed.

The second part of the template (Lines 17-31) handles a change in the status of \( c_i \) without changing its value. In the case when the status of \( c_i \) is modified from up-to-date to outdated, dest_cell is marked outdated if it is currently up-to-date (Line 24). However, if dest_cell is currently outdated and all the other source table cells are up-to-date, then a previous request must have been inserted into **PendingActivity** to validate dest_cell. This request can no longer validate dest_cell because \( c_i \) is now invalid. For this reason, we insert a compensating record (Line 21) into **PendingActivity** to prevent the previous request from validating dest_cell as will be discussed in Section 6.2. In the case when the status of \( c_i \) is modified from outdated to up-to-date, then a request record is inserted into **PendingActivity** only if all other source cells are up-to-date (Line 28).

The computable-function template in Figure 8(b) is a simplified version of that in Figure 8(a). An important difference to note is that since the function involved in the dependency is computable, the trigger executes this function automatically to update dest_cell whenever the
value of \( c_i \) is modified (Line 7). Another key difference is that whenever all the source parameters become up-to-date, the destination parameter is automatically marked as up-to-date (Lines 9, 15). This is unlike the case of the real-world dependencies in which a request record is inserted into \( \text{PendingActivity} \).

### 6.2 Logging and Resuming Pending Activities

When a real-world activity function \( F \) has all its source parameters up-to-date but its destination parameter outdated, a request record (of type \( \text{pending} \)) for \( F \) is inserted into the \( \text{PendingActivity} \) table (Lines 9, 14, and 28 in Figure 8(a)). Before serving that request, i.e., executing \( F \) and reflecting its output value into the database, the source parameters of \( F \) may change again, or may get invalidated. In the former case, more \text{pending} \) requests for executing \( F \) are inserted into \( \text{PendingActivity} \). Users have the option to either serve these requests sequentially or serve only the last request (See the \text{Resume Function} \ command below). In the latter case, a \text{compensating} \ record is inserted into \( \text{PendingActivity} \) (Line 21 in Figure 8(a)) to indicate that any previous pending requests for executing \( F \) can still be served to update the value of the destination table cell but without validating it.

The schema of the \( \text{PendingActivity} \) table consists of: a unique request id, the function name \( F \) to be externally executed, the input arguments to \( F \), an update statement that updates the destination table cell once the new results from \( F \) are known, and a status field that shows the status of the request. The status field takes one of the following values: \text{pending}, \text{completed}, \text{overwritten}, or \text{compensating}. When \( F \) is performed and its output result is available, the result is passed to the system using the following command:

\[
\text{Resume Function} \ <\text{func}_\text{name}> \ \text{Value} \ <\text{func}_\text{output}> \ \text{References} \ <\text{Request}_\text{Id}> \ [\text{Cascade}];
\]

where \( \text{Request}_\text{Id} \) references the unique request id that is assigned to each request in \( \text{PendingActivity} \). The procedure for executing the \text{Resume Function} \ command is presented in Figure 9. If the request has a status of either \text{completed}, \text{overwritten}, or \text{compensating}, then the procedure terminates without further processing (Lines 1-3). If there are previous pending requests targeting the same destination table cell and the optional \text{Cascade} \ keyword is not specified, then the procedure terminates because, in this case, requests have to be served in order (Line 6). If \text{Cascade} \ is included, then any previous pending requests are marked as \text{overwritten} \ and the current request is served (Line 8). Lines 12 and 14 update the value of the destination table cell, and depending on whether or not there are requests more recent than the one currently at hand, the destination table cell either remains invalid (Line 12) or gets validated (Line 14).

### 7 Performance Analysis

We implemented \text{HandsOn DB} via extensions to PostgreSQL that include: (1) adding new SQL syntax for creating the real-world activity functions and modeling the dependencies, (2) adding new data manipulation operations, i.e., \text{invalidate()} \ and \text{validate()}, (3) augmenting mechanisms for automatically creating (or deleting) triggers when dependencies are added (or deleted), (4) introducing new query operators in PostgreSQL with the semantics presented in Section 5, and (5) adding the \text{Resume Function()} \ command for resuming pending activities. In this section, we study the overheads associated with these extensions and demonstrate the feasibility and practicality of \text{HandsOn DB}.

#### Datasets

We use three datasets: \text{Genobase}, a real biological database of size approximately 40MB, \text{PubChem-substance}, a real chemical database of size approximately 300MB, and a synthetic dataset of size approximately 450MB. \text{Genobase} stores the gene details of the Ecoli organism along with different mutation types. \text{PubChem-substance} stores information about chemical substances, e.g., substance\_ids, sources, synonyms, compounds, and atoms. The synthetic dataset is designed primarily to stress on the cascading effect of the dependencies as will be explained later. It consists of 10 tables, i.e., \( R_1, \ldots, R_{10} \). Each table consists of ten attributes, i.e., \( c_1, c_2, \ldots, c_{10} \), in addition to the primary and foreign keys. Each table \( R_{i+1} \) contains two foreign keys that point to the primary keys of tables \( R_i \) and \( R_{i-1} \).

#### Storage

In Figure 10, we study the storage overhead imposed from adding the \text{status} \ (one bit) and \text{dependency\_id} \ (two-bytes integer) columns for each database column. The figure illustrates that the storage overhead ranges from 2% to 7% of the database size. As expected, the storage overhead is relatively insignificant. The reason is that scientific databases typically store many large-size attributes such as text and sequence fields that dominate the storage overhead. \text{PubChem-substance} shows the highest storage overhead because the average length of its attributes is smaller than those of the other two databases.

#### Adding Dependency

In Figure 11, we study the average time needed for adding a new dependency. This time involves detecting whether or not a cycle exists, and creating the required triggers. In this experiment, we vary the number of generated dependencies from \( 2^5 \) to \( 2^{11} \) (the X-axis) distributed over 5 tables with an average cascading length of 2. One half of the generated dependencies has a single source table while the other half
has two source tables. To create multiple non-overlapping dependencies over a single destination attribute, we divide this attribute into disjoint subsets and assume that each subset is inferred or computed using a different function. In the experiment, we study the two cases where the newly defined dependency either invalidates the destination table cells (labeled as ‘With Inv-Dest’) or keeps them as valid (labeled as ‘Without Inv-Dest’). Figure 11 illustrates that the size of the database does not significantly affect the execution time. The reason is that the time taken to detect whether or not a cycle exists and to create the required triggers is not influenced much by the size of the underlying database. Also, the figure illustrates that the average time taken in the case of invalidating the destination table cells is higher than that where the destination table cells are kept valid. The reason is that invalidating the destination table cells will propagate this invalidation to all dependent data items which may span multiple tables.

**Dropping Dependency:** In Figure 12, we measure the average time required for dropping a dependency. In this experiment, we vary the number of destination table cells that belong to the dropped dependency from $2^5$ to $2^{10}$ (the X-axis) and measure the required time under the cases where the destination cells are either invalidated (labeled as ‘With Inv-Dest’) or validated (labeled as ‘Without Inv-Dest’). The measured performance depends on the initial status of the database cells, e.g., invalidating table cells that are already invalid is less expensive than invalidating up-to-date cells, and the same applies for validation. Therefore, we compute each point in the figure as the average over five runs each with a different percentage of the outdated values in the destination column, i.e., {0%, 5%, 10%, 15%, 20%}. Figure 12 illustrates that dropping a dependency can be an expensive operation especially when invalidating a large number of destination table cells. The overhead associated with invalidating the destination parameters of the dropped dependency is around three or four times higher than that associated with validating the destination parameters. The reason for this difference is that most of the data items in the database are already valid, and hence the validation procedure is not as expensive as the invalidation procedure. In general, dropping a dependency is expected to be an infrequent operation especially when the number of associated destination table cells is large. Otherwise, the overhead involved in re-verifying and re-validating these destination table cells would probably dominate the overhead of the Drop Dependency operation.

**Manipulation Operations:** The performance of the data manipulation operations is presented in Figure 13. In this experiment, we use only the synthetic database that is designed primarily to enable creating long cascading paths among the database tables. Each of the ten tables contains a number of tuples that varies from 1,000 to 50,000. Each table $R_i$ contains the following dependency types among its attributes: (1) computable dependency from $c_1$ to $c_2$, (2) real-world dependency from $c_2$ and $c_3$ to $c_4$, and (3) computable dependency from $c_4$ to $c_5$. Each defined dependency targets a small subset of the destination column, and hence multiple dependencies can be defined over the destination column without overriding each other. The database contains also cross-table dependencies defined as follows: (1) computable dependency from $R_i.c_5$ to $R_{i+1}.c_1$, and (2) real-world dependency from $R_i.c_2$ to $R_{i+1}.c_7$. Using this database design, the length of a cascading path varies from 0 to 40 operations. In Figure 13, we study the average time needed to perform each of the update, invalidate, validate, or Resume Function operations. For the first three operations, each measurement represents the average over 50 randomly-selected table cells (five from each table). The figure illustrates that the update operation involves the highest cost. The reason is that the update procedure performs extra processing (including calling the user-defined functions involved in the computable dependencies) regardless of whether the updated table cell is up-to-date or outdated. This is unlike the invalidate and validate procedures that may take no actions if the table cell is already invalid or valid, respectively. The cost of the invalidate operation is less than that of the update operation because if the invalidated table cell is already outdated, then the procedure terminates without any further processing. Otherwise, it involves the cost of invalidating all dependent data items.

**Query Operators:** With respect to data querying, we define extended semantics for the comparison operators, e.g., $=\Rightarrow$, $==\Rightarrow$, and $\Rightarrow$, correspond to the true, false-positive, and false-negative evaluation of the equality operators, respectively. Other operators, e.g., $>$, $<$, and $\leq\Rightarrow$, are extended in the same way. Queries that involve the extend
operators are re-written using the standard operators according to the re-writing rules presented in Figure 16.

In Figure 14, we study the performance of the three types of selection operators over a table consisting of 50,000 tuples from the synthetic database. The select statement is in the form of Select * From R Where R.a OP <const>, where OP is one of the extended selection operators. The values in the selection column R.a have a duplication factor that varies uniformly over the range from 1 to 10. Since the performance of the selection operators depends on the percentage of the outdated values in R.a, we run each experiment over two different percentages of the outdated values, 20% and 80%, as illustrated in Figure 14. We build B+-tree indexes over both the data column involved in the where clause, i.e., R.a, and R.a’s corresponding status column, i.e., R.a_status. In the experiment, we consider three different types of comparison operators: equality, larger than, and inequality. In the case of equality, the true and false-positive selections have relatively lower overhead (compared to false-negative) because they make use of the index on the value columns to find the matching values. The false-negative operator utilizes the index on the status field (in the case of 20% outdated values), but performs a full-scan (in the case of 80% outdated values) which explains the difference in the execution time illustrated in Figure 14. The inequality comparison has the inverse behavior of the equality comparison.

With respect to joins, we focus on studying the performance of the three types of equality joins as depicted in Figure 15. We use two relations R and S from the synthetic database, each consisting of 50,000 tuples. The Select statement is in the form of Select * From R, S Where R.a OP S.b And <ExtraCond>, where OP is one of the extended equality join operators. The values in the join attributes R.a and S.b are randomly generated over the range from 1 to 10,000 with a duplication factor that varies uniformly over the range from 1 to 10. Both columns and their corresponding status attributes have B+-tree indexes. In Figure 15, we consider three different scenarios that trigger different query plans: (1) the scenario where there are no extra conditions in the select statement, called NoCond, (2) the scenario where there is a true equality selection condition on R.a, called TrueCond, and (3) the scenario where there is a false-positive equal-

![Fig. 13. Manipulation operation](image1)

![Fig. 14. Selection performance](image2)

![Fig. 15. Join performance](image3)

![Fig. 16. Re-writing rules of the extended operators](image4)

8 Conclusion

In this paper, we proposed HandsOn DB system for supporting dependencies that involve real-world activities while maintaining the consistency of derived data under update and query operations. HandsOn DB addresses several challenges that include: (1) keeping track of the potentially invalid data items and reflecting