QTrail-DB: A Query Processing Engine for Imperfect Databases with Evolving Qualities

Maha Asiri and Mohamed Y. Eltabakh
Computer Science Department, Worcester Polytechnic Institute (WPI), MA, U.S.A.

Keywords: Imperfect Database, Data's Quality, Quality Propagation, Query Optimization.

Abstract: Imperfect databases are very common in many applications due to various reasons ranging from data-entry errors, transmission errors, and wrong instruments' readings, to faulty experimental setups leading to incorrect results. The management and query processing of imperfect databases is a very challenging problem requires incorporating the data's qualities within the database engine. Even more challenging, the qualities are not static and may evolve over time. Unfortunately, most of the state-of-art techniques deal with the data quality problem as an *offline* task. In this paper, we propose the "QTrail-DB" system that introduces a new quality model based on the new concept of *"Quality Trails"*, which captures the evolution of the data's qualities over time. QTrail-DB extends the relational data model to incorporate the quality trails within the database system. We propose a new query algebra, called *"QTrail Algebra"*, that enables transparent propagation and derivations of the data's qualities within a query pipeline. QTrail-DB is developed within PostgreSQL and experimentally evaluated using real-world datasets to demonstrate its efficiency and practicality.

1 INTRODUCTION

In most modern applications it is almost a fact that the working databases may not be perfect and may contain low-quality data records (Batini and Scannapieco, 2006; Rahm and Do, 2016). The presence of such low-quality data is due to many reasons including missing or wrong values, redundant information, human errors, or network transmission errors. A science survey has revealed that 80.3% of the participant research and scientific groups have admitted that their working databases contain records of low quality, which puts their analysis and explorations at risk (Twombly, 2011). Moreover, a recent IBM report found that the cost of Poor Data Quality for the US Economy around $3 trillion per year. This includes direct costs as well as indirect costs (IBM, 2021).

Even more challenging, the qualities of the data tuples are typically not static, they may change over time (*evolve*) depending on various events taking place in the database. The emerging scientific applications are excellent examples in which tracking and maintaining the data's qualities is of utmost importance. For example, Figure 1 illustrates a possible sequence of operations that may take place in biological databases. First, a data tuple *r* (e.g., a gene tuple) can be imported from an external source to the local

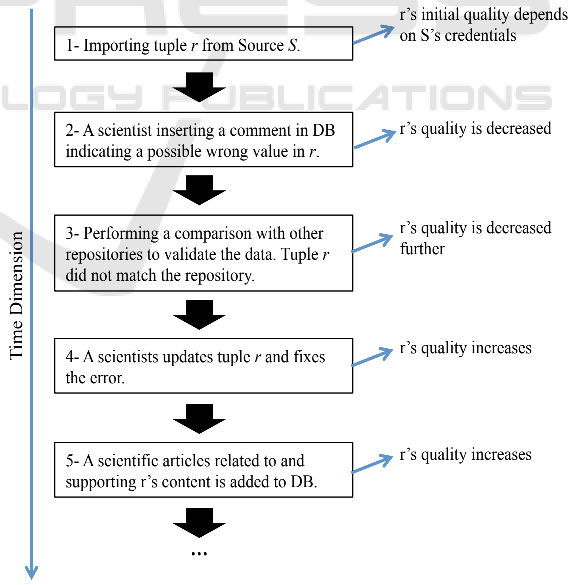

Time Dimension
1- Importing tuple r from Source S.
r's initial quality depends on S's credentials
2- A scientist inserting a comment in DB indicating a possible wrong value in r.
r's quality is decreased
3- Performing a comparison with other repositories to validate the data. Tuple r did not match the repository.
r's quality is decreased further
4- A scientists updates tuple r and fixes the error.
r's quality increases
5- A scientific articles related to and supporting r's content is added to DB.
r's quality increases
...

Figure 1: Database tuples with Evolving Qualities over Time.

database. At that time, *r* would be assigned an initial quality score depending on the source's credibility. Then, a scientist may insert a comment highlighting a possible error in the tuple (e.g., the gene's start position does not seem correct), based on which *r*'s quality should be decreased. After a while, a verification step that compares the local data with an external repository may confirm that r contains an incorrect value, which will further decrease r's quality. Subsequent actions in the database may either increase or decrease r's quality over time, e.g., Steps 4 and 5 in Figure 1, which are an update operation on r (e.g., correcting the gene's start position), and the addition of a scientific article matching r's new content, respectively, should both enhance r's quality. In general, each tuple in the database may have its quality changing over time based on different operations taking place in the database.

In such imperfect databases with dynamic and evolving qualities over time, the standard query processing that treats all tuples the same while ignoring their qualities is indeed a very limited approach. For example, several interesting and challenging questions may arise beyond the standard data querying, which include:

1. What was the quality of tuple r before the last revision?
2. Why r's quality has drastically dropped at time t, and what did we do to fix that?
3. Given my complex query, e.g., involving selection, joins, grouping and aggregation, and set operators, what is quality of each output tuple? Can I trust the results and build further analysis on them or not?

Certainly, supporting these types of questions is of critical importance to end-users and high-level applications. It warrants the need for fundamental changes in the underlying DBMS. In this paper, we propose the *"QTrail-DB"* system, *an advanced query processing engine for imperfect databases with evolving qualities*. We identify two major tasks to be addressed, which are:

Task 1−Systematic Modeling of Evolving Qualities: With the large scale of modern databases, even a very small percentage of low-quality data may translate to a very large number of low-quality records. This makes it very challenging and time-consuming process. Therefore, the underlying database engine must be able to capture and model the data qualities in a systematic way, and also keep track of their evolution over time (Refer to Questions 1 & 2).

Task 2−Quality Propagation and Assessment of Query Results: It is a continuous process of collecting and generating data of various degrees of qualities—with possible interleaving of offline efforts to verify and fix the imperfect tuples. Therefore, it is unavoidable to query the data while having tuples of different qualities. Each tuple r in the output results should have an inferred and derived quality based on input tuples contributed to r's computation (Refer to Question 3).

QTrail-DB proposes a full integration of the data's qualities into all layers of a DBMS. This integration includes introducing a new quality model that captures the evolving qualities of each data tuple over time, called a *"Quality Trail"* and proposing a new relational algebra, called *"QTrail Algebra"*, that enables seamless and transparent propagation and derivations of the data's qualities within a query pipeline.

The key contributions of this paper are summarized as follows:

- Proposing the *"QTrail-DB"* system that treats data's qualities as an integral component within relational databases. In contrast to existing related work, QTrail-DB is the first to quantify and model the data's qualities, and fully integrate them within the data processing cycle. (Section 2)
- Introducing a new quality model based on the new concept of *"quality trails"* that captures the evolving quality history of each data tuple over time and a new query algebra, called *"QTrail Algebra"* that extends the semantics of the standard query operators to manipulate and propagate the quality trails.(Section 3 and 4)
- Developing the QTrail-DB prototype system within the PostgreSQL engine, and evaluating its performance using real-world biological datasets. (Sections 5 and 6)

2 RELATED WORK

Due to its critical importance, data quality has been extensively studied in literature. The most related to our work are the following.

Cleaning and Repairing Technique: A main thread of research is on data cleaning, repairing, and cleansing, where potential low-quality data records are identified, and then fixed. The underlying techniques in these system vary significantly from fully-automated heuristics-based techniques, comparison-based with external sources and repositories, and rule-driven techniques, to human-in-the-loop mechanisms. With the variety of algorithms and techniques for data cleaning, several extensible and generic frameworks have been proposed to integrate these algorithms, e.g., (Dallachiesa et al., 2013). The common theme in all of these systems is that they all work in total isolation from query processing.

Quality Assessment Techniques: On the other hand, very little attention is given to quality assessment at query time. It has been addressed in the context of mining operations, sensor data, and relational databases. The core of these techniques is based on statistical assumptions about the underlying data. And then, each technique studies its domain-specific operations and how they affect the statistical measures.

A major limitation in these systems is the assumed statistics may not be available in many applications. For example, the work in (Ballou et al., 2006)—which is the most related to QTrail-DB—assume that the probability of error in each column in the database is known in advance, which is not the case in many applications. And even if this knowledge is available, it a coarse-grained knowledge over an entire column and not tied to specific tuples.

Uncertain and Probabilistic Databases: Another big area of research is focusing on uncertain and probabilistic databases (Galindo et al., 2006; Widom, 2005a). In these systems, a given data value can be uncertain, and hence it is represented by a possible set of values, a probability distribution function over a given range, or a probability of actual presence. In uncertain databases, the query engine is extended to operate on these uncertain values and tuples, and enforce correct semantics (called *"possible worlds"*). Although uncertainty is related to data qualities in some sense, these systems are fundamentally different from QTrail-DB since the notion of "quality" is not part of these systems.

3 QTrail-DB DATA & QUALITY MODELS

QTrail-DB has an extended data model, where each data tuple carries a *"quality trail"* encoding the evolving quality of this tuple. More formally, for a given relation R having n data attributes, each data tuple $r \in R$ has the schema of: $r = \langle v_1, v_2, ..., v_n, Q_r \rangle$, where $v_1, v_2, ..., v_n$ are the data values of r, and Q_r is r's quality trail. Q_r is a vector in the form of $Q_r = \langle q_1, q_2, ..., q_z \rangle$, where each point q_i is a quality transition defined as follows.

Definition 3.1 (Quality Transition)**.** *A quality transition represents a change in a tuple's quality and it consists of a 4-ary vector ⟨score, timestamp, triggeringEvent, statistics⟩, where* **"score"** *is a quality score ranging between 1 (the lowest quality) and MaxQuality (the highest quality),* **"timestamp"** *is the time at which the score becomes applicable,* **"triggeringEvent"** *is a text field describing the event that triggered this quality transition, and* **"statistics"** *field contains various statistics that will be maintained and updated during query processing.Only* **"score"**, *and* **"timestamp"** *are mandatory fields, while* **"triggeringEvent"**, *and* **"statistics"** *are optional fields.*

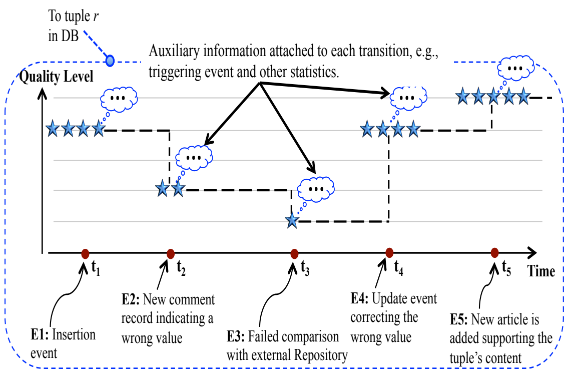

To tuple r in DB
Auxiliary information attached to each transition, e.g., triggering event and other statistics.
Quality Level
Time
t1
t2
t3
t4
t5
E1: Insertion event
E2: New comment record indicating a wrong value
E3: Failed comparison with external Repository
E4: Update event correcting the wrong value
E5: New article is added supporting the tuple's content

Figure 2: Example of r's Quality Trail Corresponding To Operations in Figure 1.

Since r's quality is evolving over time, the length of Q_r's vector is also increasing over time by the addition of new transitions (Refer to Figure 2). The quality trail is formally defined as follows.

Definition 3.2 (Quality Trail)**.** *A quality trail of a given tuple $r \in R$ is denoted as Q_r and is represented as a vector of quality transitions. The transitions in Q_r are chronologically ordered, i.e., for all i, $Q_r[i].timestamp < Q_r[i+1].timestamp$. Moreover, the quality transitins have a stepwise changing pattern, i.e., $Q_r[i]$ is the valid transition over the time period $[Q_r[i].timestamp, Q_r[i+1].timestamp)$.*

Referring to the data tuple r from Figure 1, its corresponding quality trail is depicted in Figure 2. With each of the actions highlighted in Figure 1, r's quality trail will change (evolve) from the L.H.S (the insertion time) to the R.H.S (the current time). Each point in the quality trail is a quality transition. For example, at time t_4, a new quality transition is added to the trail consisting of: $\langle 4, t_4, \text{"}updating\ a\ wrong\ value\text{"}, \{...\} \rangle$. This transition remains valid (the most recent one) until time t_5 when a new transition is added. The *statistics* field and its usage will be discussed in more detail in Section 4.

4 QUALITY PROPAGATION AND ASSESSMENT OF QUERY RESULTS

In this section, we present the extended query processing engine of QTrail-DB for propagating the quality trails within a query plan. We propose a new SQL

```
Merge Operator Ω(Q_1 , Q_2 , … , Q_n)
Output:
    - Quality trail Q_o  ← Initially has no transitions

1.  Position a sweep line L to the left-most transition in Q_1 , Q_2 , … , Q_n
2.  While (more transitions are available) Do
3.      Move L right to the next transition at time t
4.      S={s_1, s_2, …, s_n} ← Set of active transitions at t from Q_1 , Q_2 , … , Q_n
5.      s_out ← A new output transition
6.      s_out.score = Min( S.s_i.score ), 1 ≤ i ≤ n
7.      s_out.timestamp = t
8.      s_out.statistics = StatsCombine(S.s_i.statistics), 1 ≤ i ≤ n
9.      s_out.triggeringEvent = Null
10.     Q_o.addTransition(s_out)
11. End While
12. Return Q_o
```

Figure 3: Pseudocode of the *Merge* Ω Operator.

algebra, called *"QTrail Algebra"*, in which the standard query operators have been extended to seamlessly manipulate the quality trails associated with each tuple. In this section, we assume the quality trails have been created and maintained (The focus of Section 5), and thus we will focus now on the query-time propagation.

Fortunately, in the provenance literature, the propagation semantics of the tuples' lineage is a well studied problem under the different operators. In specific, we use the same semantics as in the Trio system (Widom, 2005b). Therefore, after each algebraic transformation, we can track the input tuples contributing to a specific output tuple without the need for re-inventing the wheel. Yet, the unsolved challenge is *how to translate this knowledge to derivations over the quality trails?*. In the following, we study the semantics of deriving the quality trails of each output record from its contributing input records.

Selection Operator ($\sigma_p(R)$): The operator applies data-based selection predicates p over relation R, and reports the qualifying tuples. Predicates p reference only the data values $v_1, v_2, ..., v_n$ within the tuples. The extension to the selection operator is straightforward since the content of the qualifying tuples do not change, and thus the output quality trails remain unchanged. The algebraic expression is: $\sigma_p(R) = \{r = \langle v_1, v_2, ..., v_n, Q_r\rangle \in R \mid p(r) = True\}$

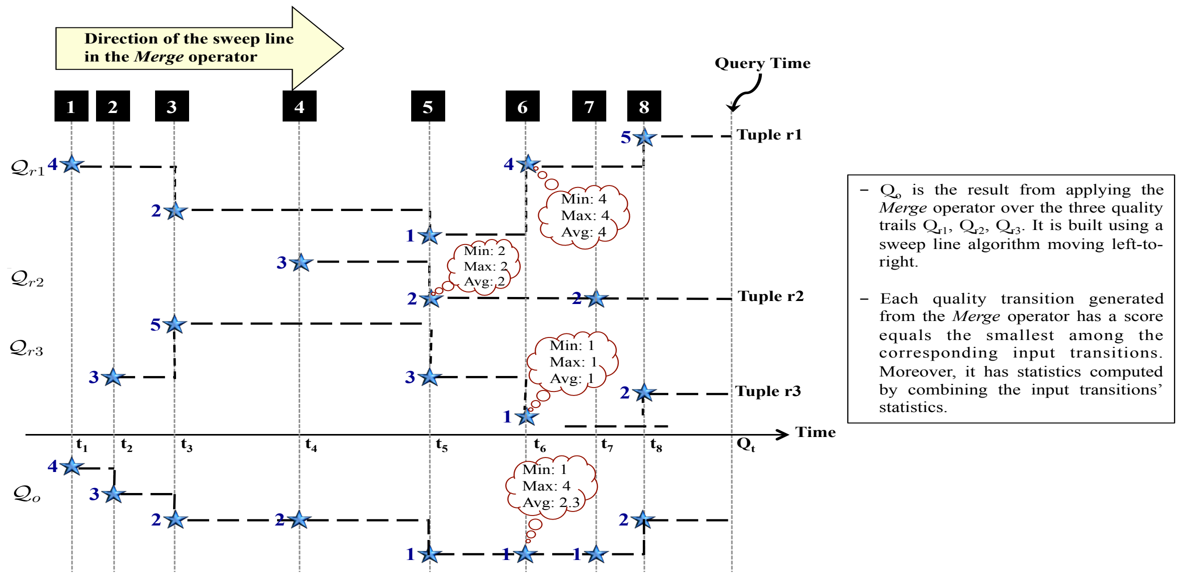
Projection Operator $\pi_{a_1,a_2,...,a_n}(R)$: In QTrail-DB, the quality trails are at the tuple level, and not tied to specific attribute(s) within the tuple. Therefore, the projection operator will not change the quality of its input tuples. That is: $\pi_{a_1,a_2,...,a_n}(R) = \{r' = \langle a_1, a_2, ..., a_n, Q_r\rangle\}\ \forall\ r \in R$.

Merge Operator ($\Omega(Q_1, Q_2, ...)$): Several of the relational operators, e.g., join, grouping, aggregation, among others, involve merging multiple input tuples together to form one output tuple. Therefore, the corresponding input quality trails may also need to be merged and combined together. To perform this merge operation over quality trails, we introduce the new *Merge* operator $\Omega(Q_1, Q_2, ...)$. This operator is not a physical operator, instead it is a logical operator that executes within other physical operators, e.g., join, grouping, and duplicate elimination.

The *Merge* operator's logic is presented in Figure 3, and its functionality is illustrated using the example in Figure 4. Assume combining three tuples r_1, r_2, and r_3 having quality trails Q_{r1}, Q_{r2}, and Q_{r3}, respectively. All quality trails are typically aligned from the R.H.S (which is the query time Q_t), i.e., each quality trial must have a valid transition at time Q_t. However, the trails are not necessarily aligned from the L.H.S since the data tuples may be inserted into the database at different times (See Figure 4). The quality trail of the output tuple Q_o is derived using a *sweep line* algorithm over the input quality trails starting from left to right and jumping over the transition points as illustrated in Figure 4 (Lines 1-3 in Figure 3). The basic idea behind the algorithm is that the quality of the output tuple at any given point in time t should be the lowest among the qualities of the contributing tuples at time t.

Referring to the example in Figure 4, the sweep line starts at Position 1, where only Q_{r1} exists and has a quality level *4-star*, which will be produced in the output. The line then jumps to Position 2, where Q_{r3} starts participating with a quality level *3-star*, and hence a *3-star* transition will be added to Q_o. The sweep line keeps moving to the subsequent positions, and at each position, it calculates the lowest quality score among the input participants to be the output's quality score at this position (Lines 5-7 in Figure 3). For example, referring to the example in Figure 4(a), at time t_4, the contributing input qualities from Q_{r1}, Q_{r2}, and Q_{r3}, are *2-star*, *3-star*, and *5-star*, and thus the corresponding quality transition on Q_o will have a *2-star* score.

Although Q_o's quality scores reflect only the lowest score among the input values, the *statistics* field associated with each quality transition is intended to provide deeper insights on the other values contributing to the score. Initially, the statistics associated with each quality transition, e.g., `Min`, `Max`, `Avg`, are set to the transition's score value as illustrated in Figure 3. And then, as the transitions get merged, new statistics are computed and get attached to the new quality transition. For example, the sweep line at Position 6 encounters scores *4-star*, *2-star*, and *1-star* transitions along with their initial statistics. Notice that Q_{r2}'s active transition at Position 6 is still the *2-star* transition



- Q_o is the result from applying the *Merge* operator over the three quality trails Q_{r1}, Q_{r2}, Q_{r3}. It is built using a sweep line algorithm moving left-to-right.
- Each quality transition generated from the *Merge* operator has a score equals the smallest among the corresponding input transitions. Moreover, it has statistics computed by combining the input transitions' statistics.

Figure 4: Example of the *Merge* Operator in QTrail-DB.

Function Name	Description
QTransition[] getQualityTrail()	Returns r's quality trail as an array of quality transitions.
Int getSize()	Returns the number of transitions in r's quality trail
QTransition[] addTransition(QTransition *q*)	Augments *q* to the right-most side of r's quality trail. The function returns the new extended quality trail.
QTransition[] replaceTransition(Int *pos*, QTransition *q*)	Replaces the quality transition at position *pos* with the new transition *q*. The function returns the new modified trail.
QTransition[] trim(Char *direction*, Int *num*)	Trims r's quality trail and retains only the first *num* transitions starting from the L.H.S or R.H.S (depending on *direction*). The function returns the new trimmed trail.
…	…

Figure 5: Manipulation Functions on *r.QTrail* Attribute.

occurred at Position 5 (at time t_5). These statistics will be combined by the *Merge* operator to compute the new statistics of the Q_o's new transition (Line 8 in Figure 3).

Finally, the newly created transition is added to the output quality trail (Line 10 in Figure 3).

More operators are presented in details in the full version of this paper (Author A, 2023).

5 CREATION & MAINTENANCE OF QUALITY TRAILS

In this section, we present the creation and maintenance mechanisms of the quality trails. Since QTrail-DB is a generic engine, the goal is to design a set of APIs that will act as the interface between QTrail-DB and the external world. More specifically, QTrail-DB allows the database developers to manipulate the quality trails as any other attribute in the database. The quality trails are designed as special attributes added to the database relations, i.e., each relation *R* has an automatically-added special attribute called *"QTrail"*. *QTrail* attribute is of a newly added user-defined type representing an array of quality transitions (Refer to Definitions 3.1, and 3.2). On top of this new type, a set of manipulation functions has been developed as presented in Figure 5. These built-in functions are by no means comprehensive, but they are basic functions on top of which the database developers may create more semantic-rich functions.

In Figure 5, we present few of the developed functions including the descriptions of each (more details available in the paper full version (Author A, 2023)). In addition to these functions, we have also developed a set of functions to manipulate a given quality transition, e.g., building a quality transition, and setting or retrieving specific fields within a transition[1].

6 EXPERIMENTS

Setup: QTrail-DB is implemented within the PostgreSQL DBMS (Stonebraker et al., 1990). A Quality Trail is modeled as a new data type, i.e., a dynamic array of quality transitions. The default stor-

[1]Other storage schemes are possible without affecting the core functionalities of QTrail-DB. Only the implementation of the APIs may change.

age scheme is called *"QTrail-Scheme"*, in which each of the users' relations is automatically augmented with a new *"QTrail"* column as presented in Section 5. QTrail-DB is experimentally evaluated using an AMD Opteron Quadputer compute server with two 16-core AMD CPUs, 128GB memory, and 2 TBs SATA hard drive. QTrail-DB is compared with the plain PostgreSQL DBMS to study the overheads associated with the new functionalities (w.r.t both time and storage).

Application Datasets: We use a subset of the curated UniProt real-world biological database (The Universal Protein Resource Databases, 2023). UniProt offers a comprehensive repository for protein and functional information for various species. We extracted four main tables including `Protein`, `Gene`, `Publication`, and `Comment`.

Our dataset consists of approximately 750,000 protein records ($\approx$ 4.7GBs), 1.3 x 10^6 gene records ($\approx$ 8GBs), 12 x 10^6 publication records ($\approx$ 4.5GBs), and 8 x 10^6 comment records ($\approx$ 6.5GBs). Thus, the total size of the dataset is approximately 24GBs.

Workload: We focus on tracking the qualities of the tuples in the `Gene` and `Protein` tables under the addition of new publications and comments. The quality score varies between 1 (the lowest quality), and 10 (the highest quality). To build the quality trails, we implemented an *"After Insert"* database trigger on each of the `Publication` and `Comment` tables such that with the insertion of a new publication or comment, the quality of the corresponding gene or protein will be updated. We assume that the insertion of a new publication increases the quality (unless the quality score is already the maximum, in which case the new quality transition will have the same score as the previous one). For the comment values, each comment in UniProt has a code indicating the type of this comment. One of these types is *"CAUTION"*, which indicates a possible error or confusion in the data. All comments having the *"CAUTION"* type are assumed to decrease the quality (unless the quality score is already the minimum, in which case the new quality transition will have the same score as the previous one). For the other comment values, we randomly labeled each one as "+", "-", or "$\sim$", which indicates increasing, decreasing, or retaining the previous quality score, respectively [2].

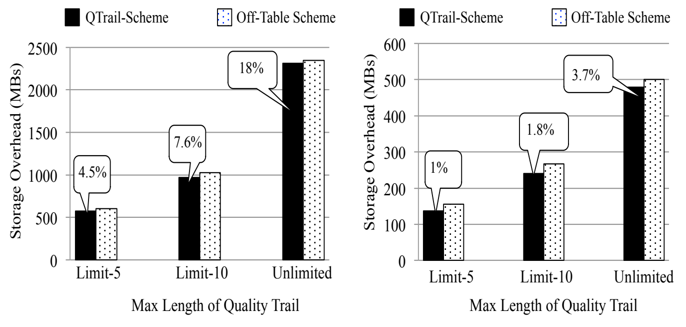
Unless otherwise is specified, we assume the following: (1) Each increase or decrease in the quality score is performed one step at a time, i.e., ± 1, and (2) If a quality transition is storing statistics— Referred to as *"Full Transitions"*— then three types of statistics are maintained, which are the {`Min`, `Max`, `Avg = (Sum, Count)`}. The insertions of the publication and comment records are randomly interleaved because the database does not maintain a global timestamp ordering the records' creation. When evaluating the query performance of QTrail-DB, we compare against the standard query processing in which the quality trails are not even stored in the database. Finally, the query optimizer of PostgreSQL has not been touched or modified, and hence the queries used in the evaluation are optimized in the standard way.

Storage and Maintenance Evaluation: In Figure 6, we study the storage overhead introduced by the quality trails. To put the comparison into perspective, we compare *"QTrail-Scheme"* with another alternative where the quality trails of a given table *R* are stored in a separate table *R-QTrail (OID, QTrail)* that has one-to-one relationship with *R*. This scheme is referred to as *"Off-Table Scheme"*. We study the storage overheads under the two cases of: (1) *Full Transitions*, where each quality transition has content in all its fields; the mandatory ones (*score*, and *timestamp*), and optional ones (*triggeringEvent*, and *statistics*) (Figure 6(a)). In this case, the *triggeringEvent* field is a string of length varying between 50 and 100 bytes. And (2) *Minimal Transitions*, where each quality transition has content in only the mandatory fields (Figure 6(b)).

In each of the two figures, we measure the overhead under different constraints on the maximum size of a quality trail (the x-axis). The values *Limit-5*, *Limit-10*, and *Unlimited* indicate keeping only up to the last 5, 10, or ∞ transitions. The y-axis shows the absolute storage overhead, while the percentages inside the rectangle boxes show the overhead—more specifically that of the QTrail-Scheme—as a percentage of the sum of `Gene` and `Protein` sizes ($\approx$ 12.7GBs). As Figures 6(a) and 6(b) show, there is no big difference between both storage schemes in all cases. The *Off-Table* scheme is slightly higher because of the storage of the unique tuple Id values (*OID* column). In general, the quality trails do not introduce much storage overhead even under the worst case where the entire quality history is stored, e.g., the overhead is around 18% (for Full Transitions), and 3.7% (for Minimal Transitions). It is worth highlighting that under the *Unlimited* case, the longest quality trail consisted on 37 transitions.

In Figure 7, we study the maintenance overhead of the quality trails. We consider the *Unlimited* case of quality trails. To have fair comparison, we measure the time of updating a quality trail, e.g., adding new transitions, w.r.t the time of updating other tra-

[2] We used random labeling since developing a free-text semantic extraction tool (or leveraging an existing tool) is not the focus of this paper.



(a) Storage Overhead Under Full Transitions (b) Storage Overhead Under Minimal Transitions

Figure 6: Quality Trail Storage Overhead.

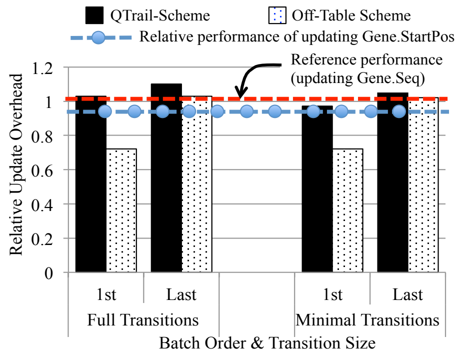
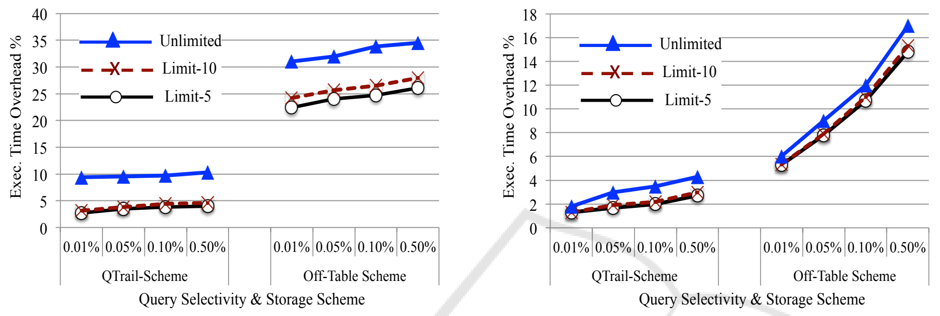


Figure 7: Quality Trail Update Performance.



Select Id, Name, Seq
From Gene
Where Id > *val1*
And Id < *val2*;

Select Id, Name, Seq
From Protein
Where Id > *val1*
And Id < *val2*;

(a) SP Query Performance (No Index) (b) SP Query Performance (With Index) (c) SP Query Templates

Figure 8: Performance of Select-Project (SP) Queries.

ditional fields, e.g., updating a text or integer fields. In all cases, the update operation takes place from within an *After Insert* trigger over the Comment table as described in the experimental workload above. Inside the trigger, the corresponding gene is retrieved to update its data or its quality trail (See Example 2 in Section 5). The retrieval from the DB uses a B-Tree index on the Gene.ID column. Each operation is repeated 20 times, and their average is what we report. As illustrated in Figure 7, we use the operation of updating a string field, more specifically replacing a segment of Gene.Seq field with another segment, as our reference operation. That is, its execution time is normalized to value 1, and the other operations will be measured relative to this operation. The relative performance of updating an integer field, more specifically Gene.StartPos, it also depicted in the figure. On average, the overhead of updating the integer field is around 94% of updating the string field.

For updating the quality trails, we consider both the *QTrail-Scheme*, and the *Off-Table Scheme*, and the two cases of *Full Transition*, and *Minimal Transition*. For each case, we run a batch that consists of 20 transactions inserting records into the Comment table, which yield to updating the genes' quality trails. On the x-axis, we report the average performance over the entire batch under two scenarios: (1) The batch is the 1^{st}, i.e., all quality trails are empty, and (2) The batch is the *Last*, where all other records have been inserted and the quality trails are almost complete. The results in Figure 7 show that operating on and updating a quality trail structure is very comparable to updating other database fields. Under the *QTrail-Scheme*, where the table to be queried and update is the Gene table, the relative overhead ranged from 0.98% (The 1^{st} batch with minimal transitions) to 1.11% (The last batch with full transitions). The performance of the *Off-Table Scheme*, where the table to be queried and updated is called Gene-QTrail, is almost the same except for the 1^{st} batch case, where the Gene-QTrail table is empty.

SP Query Performance: In the next experiments, we evaluate the propagation and derivation of the quality trails at query time under various types of queries. In Figure 8, we present the performance of Select-Project (SP) queries. The query templates are presented in Figure 8(c). On the x-axis of Figures 8(a) and 8(b), we vary the query selectivity from 0.01% (corresponds to 75 protein tuples, or 130 gene tuples) to 0.5% (corresponds to 3,750 protein tuples, or 6,500 gene tuples), and consider the two storage schemes *QTrail-Scheme* and *Off-Table Scheme*. The y-axis measures the propagation overhead of the quality trails w.r.t the standard query performance, i.e., no quality trail storage or propagation. We consider the propagation of the quality trails under the max size constraints of *Limit-5*, *Limit-10*, and *Unlimited*. Under each configuration, we execute 5 queries on each of the `Gene` and `Protein` tables, and then report the average of the observed overheads across the 10 queries. Figures 8(a) and 8(b) show the results under the cases where the queries are executed without and with an index, respectively.

The results show a big difference between the *QTrail-Scheme* and the *Off-Table Scheme*. This is mainly because the two operators that directly read from disk have different implementation under the two storage schemes. In the *QTrail-Scheme*, they are implemented such that they read the quality trails from the data tuples without any additional overhead. Whereas in the *Off-Table Scheme*, they are implemented to join the data tuples with the other table that contains the quality trails. All the other operators are independent of the physical storage of the quality trails as they read them from the operators' buffers in the query pipeline. Since join is an expensive operation, the *Off-Table Scheme* encounters higher overhead. In Figure 8(a), the query selectivity does not play a big factor because a complete table scan is performed regardless of the selectivity, which dominates most of the cost. In contrast, in Figure 8(b), an index is used to select the data tuples satisfying the query's predicates, and thus the performance is more sensitive to the query selectivity. For the *Off-Table Scheme*, the overhead increases as the selectivity increases—and consequently the join cost increases. As the figure shows the *Off-Table Scheme* encounters between 2.5x and 6x higher overhead compared to the *QTrail-Scheme*. It is important to highlight that the select and project operators do not apply any manipulation over the quality trails, and thus the encountered overheads are mostly due to the additional storage introduced by the quality trails.

Additional experiments and results are available in the extended version of this paper (Author A, 2023).

7 CONCLUSION

We proposed QTrail-DB as an advanced query processing engine for imperfect databases with evolving qualities. At the conceptual level, QTrail-DB enables high-level applications to model their data's qualities inside the database system, keep track of how the qualities evolve over time, and build more informed decisions based on the automatically quality-annotated query results. At the technical level, QTrail-DB involves several novel contributions including: (1) Introducing a new quality model based on the new concept of *"quality trails"* in contrast to the commonly-used *single-score* quality model, (2) Extending the relational data model to include the quality trails, and (3) Proposing a new query algebra, called *"QTrail Algebra"*, which extends the standard query operators as well as introduces new quality-related operators for the propagation and derivation of quality trails at query time. The experimental evaluation has shown the practicality of QTrail-DB, and the efficiency of its design choices.

REFERENCES

Author A, A. B. (2023). Anonymized title of the paper.

Ballou, D. P., Chengalur-Smith, I. N., and Wang, R. Y. (2006). Sample-Based Quality Estimation of Query Results in Relational Database Environments. *IEEE Knowledge and Data Engineering*, 18(5).

Batini, C. and Scannapieco, M. (2006). Data Quality: Concepts, Methodologies and Techniques. *Addison-Wesley*.

Dallachiesa, M., Ebaid, A., Eldawy, A., Elmagarmid, A. K., Ilyas, I. F., Ouzzani, M., and Tang, N. (2013). NADEEF: a commodity data cleaning system. In *SIGMOD Conference*, pages 541–552.

Galindo, J., Urrutia, A., and Piattini, M. (2006). Fuzzy databases: Modeling, design, and implementation. *Idea Group Publishing*.

IBM (2021). The high cost of poor data quality for the us economy. https://www.ibm.com/thought-leadership/institute-business-value/report/poor-data-quality. Accessed on May 5, 2023.

Rahm, E. and Do, H. H. (2016). Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 39(2):3–13.

Stonebraker, M., Rowe, L. A., and Hirohama, M. (1990). The implementation of POSTGRES. *TKDE*, 2(1):125–142.

The Universal Protein Resource Databases (2023). http://www.ebi.ac.uk/uniprot/.

Twombly, M. (2011). Science online survey: Support for data curation. *Science Journal*, 331.

Widom, J. (2005a). Trio: A system for integrated management of data, accuracy, and lineage. *CIDR*, pages 262–276.

Widom, J. (2005b). Trio: A system for integrated management of data, accuracy, and lineage. *CIDR*, pages 262–276.