Joins In Quality-Aware Database Systems

A Major Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Yuchen Liu (CS)

March 4th, 2016

Approved by:

Professor Mohamed Y. Eltabakh (CS)

This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review.

Abstract

The purpose of this project is to create a quality-aware database system that fundamentally extends the standard database management systems to support imperfect database with evolving qualities. In the real world, the management and query processing of imperfect databases is a very challenging problem as it requires incorporating the data's qualities within the database engine. In this project, quality-aware database introduces a new quality model that captures the evolution of the data's qualities over time and a new query algebra focused on select and join operators that enables seamless and transparent propagation and derivations of the data's qualities within a query pipeline. As a result, a query's answer will be automatically annotated with quality-related information at the tuple level.

Summary

As with any other model, integrity is one of the most important consideration of database, which means the database should ensure the accuracy of the representation of the real world. Yet, our knowledge of the real world is imperfect thus making imperfect databases very common in many applications. This project will introduce a quality-aware database system to support imperfect database with evolving qualities.

In order to create this new quality-aware database system, this project will include the background information about database system, imperfect database, low quality data and PostgreSQL. After this, it will conduct a research about related work to learn about existing techniques used for handling data quality problem. Next, this project will describe quality-aware database system in detail. Finally, this project will test the performance of quality-aware database system and analyze the test result.

Table of Contents

Chapter 1 Introduction **6**

Chapter 2 Background **10**

2.1 The Evolution of Database Systems 10

2.1.1 Early Database Management Systems 11

2.1.2 Relational Database Management Systems 11

2.2 The Introduction of Imperfect Database 12

2.2.1 Imperfect Information 12

2.2.2 Imperfect Manipulation 12

2.2.3 Other Imperfections 13

2.3 The High Cost of Low Quality Data 13

2.4 The Introduction of PostgreSQL 13

Chapter 3 Related Work **15**

Chapter 4 System Design & Methodology **18**

4.1 Quality-Aware Database Data & Quality Models 18

4.1.1 Definition of Quality Node 18

4.1.2 Definition of Quality 20

4.2 Quality Propagation 21

4.2.1 Selection Operation 21

4.2.2 Join Operation 21

4.2.3 Properties of Merging Qualities 23

4.3 Specific Implements in PostgreSQL 25

4.3.1 src/backend/parser/parser.c 25

4.3.2 src/include/parser/parser.h 28

4.3.3 JoinHelper.sql 28

Chapter 5 Results and Analysis **31**

5.1 Description of Test Tables 31

5.2 Process of Testing 32

5.3 Result of Testing 33

5.4 Analysis of Result 34
Chapter 6 Conclusion **38**
Reference **39**
Appendix – Implemented Functions' Code **43**

Chapter 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. The presence of such low-quality data is due to many reasons including missing or wrong values, redundant and conflicting information from multiple sources, human errors in data entry, machine and network transmission errors, or even wrong assumptions or instruments' calibration during scientific experimentations that lead to inaccurate results.

Even more challenging, the qualities of the data tuples are typically not static, instead they may change over time 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 1.1 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 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, 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 and trustworthy changing over time based on different operations taking place in the database.

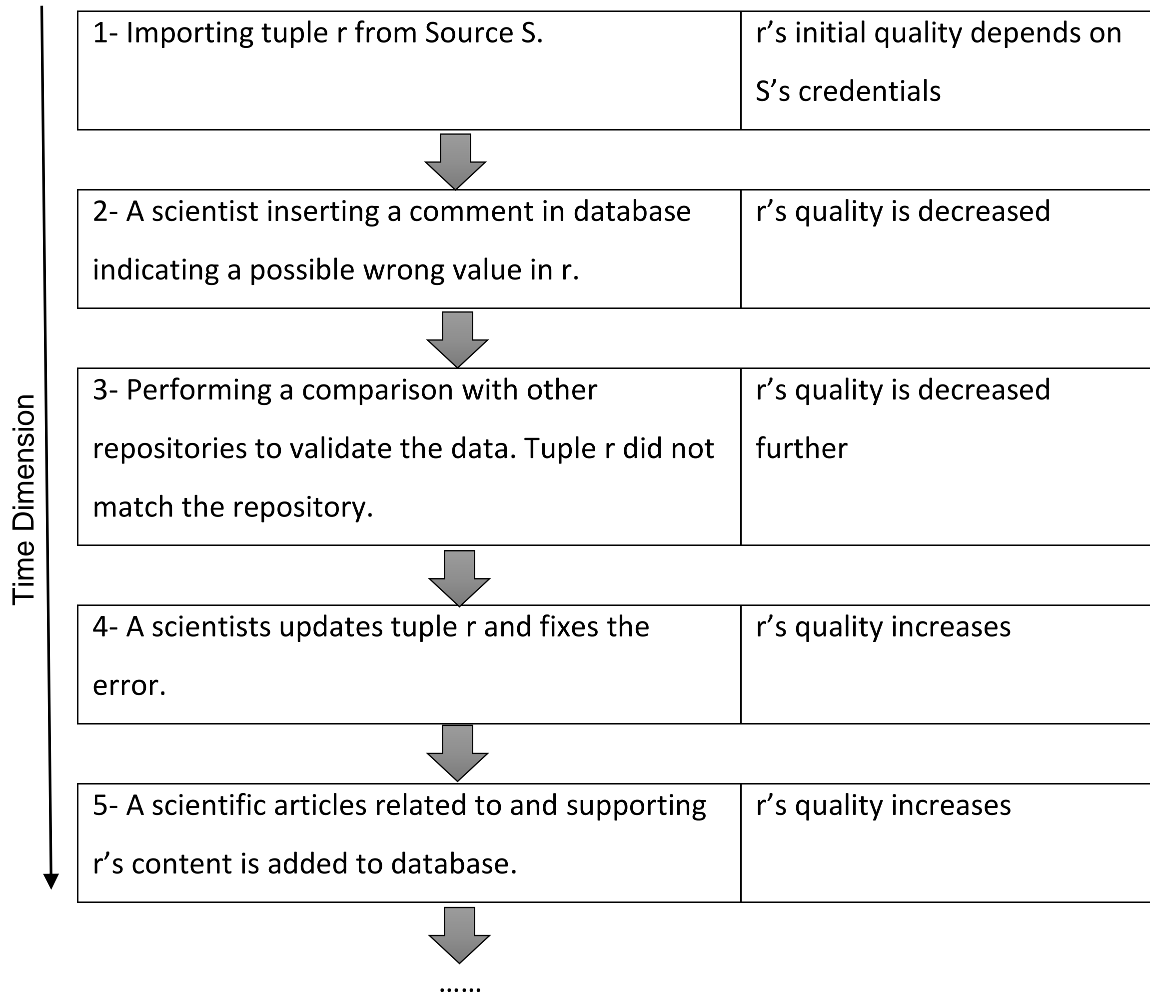

Figure 1.1: Database tuples with Evolving Qualities over Time.

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? (4) Given

my query, how to execute it on only the high-quality tuples, e.g., the quality is above a certain thresh-old? How to join, select, or order the tuples based on their qualities? And (5) among the low-quality tuples in the database, which ones are more important, e.g., frequently participate in queries' answers, to investigate first?

Certainly, supporting these types of questions is of critical importance to end-users and high-level applications. On one hand, without modeling and keeping tracking of the quality information in a systematic way, crucial information will be lost. On the other hand, without assessing the quality of the output results, scientists and decision makers will become less confident about the obtained results. And hence, building any further analysis on low-quality data may not only lead to wrong decisions, but also result in wasting scientists' efforts, resources, and budgets.

It is clear that supporting these types of questions warrants the need for fundamental changes in the underlying DBMS. In this project, we identify three 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 percent-age of low-quality data may translate to a very large number of low-quality records. This makes it very challenging and time-consuming process to identify, isolate, or fix these records instantaneously. 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, e.g., when and why the quality changes (Refer to the aforementioned Questions 1 & 2).

Task 2 Quality Propagation and Assessment of Query Results: It is impractical to assume that applications can freeze their working databases until all records have been fixed, and then enable them for querying. 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. Hence, the query processing engine must be extended to manipulate not only the data values but also their associated 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 the aforementioned Question 3).

Task 3 Quality-Driven Processing and Curation: Another important type of processing—beyond only propagating and de-riving the output’s quality—is the ability to query the data based on their qualities, i.e., quality-driven processing. This includes the ability to, for example, select, join, or order the data tuples based on their qualities, and possibly combine such quality-driven processing with the standard query operators in a single query plan (Refer to the aforementioned Question 4). Another type of analytics is the quality-driven curation in which end-users may want to, for example, track how low-quality tuples affect queries’ results, or rank the low-quality tuples according to their participation in queries for investigation and fixing purposes (Refer to the aforementioned Question 5). Enabling this type of quality-driven processing mandates core changes in the database engine.

As a big research project and due to the limited scope of my MQP, I focused on the task1 and partial of task2 (selection and join operators) while deferring the rest of Task 2 and Task 3 for future work. Quality-Aware database 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”, and proposing a new relational algebra when a SQL command requires selection or join operators, called “Quality Algebra”, which enables seamless and transparent propagation and derivations of the data’s qualities within a query pipeline. Hence, when users input a query which include selection and join operators, the output tuple from this query will be annotated with its derived and inferred quality based on the contributing input tuples.

Chapter 2 Background

This chapter takes a look at the evolution about database system, the introduction of imperfect database and the impact of low quality data and addresses more details on PostgreSQL, because this quality-aware database is developed within PostgreSQL.

2.1 The Evolution of Database Systems

Nowadays database which is a collection of information that can exist over a long time are essential to every business more than at any previous time. When users search the internet, there are databases behind the scenes serving up the information based on the users' requests. As for corporations, the ability to acquire, manage and analyze data about its operations is one of the key factors to determine the success of this corporation, so they always use databases to maintain the important records. In addition, databases also play an important role in many scientific investigations, such as the investigation of the human genome.

A database management system, or DBMS is a powerful tool designed to assist in creating and managing large amounts of data with efficiency along with appropriate security measures. Today, the need for such system is growing rapidly. The following Figure 2.1 shows the expectation of a DBMS.

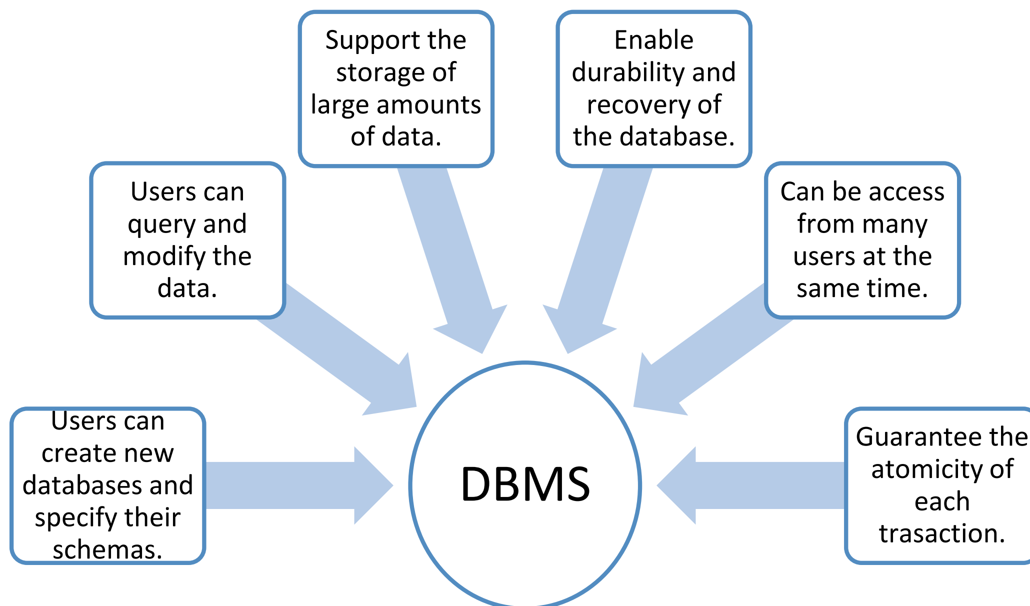

Figure 2.1: the expectation of a DBMS.

2.1.1 Early Database Management Systems

In the late 1960s, the first commercial database management systems that evolved from file system appeared. Although these systems have many weakness, they still can be regard as a huge step because they can hold data for a long period of time and support the storage of large amounts of data. However, they cannot guarantee the durability, which means the data might be lost. Additionally, these systems do not directly support querying the data in the files. Finally, while concurrent access to files by multiple users or process is allowed in these systems, if several users modify the same file at the same time, partial of them would fail to appear in the file.

2.1.2 Relational Database Management Systems

As proposed by Edgar Codd at IBM's San Jose Research Laboratory in 1970, Relational database management system is a database system based on the relational model. Unlike the early database management systems, this kind of database systems provide the user with an organized table view of data. In 1980s, relational database systems establish their status as the dominant type of DBMS and continued to gain widespread use. Even today, we have NoSQL DBMS, in-memory DBMS and many other categories of DBMS, we cannot ignore the popularity of relational database system that altered the commercial landscape.

Developed as part of IBM's System R project, SQL was standardized in the late 1980s. SQL-- Structured Query Language is a high-level programming language designed for managing data in a relational database management system. With SQL, the efficiency of database programmers increased greatly. At present, SQL is the most important query language based on the relational model.

2.2 The Introduction of Imperfect Database

As with any other model, integrity is one of the most important consideration of database, which means the database should ensure the accuracy of the representation of the real world. Yet, our knowledge of the real world is imperfect thus making imperfect databases very common in many applications due to the various reasons.

The following subchapter will describe several different kinds of imperfections within databases.

2.2.1 Imperfect Information

There are three main reasons for imperfect information: error, imprecision, and uncertainty.

1. Error: The information in a database is different from the true information in the real world, and we can say this database information is erroneous. Error is the simplest reason of imperfect information.
2. Imprecision: The information in a database provides a set of possible values, however, in the real world, this value should be unique and is just one element of this set. There are many kinds of imprecision such as disjunctive information (the value is either a or b), negative information (the value is not c), range information (the value is between a and b) and information with error margins (the value is a ± b), in some paper, null value also be regarded as one kind of imprecision.
3. Uncertainty: The information in a database with qualified certainty denotes uncertainty. This is different from the previous one, given an example, “the value is either a or b” is imprecision, and “the value is probably a” is uncertainly.

2.2.2 Imperfect Manipulation

Besides imperfect information, Manipulation and processing imperfection also have impact on the integrity of databases.

Transformations are operations that derive new descriptions from stored descriptions. Query, as the most frequent type of transformation, is often imperfect due to some reasons. For example, the result of a query written by users who do not have

enough knowledge about the information available in the database exhibits a high level of imperfection.

Modifications, which include update and restructuring, can also cause imperfections. Like transformations, modifications are defined by users. The users' uncertainty or imprecision during the process of manipulation and the lack of enough knowledge of database system or the information in this database would lead to the imperfection.

2.2.3 Other Imperfections

There still exist other kinds of imperfections such as faulty experimental setups and imperfect processing (e.g., a recursive query in a specific database system might be terminated after a predetermined period of time).

2.3 The High Cost of Low Quality Data

In the scientific investigations' field, a recent 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.

Moreover, in the business field, the high cost of low quality data is enterprise-threatening. A research has shown that the business cost of low quality data may be as high as 10-25% of an organization's revenue. Besides the high cost, low quality data often breed mistrust and lead to bad or delayed decisions. It has been reported in that wrong decisions and uninformed analysis resulting from imperfect databases cost US businesses around 600 billion dollars each year.

Finally, in the healthcare field, which is a field that close to human's life. It is not tolerated if any low quality data endangers patients' life. 314 billion dollar is the cost of low quality data for each year.

2.4 The Introduction of PostgreSQL

PostgreSQL, often simply Postgres, is one of the world's most advanced open source object-relational database management system. It has been developed over 25

years (the first version is released to a small number of users in June 1989) by the PostgreSQL Global Development Group which is a diverse group of many companies and individual contributors. In this project, the quality-aware database is developed within PostgreSQL system. The backend flowchart of PostgreSQL below shows how PostgreSQL system deals with a SQL command.

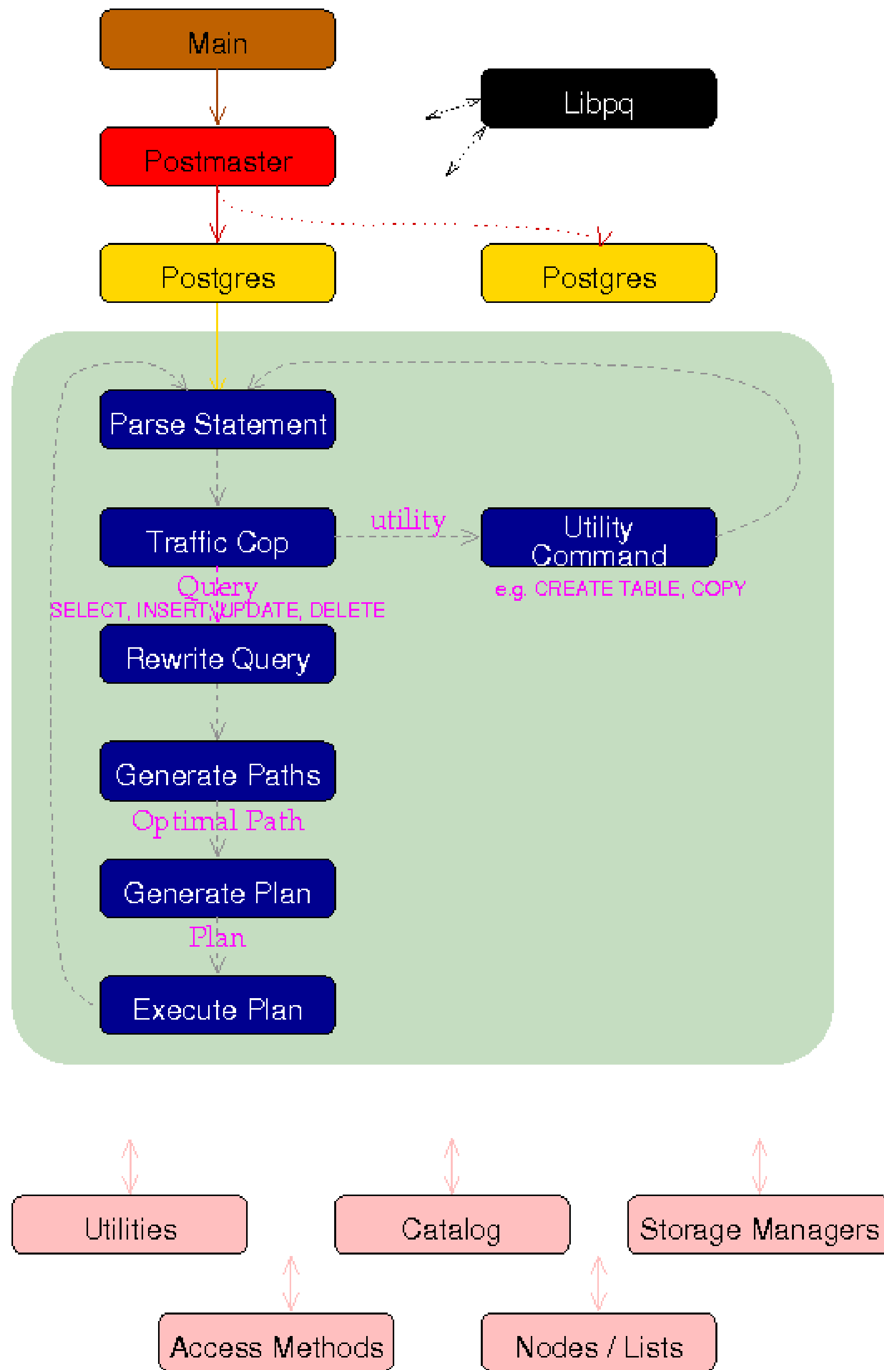

Figure2.2 Backend Flowchart of PostgreSQL

Chapter 3 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 Techniques [1, 2, 5-8, 11-13, 15, 17, 18, 22 and 33]: 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. The common theme in all of these systems is that they all work offline and in total isolation from query processing. And since the data is evolving and growing rapidly in all modern applications, the repairing task is a never-ending time-consuming process. Therefore, it is inevitable that the data will be subject to querying, analysis, and decision making, while it contains low-quality records. Unfortunately, during query processing, none of the above techniques can provide any support for assessing the quality of the results or enabling quality-aware processing. Even worse, if these techniques have identified potential erroneous records and marked them as pending verification or fixing—which may take long time to complete, there is no mechanism to integrate such observations into query processing.

Quality Assessment Techniques [1, 3, 21, 23, 25 and 30]: On the other hand, very little attention is given to quality assessment at query time. It has breed 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, e.g., defining statistical measures such as completeness, soundness, and probability of error. And then, each technique studies its domain-specific operations and how they affect the statistical measures. A major limitation in these systems is that the assumed statistics may not be available in many applications. For example, the work in —which is the most related to the quality-aware database—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, e.g., 1% error rate in column X means that among every 100 values in X , it is expected to find 1 error. Consequently, the estimated output quality is also coarse-grained and cannot be linked to specific tuples. Moreover, these systems assume a single-score quality model without taking into account the fact that data records are long-lived and their qualities evolve over time.

Quality-Aware database is fundamentally different from the above mentioned two categories in that:

1. It proposes a more rich quality model based on the quality trails instead of the single-score quality model,
2. It does not put any assumptions on the data's characteristics, e.g., estimated error rate, instead the quality trails will be incrementally created and maintained as the database evolves over time.
3. Its quality model is fully integrated within the query processing engine.

Quality-Aware database is complementary to the cleaning and repairing techniques in that they together can provide a more comprehensive solution that combines the online quality-aware query processing, and the offline repairing process, respectively.

Uncertain and Probabilistic Databases [1, 16, 28 and 32]: Another big area of research is focusing on uncertain and probabilistic databases. In these systems, a given data value (or an entire tuple) 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 quality-aware database since the notion of "quality" is not part of these systems. Therefore, the uncertain and probabilistic databases can neither model or keep track of the data's qualities, nor enable advanced quality-driven query processing as proposed by the quality-aware database system.

Data Lineage and Provenance [1, 9, 10, 20 and 32]**:** Data provenance is directly related to data quality since the tuples' qualities are based on their provenance. Several systems have addressed the derivation and propagation of the provenance information, and even some systems such as Trio have combined the uncertainty with the provenance. However, there are three key distinctions between quality-aware database and the provenance-based systems, which are:

1. Provenance systems do not provide quantifiable measures on which queries can interact, i.e., lineage information are usually opaque objects with no easy way to apply conditions or transformations on.
2. The storage overhead from the propagated provenance information can be overwhelming and may even exceed the size of the output data, especially under aggregation operations. For example, under relatively simple aggregation queries, some output tuples may carry 100s of provenance links attached to them, but the question is "What does this metadata mean?"
3. Existing provenance techniques do not capture the history (or evolution) of the data tuples, and thus executing the same query Q at times t_1 and t_2 would return the same provenance information even if some changes between t_1 and t_2 have altered the data's qualities.

Quality-Aware database addresses these issues through its new quality and data models, and its extended query engine.

Chapter 4 System Design & Methodology

The purpose of this chapter is to provide an understanding of the system design and methodology in our project. As mentioned in the previous chapter, the project goal was to create a quality-aware database system that fundamentally extends the standard DBMSs to support imperfect database with evolving qualities. This chapter takes a closer look at the data model that used in the project to capture the evolution of tuple's quality and the propagation of this quality. In addition, this chapter also illustrates each specific implement in the PostgreSQL database system.

4.1 Quality-Aware Database Data & Quality Models

Besides original data values, each data tuple in quality-aware database additionally carries a "Quality". This "Quality" is an extended data model 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: $r = \{v_1 ,v_2 ,...,v_n ,Qr\}$, where v_1 to v_n are the original data values of r, and Q_r is r's quality. Q_r is an array in the form of $Q_r = \{q_1 ,q_2 ,...,q_n \}$, where each element q is a quality node defined as follows.

4.1.1 Definition of Quality Node

Quality node (QNode) represents a change in a tuple's quality and it consists of three fields, which are "Qscore", "Qtime" and "TriggerEvent". All of three fields are mandatory. The Figure 4.1 below shows the structure of quality node and the description of each field.

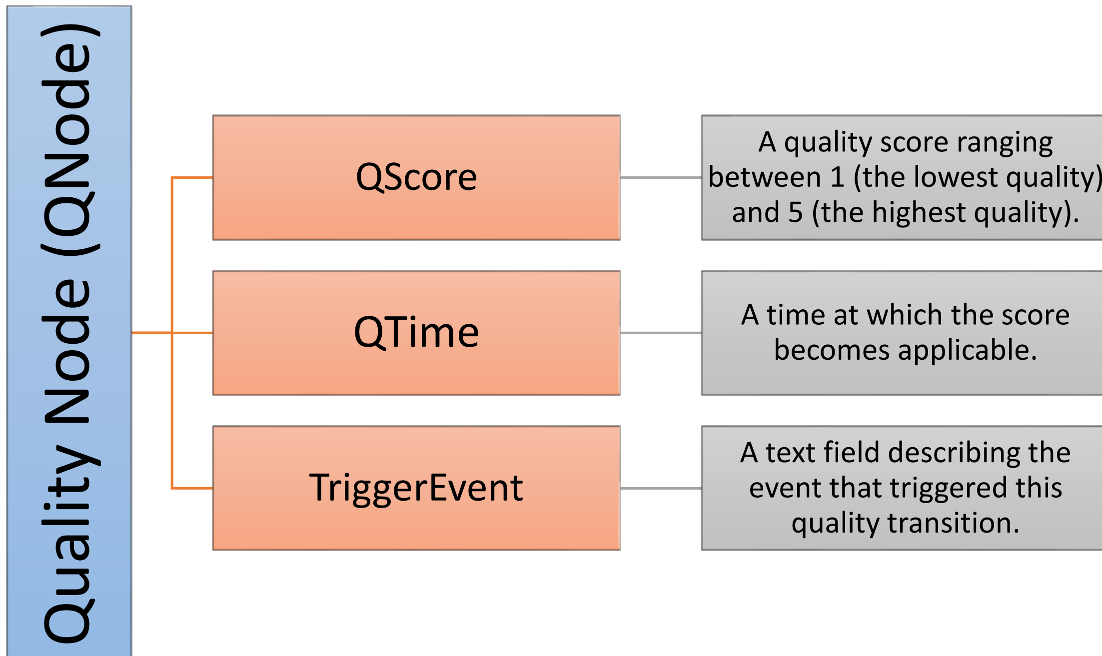

Figure 4.1 Description of Quality Node (QNode)

Since quality of tuple r is evolving over time, the length of Q_r array is also increasing dynamically over time by the addition of new transitions (Refer to Figure 4.2). The "Quality" is formally defined in Chapter 4.1.2.

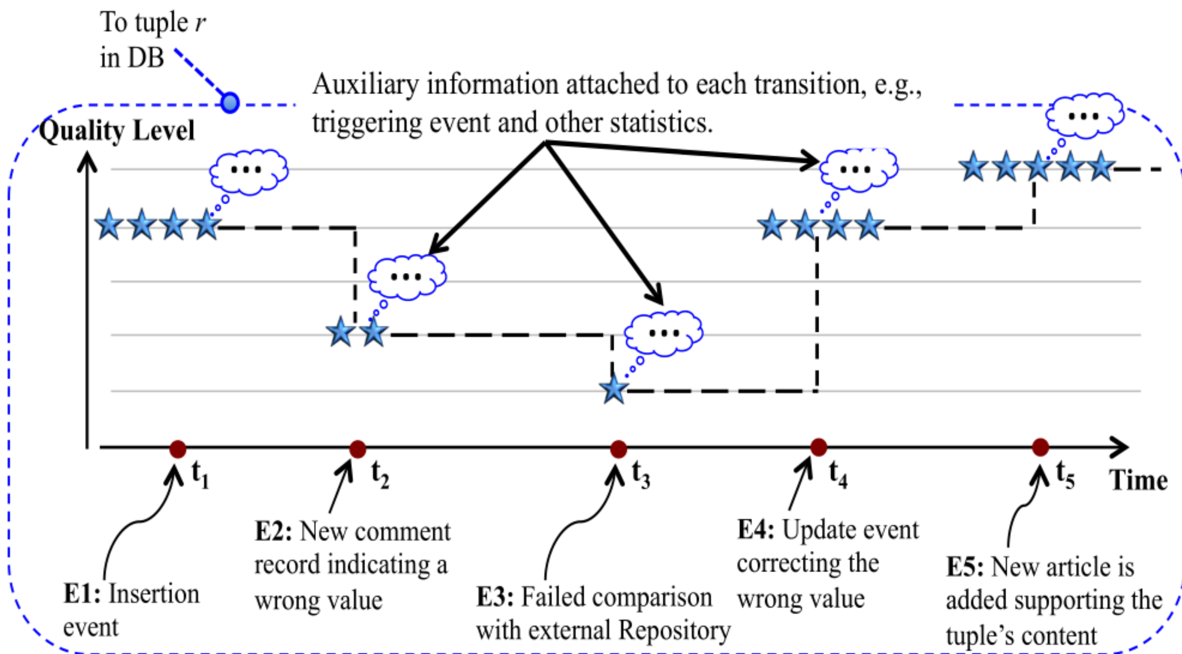

Figure 4.2 Example of r's Quality Corresponding to Operations in Figure 1.1.

4.1.2 Definition of Quality

A quality of a given tuple r ∈ R is denoted as Q_r and is represented as an array of quality nodes (QNode). The transitions in Q_r are chronologically ordered, i.e., Q_r[i].Qtime < Q_r [i + 1].Qtime ∀ i. Moreover, the quality nodes have a stepwise changing pattern, i.e., Q_r [i] is the valid transition over the time period [Q_r [i]. Qtime, Q_r [i + 1]. Qtime). The Table 4.3 below shows how to use command to insert an initial quality of a given tuple and update it corresponding to the quality evolution in Figure 4.2.

Time	Trigger Event	Command
t_1	E_1: Import tuple from a trusty source S.	INSERT INTO *TableName* VALUES(v_1,……,v_n, '{"(5,current_timestamp,import from S)"}');
t_2	E_2: A comment *c* indicates a wrong value.	UPDATE *TableName* SET quality = ARRAY_APPEND(quality,'(3, current_timestamp, command *c* indicates value v_x is wrong)') WHERE *id =3*;
t_3	E_3: Failed comparison with external repository	UPDATE *TableName* SET quality = ARRAY_APPEND(quality,'(2, current_timestamp, Failed comparison with external repository)') WHERE *id =3*;
t_4	E_4: Update event correcting the wrong value	UPDATE *TableName* SET quality = ARRAY_APPEND(quality,'(4, current_timestamp, wrong value v_x has been updated)') WHERE *id =3*;
t_5	E_5: New article is added supporting the tuple's content	UPDATE *TableName* SET quality = ARRAY_APPEND(quality,'(5, current_timestamp, article *ArticalTitle* is added supporting the tuple's content)') WHERE *id =3*;

Table 4.3: Command corresponding to the quality evolution in Figure 4.2

4.2 Quality Propagation

This chapter presents the details for propagating the quality within a query plan that include selection or join operator. A new SQL algebra, called "Quality Algebra" has been defined in order to enable such derivation and propagation in a transparent and pipelined way, in which the selection and join operators have been extended to seamlessly manipulate the quality trails associated with each tuple. In quality-aware database, selection and join operators will consume and produce tuples conforming to the data model presented in Chapter 4.1. The assumption for this chapter is that the quality have been created and maintained (The focus of Table 4.3).

4.2.1 Selection Operation

The operator applies data-based selection predicates p over relation R, and reports the qualifying tuples. Predicates p reference only the data values $v_1, v_2, \dots, v_n$ 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:

$\sigma_p(R) = \{r = (v_1, v_2, \dots, v_n, Q_r) \in R \mid p(r) = \text{True}\}$

4.2.2 Join Operation

Join operator is an instruction to a database to combine data from more than one tables. Therefore, the corresponding input qualities need to be merged and combined together. The logic of join operation is shown below.

1. Initialize output quality Qo which is an array of QNode ← initially has no QNode.
2. Initialize time t ← Earliest QNode time in all qualities.
3. While (time t is not the latest time in all qualities) Do
    a. Insert QNode $Node_i$ at time t from each Quality Q_i (if Q_i exists at t) into set S = {$QNode_1$, $QNode_2$, …, $QNode_n$} ← Set of QNode at t from $Q_1, Q_2, \dots, Q_n$
    b. $QNode_{out}$ ← A new output transition

c. $QNode_{out}$.QScore = Min(S.$QNode_i$.score), $1 \le i \le n$

d. $QNode_{out}$.QTime = time t

e. $QNode_{out}$.TriggeringEvent = Null

f. Qo.add($QNode_{out}$)

g. Set t equal to next earliest QNode time in all qualities.

4. End While
5. Return Qo

Join operation's functionality is illustrated using the example in Figure 4.2. Assume joining three tuples r_1, r_2 , and r_3 having qualities Q_{r1} ,Q_{r2} , and Q_{r3} , respectively. All qualities 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 4.4).

The basic idea behind the algorithm is that, when we merge qualities, 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. This is based on the intuition that low-quality inputs produce low-quality outputs, and that an output tuple should have a high quality at time t only if all its contributing input tuples have high qualities at t.

Referring to the example in Figure 4.4, the earliest time is at Position 1, where only Q_{r1} exists and has a quality level 4-star, which will be produced in the output. Then, the time t jumps to Position 2, where Q_{r3} starts participating with a quality level 3-star, and hence a 3-star QNode will be added to Q_o . The sweep time t keeps moving to the subsequent positions. At each position, it calculates the lowest quality score among the input participants to be the output's quality score, sets quality time equals time t, and sets trigger event equals null (Step 3b-f in the logic of join operation, which merges qualities) at this position. For example, referring to the example in Figure 4.4, 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.

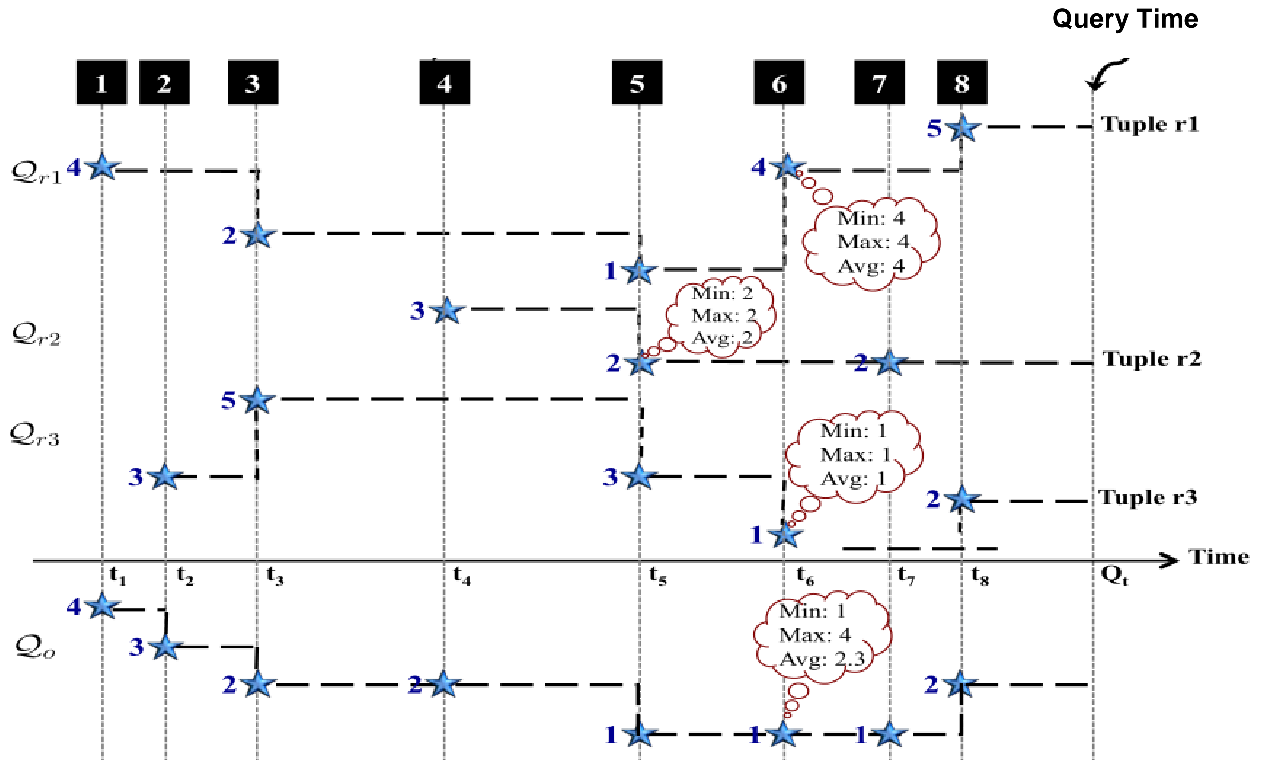

Figure 4.4 Example of the Join operator in Quality-Aware database

4.2.3 Properties of Merging Qualities

From Chapter 4.2.2 the logic of join operator in quality-aware database system, we can find out that, when we merge qualities, the key step is getting result QScore at each time point, which should be the lowest among all the QScores of the contributing tuples at that time point. Because Min operator is commutative and associative, in other words, min(x,y) = min(y,x) and min(min(x,y),z)=min(x,min(y,z)) = min(x, y, z), merge(x,y) = merge(y,x) and merge(merge(x,y),z)=merge(x,merge(y,z)) = merge(x, y, z), thus the process of merging qualities is commutative and associative.

Proof of Associative Property of Minimum:

If we want to prove min(min(x,y),z)=min(x,min(y,z)) = min(x, y, z), there are the following cases to consider:

(1):$x \leqslant y \leqslant z$

(2):$x \leqslant z \leqslant y$

(3):$y \leqslant x \leqslant z$

(4):$y \leqslant z \leqslant x$

(5):$z \leqslant x \leqslant y$

(6):$z \leqslant y \leqslant x$

(1) min(min(x,y),z)=x   min(x,min(y,z)) =x   min(x, y, z)=x

(2) min(min(x,y),z)=x   min(x,min(y,z)) =x   min(x, y, z)=x

(3) min(min(x,y),z)=y   min(x,min(y,z)) =y   min(x, y, z)=y

(4) min(min(x,y),z)=y   min(x,min(y,z)) =y   min(x, y, z)=y

(5) min(min(x,y),z)=z   min(x,min(y,z)) =z   min(x, y, z)=z

(6) min(min(x,y),z)=z   min(x,min(y,z)) =z   min(x, y, z)=z

Thus in all cases it can be seen that the result holds.

Proof of Commutative Property of Minimum:

If we want to prove min(x,y) = min(y,x), there are the following cases to consider:

(1): $x<y$

(2): $x=y$

(3): $x>y$

(1) min(x,y) =x   min(y,x)=x

(2) min(x,y) =x   min(y,x)=x

(3) min(x,y) =y   min(y,x)=y

Thus in all cases it can be seen that the result holds.

4.3 Specific Implements in PostgreSQL

This chapter demonstrates each specific modification in PostgreSQL to implement the quality models "Quality" and new SQL algebra "Quality Algebra" that has been illustrated in Chapter 4.2.

4.3.1 src/backend/parser/parser.c

File "parser.c" can be regarded as main entry point and driver for PostgreSQL grammar. Table 4.5 and 4.6 below describe each modification in the existed function and additional helper functions.

Function Name	Function Prototype
raw_parser	List *raw_parser(const char *str)
str_replace	char *str_replace (const char *source, char *find, char *rep)
useseqscan	int useseqscan(const char* string)
findfrom	char *findfrom(const char* string)
split	void split(char **arr, char *str, const char *del)
helpmerge	char *helpmerge(char *str)

Table 4.5 Modified Functions and Corresponding Prototypes

Function Name	Description
raw_parser	Raw_parser is an existed function inside parser.c file. When a query has been given in string form (const char *str), this function does lexical and grammatical analysis, as the result, it returns a list of “raw” parser trees (need to be analyzed by “analyze.c” and related files). Here I add three check conditions for input SQL command: 1. If this command is used for creating a table, then raw_parser adds a new attribute called “Quality”, which is an array of QNode (defined in JoinHelper.sql), at the end of table. As a result, when users create a table, the quality model can be automatically added as an additional attribute. e.g., “CREATE TABLE student (sid int, sname text, sgender char(1)); “ →“create table student (sid int, sname text, sgender char(1), Quality QNode[]); “ 2. If this command is used for selecting attributes in a table, then raw_parser adds attribute “Quality” at the end of SELECT statement. As a result, the quality information can be printed out at tuple level. e.g., “SELECT sname FROM student WHERE sid=1;” →“SELECT sname, Quality FROM student WHERE sid=1;” 3. If this command is used for joining several tables (e.g., table$_1$, table$_2$,……,table$_n$), then raw_parser adds “merge*n*(table$_1$.Quality, table$_2$.Quality,……,table$_n$.Quality) as quality” at the end of SELECT statement (merge*n* is defined in JoinHelper.sql). As a result, the quality of the result of join operation can be printed out at tuple level. e.g., “SELECT sname, cname FROM student,participantin WHERE student.sid = participantin.sid;” → “SELECT sname, cname, merge2(student.Quality, participantin.Quality) AS quality FROM student,participantin WHERE student.sid = participantin.sid;”
str_replace	str_replace is a helper function. If in the string *source*, there is a substring *find*, then str_replace will replace substring *find* with substring *rep* and return the new string.

	For example, str_replace("This is a table.", "table", "chair") gives the result "This is a chair.".
useseqscan	Useseqscan is a helper function. If in the SQL command *string*, there is only one table in the FROM statement, then useseqscan return 0, otherwise return 1. For example, useseqscan("SELECT sname FROM student WHERE sid=1;") gives the result 0, because there is only one table in the FROM statement which is "student" here. Useseqscan("SELECT sname, cname FROM student,participantin WHERE student.sid = participantin.sid;") gives the result 1, because there is more than one table in the FROM statement which are "student" and "participantin" here;
findfrom	Findfrom is a helper function. It extracts the tables' names in the FROM ststement from a given SQL command. For example, findfrom("SELECT sname FROM student WHERE sid=1;") returns "student" findfrom("SELECT sname, cname FROM student, participantin WHERE student.sid = participantin.sid;") returns "student, participantin".
split	Split is a helper function. Given a source string *str* , a string *del* as delimitation and an array of string, this function split the source string *str* into several substrings based on *del*, and then store each substring into given array *arr* and return this array *arr*. For example, split ("student, professor, course") returns {"student","professor","course"}.
helpmerge	Helpmerge is a helper function, given a string *str* that contains n tables' names from FROM statement, this function generates the corresponding merge*n* function statement for these n tables and return the statement generated. For example, helpmerge("*table1, table2, table3*") returns ", merge3(*table1*.Quality, *table2*.Quality, *table3*.Quality) as quality from".

Table 4.6 Modified Functions and Corresponding Descriptions

4.3.2 src/include/parser/parser.h

Parser.h contains the definitions for the "raw" parser (flex and bison phases only) and the prototype from each functions in parser.c. Because several additional helper functions has been added in parser.c file, this header file should include the prototypes of each additional helper functions.

These prototypes are:

```
extern char *str_replace (const char *source, char *find,  char *rep);
extern int useseqscan(const char* string);
extern char *findfrom(const char* string);
extern void split(char **arr, char *str, const char *del);
extern char *helpmerge(char *str);.
```

4.3.3 JoinHelper.sql

JoinHelper.sql should be run when users create a new database. It creates new data type "QNode" to implement the quality model and creates functions merge*n* to merge qualities from each tuple that need to be joined together and return the result quality. Figure 4.7 below shows the logic of merge2 function which merge the qualities from two tuples and return the result quality.

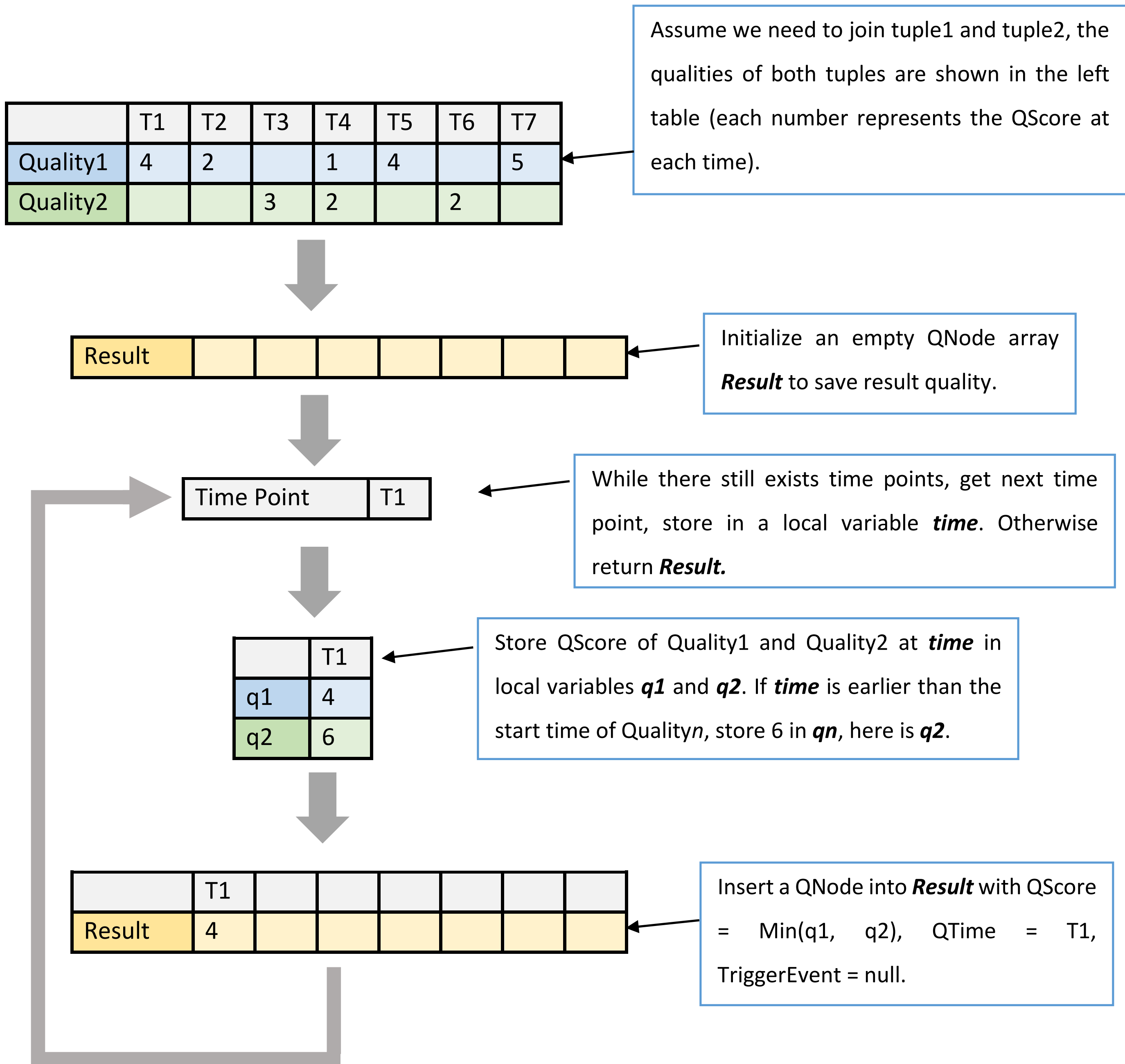

Figure 4.7 Logic of merge2 Function

Besides function merge2, there are merge3, merge4 and merge5 which merge the qualities from three, four and five tuples and return the result quality. Except merge2, the rest of functions use merge*n-1* to process n-1 tuples and use merge2 to merge the result quality from n-1 tuples and the n_{th} tuple. Because of the associative and

commutative properties illustrated in Chapter 4.3.3, the logics of merge*3 – 5* work. For example, the Figure 4.8 below shows the logic of merge4.

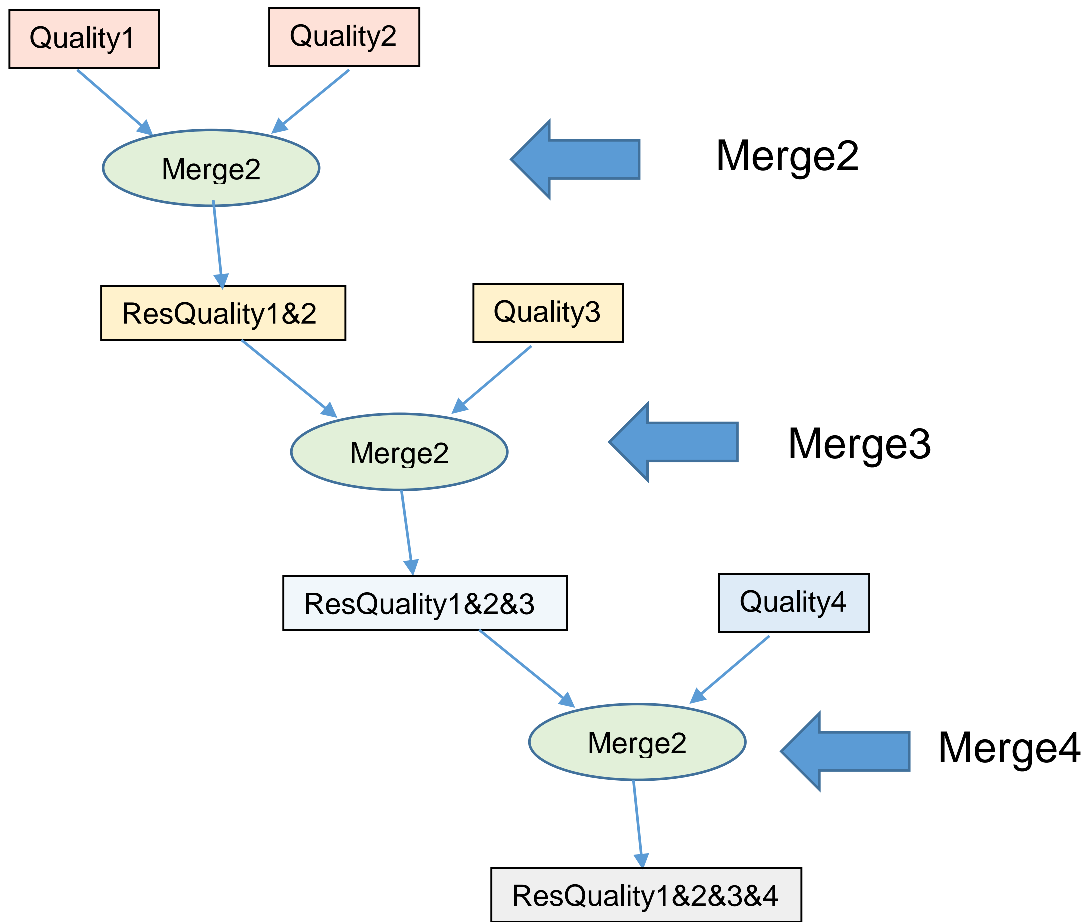

Figure 4.8 Logic of Function merge4

Chapter 5 Results and Analysis

This chapter focuses on the testing and analysis of performance of Quality-Aware database. As mentioned in the previous chapters, Quality-Aware database introduces a new quality model that captures the evolution of the data's qualities over time and a new query algebra focused on select and join operators that enables seamless and transparent propagation and derivations of the data's qualities within a query pipeline. So, this chapter utilizes two tables with several different size of inserted data sets to test the performance of selection and join operation with/ without quality model.

5.1 Description of Test Tables

Table "supplier" and table "product" are created as the test tables to test the performance of quality-aware database. Following Table 5.1 and 5.2 show the scheme of each table and description of each attribute.

Attribute	Type	Description
Sid	Integer	A unique supplier id number for each supplier.
Sname	Text	Name of supplier.
Age	Integer	Age of supplier.
Gender	Character(1)	Gender of supplier. ("F" stands for female and "M" stands for male)
Address	Text	Address of supplier.

Table 5.1 Description of Table "supplier"

Attribute	Type	Description
Pid	Integer	A unique product id number for each product.
Sid	Integer	Supplier id number for who supply this product.
Pname	Text	Name of product.
PType	Text	Type of product.
Instock	Character(1)	Whether this product is in stock or not. ("T" stands for in stock and "F" stands for out of stock)

Table 5.2 Description of Table "product"

5.2 Process of Testing

There are three different size of “supplier” data sets (with and without quality): 10 000 records, 50 000 records and 100 000 records and one fixed size of “product” data set (with and without quality): 50 000 records. In this project, the performance of an operation, which is need to be analyzed, is determined by the execution time. In other words, short execution time means great performance and long execution time means poor performance. Detailed process of testing shows below.

Step1 Create table “supplier” and “product” (refer to Chapter 5.1)

Step2 Create 8 SQL files to insert different sizes and different kinds of data.

1. File1: 10,000 supplier data (with quality)
2. File2: 50,000 supplier data (with quality)
3. File3:100,000 supplier data (with quality)
4. File4: 50,000 product data (with quality)
5. File5: 10,000 supplier data (without quality)
6. File6: 50,000 supplier data (without quality)
7. File7: 100,000 supplier data (without quality)
8. File8: 50,000 product data (without quality)

Step3 Execute File1 to insert first data set into “supplier” table.

Step4 Input command “**EXPLAIN ANALYZE SELECT sid, sname FROM supplier WHERE age>30;**”, record the execution time of selection for three times (after recording current execution time, log off system and log in, then run the command to record the next execution time).

Step5 Delete data in the table “supplier” using “**DELETE FROM supplier;**”, prepare for insertion next time.

Step6 Repeat **Step3, 4, 5** for File 2, repeat **Step3, 4** for File3.

Step7 Execute File4 to insert first data set into “product” table.

Step8 Input command “**EXPLAIN ANALYZE SELECT pname, sname FROM supplier,product WHERE product.sid = supplier.sid;**”, record the execution time of join operation for three times (after recording current execution time, log off system and log in, then run the command to record the next execution time).

Step9 Make the size of data in the table "supplier" equal 50,000 by using "**DELETE FROM supplier WHERE sid>=50000;".**

Step10 Repeat **Step8.**

Step11 Make the size of data in the table "supplier" equal 10,000 by using "**DELETE FROM supplier WHERE sid>=10000;".**

Step12 Repeat **Step8.**

Step13 Remove quality model and query algebra. Change to original RDBMS.

Step14 Repeat **Step3~12** for File5, 6, 7, 8.

Step15 Add quality model and query algebra. Change back to quality-aware database.

5.3 Result of Testing

Selection with Quality:

	First time (ms)	**Second time (ms)**	**Third time (ms)**	**Average time(ms) (First time + Second time + Third time)/3**
10,000 records	5.554	5.656	5.592	5.601
50,000 records	20.613	20.525	20.539	20.559
100,000 records	41.190	39.700	42.654	41.181

Table 5.3 Testing Result of Selection with Quality

Selection without Quality

	First time (ms)	**Second time (ms)**	**Third time (ms)**	**Average time(ms) (First time + Second time + Third time)/3**
10,000 records	3.636	3.861	3.893	3.797
50,000 records	20.082	21.274	20.632	20.663
100,000 records	42.437	40.686	40.863	41.329

Table 5.4 Testing Result of Selection without Quality

Join with Quality

	First time (ms)	Second time (ms)	Third time (ms)	Average time(ms) (First time + Second time + Third time)/3
10,000 records	1403.701	1392.092	1399.189	1398.327
50,000 records	2913.619	2938.804	2937.110	2929.844
100,000 records	2964.514	2994.645	2979.916	2979.692

Table 5.5 Testing Result of Join with Quality

Join without Quality

	First time (ms)	Second time (ms)	Third time (ms)	Average time(ms) (First time + Second time + Third time)/3
10,000 records	56.652	54.110	57.321	55.998
50,000 records	105.992	107.973	105.191	106.385
100,000 records	138.540	137.221	139.022	138.261

Table 5.6 Testing Result of Join without Quality

5.4 Analysis of Result

I used two vertical bar charts to display the average execution time of selection and join operator because:

1. Vertical bar chart is easy to compare sets of data between different groups at a glance
2. Vertical bar chart is easy to see the relationship of the data between the x and y axes
3. Vertical bar chart is effective to present trends or changes over time

In each bar chart there are two series: *With Quality* and *Without Quality*, and each series has three categories: *10,000 RECORDS, 50,000 RECORDS* and *100,000 REOCRDS*. The resource of data in Figure 5.7: last column (*Average time)* in Table 5.3 – 5.4 in Chapter 5.2. The resource of data in Figure 5.8: last column (*Average time)* in Table 5.5 – 5.6 in Chapter 5.2.

Figure 5.7 Average Execution Time of Selection

From Figure5.7 above, we can find: with the increase of the records, the execution time of selection in both database systems also increase. When there are 10,000 records, the execution of selection in quality-aware database system (selection with quality) is slower than in original database system (selection without quality). The reason is that the size of 10,000 records in quality-aware database system is larger than in original database system, because as mentioned in Chapter 4.1, besides original data values, each data tuple in quality-aware database additionally carries a “Quality”. But if the number of records increase to 50,000 or more than 50,000, there is no significant difference between the execution time of quality-aware database system and original database

system. Because when databases deal with large size of data, the execution time mainly contributes to carrying data from cache to main memory.

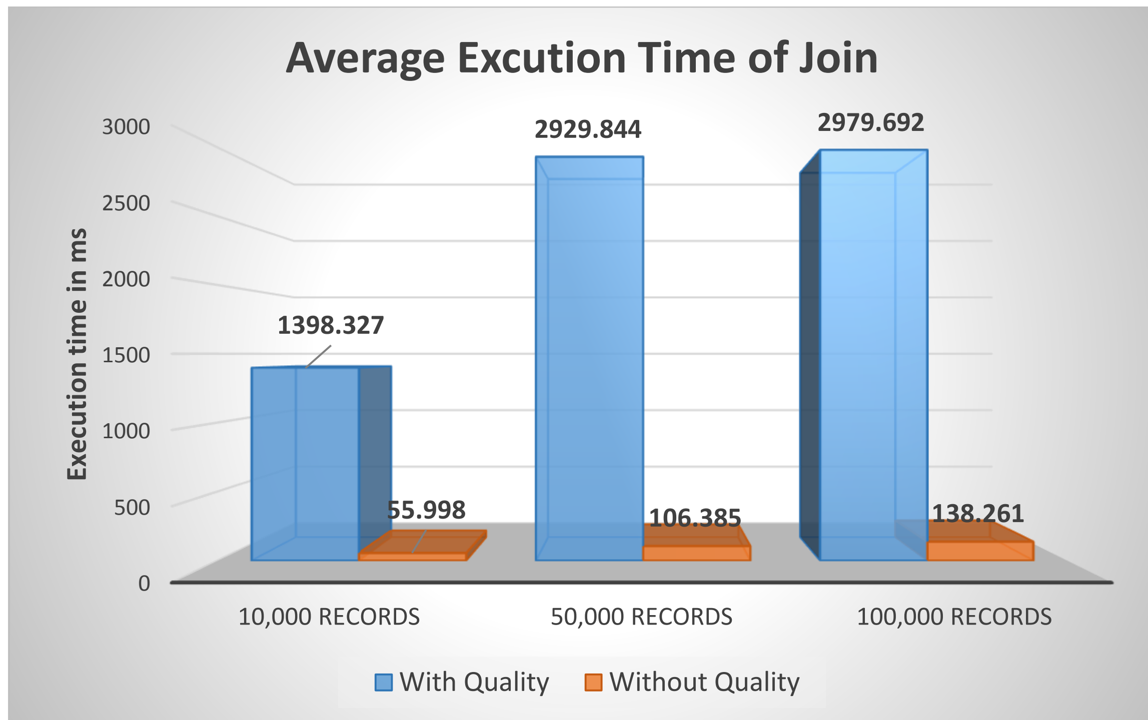

Figure 5.8 Average Execution Time of Join

From Figure5.8 above, we can find that: with the increase of the records, the execution time of join operator in both database systems also increase. The execution of join in quality-aware database system (selection with quality) is much slower than in original database system (selection without quality). Because when quality-aware database system processes selection, it just need to print out one additional attribute, however, when it processes join operator, it need to call *joinhelper.sql* to deal with several qualities, obtain the result quality of join result and print it out. In order to keep track of where the execution time of join in quality-aware database is going, I commented out each step in function *merge2*, compiled, ran the join command "**EXPLAIN ANALYZE SELECT pname, sname FROM supplier,product WHERE product.sid = supplier.sid;**", recorded the execution time for three sizes of record (10 000, 50 000 and 100 000) and calculated the average execution time among these sizes for each step. Finally used a pie chart (see Figure 5.9) to display. It can be seen from this pie chart that getting each time

point takes 24% of total time; after that, finding the QScores in both qualities at this time point takes 33% of total time, which is the most expensive step in *merge2*; then, merge QScores and construct result Quality at this time point takes 25% of total time.

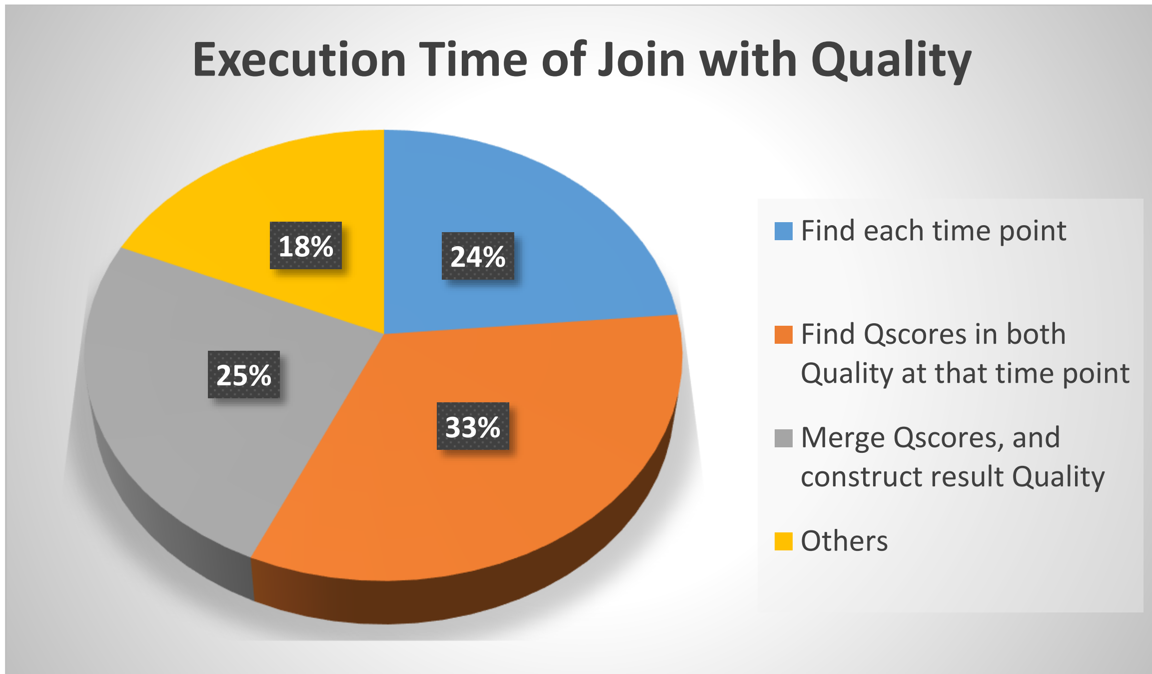

Figure 5.9 Execution Time of Join with Quality

Chapter 6 Conclusion

In this project, a quality-aware database system has been created to deal with the imperfect database. This quality-aware database fundamentally extends PostgreSQL and contains a new quality model—"Quality" to capture the evolution of the data's qualities over time and a new query algebra—"Quality Algebra" for propagation and derivations of the data's qualities when do selection and join operation.

This project introduced the background research about database system and imperfect database, the harm of low quality data and PostgreSQL. After that, some related works has been included, which provide different techniques to handle data-quality problems. Next, this project took a closer look at new data model —"Quality" and new query algebra—"Quality Algebra" with illustration of each specific implement in the PostgreSQL database system. Finally, a test has been conducted to compare the performance of quality-aware database system and standard database system, and an analysis of the test result has been included in this project.

Reference

[1] Anh Pham, Glusher Kooner, Mohamed Y. Eltabakh. QTrail-DB: A Query Processing Engine for Imperfect Databases with Evolving Qualities

[2] A. Arasu, C. Ré, and D. Suciu. Large-Scale Deduplication with Constraints Using Dedupalog. In ICDE, pages 952–963, 2009.

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

[4] C. Batini and M. Scannapieco. Data Quality: Concepts, Methodologies and Techniques. Addison-Wesley, 2006.

[5] G. Beskales, M. A. Soliman, I. F. Ilyas, and S. Ben-David. Modeling and Querying Possible Repairs in Duplicate Detection. Proc. VLDB Endow., 2(1):598–609, 2009.

[6] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A Cost-based Model and Effective Heuristic for Repairing Constraints by Value Modification. In Proceedings of ACM SIGMOD International Conference on Management of Data, pages 143–154, 2005.

[7] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional Functional Dependencies for Data Cleaning. In ICDE, pages 746–755, 2007.

[8] L. Bravo, W. Fan, and S. Ma. Extending Dependencies with Conditions. In Proceedings of the 33rd International Conference on Very Large Data Bases, pages 243–254, 2007.

[9] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In SIGMOD, pages 539–550, 2006.

[10] P. Buneman, S. Khanna, and W. Tan. Why and where: A characterization of data provenance. Lec. Notes in Comp. Sci., 1973:316–333, 2001.

[11] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving Data Quality: Consistency and Accuracy. In Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB), pages 315–326, 2007.

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

[13] A. Ebaid, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, J.-A. Quiane-Ruiz, N.Tang, and S. Yin. NADEEF: A Generalized Data Cleaning System. Proc. VLDB Endow., 6(12):1218–1221, 2013.

[14] W. Eckerson. Data Quality and the Bottom Line: Achieving Business Success through a Commitment to High Quality Data. In The Data Warehousing Institute, 2002.

[15] H. Galhardas, A. Lopes, and E. Santos. Support for User Involvement in Data Cleaning. In Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery, pages 136–151, 2011.

[16] J. Galindo, A. Urrutia, and M. Piattini. Fuzzy databases: Modeling, design, and implementation. Idea Group Publishing, 2006.

[17] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC Data-cleaning Framework. Proc. VLDB Endow., 6(9):625–636, 2013.

[18] J. Han, D. Jiang, and L. Li. Automatic Accuracy Assessment via Hashing in Multiple-source Environment. Journal of Expert Systems with Applications, 37(3):2609–2620, 2010.

[19] A. Haug, F. Zachariassen, and D. van Liempd. The costs of poor data quality. Journal of Industrial Engineering and Management, 4(2):168–193, 2011.

[20] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying Data Provenance. In SIGMOD, pages 951–962, 2010.

[21] A. Klein and W. Lehner. Representing Data Quality in Sensor Data Streaming Environments. Journal of Data and Information Quality, 1(2):10:1–10:28, 2009.

[22] A. Lopatenko and L. Bravo. Efficient Approximation Algorithms for Repairing Inconsistent Databases. In ICDE, pages 216–225, 2007.

[23] A. Motro and I. Rakov. Estimating the Quality of Data in Relational Databases. In In Proceedings of the 1996 Conference on Information Quality, pages 94–106. MIT, 1996.

[24] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental Maintenance for Non-Distributive Aggregate Functions. In VLDB, pages 802–813, 2002.

[25] R. K. Pon and A. F. Cardenas. Data quality inference. In International Workshop on Information Quality in Information Systems (IQIS), pages 105–111, 2005.

[26] M. N. Radziwill. Foundations for Quality Management of Scientific Data Products. Quality Management Journal, 13(2):7–21, 2006.

[27] T. C. Redman. The Impact of Poor Data Quality on the Typical Enterprise. Commun. ACM, 41(2):79–82, 1998.

[28] S. Singh and et al. Database support for probabilistic attributes and tuples. ICDE, pages 1053–1061, 2008.

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

[30] Y. Su, D. Li, and J. Peng. Modeling Information Quality Risk in Data Mining. In Wireless Communications, Networking and Mobile Computing (WiCOM), pages 1–4, 2008.

[31] M. Twombly. Science online survey: Support for data curation. Science Journal, 331, 2011.

[32] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. CIDR, pages 262–276, 2005.

[33] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided Data Repair. Proc. VLDB Endow., 4(5):279–289, 2011.

[34] E. Zimnyi and A. Pirotte. Imperfect Information in Relational Databases. In Uncertainty Management in Information Systems, pages 35–87. Springer US, 1997.

[35] English, L. (1998, January 1). The High Costs of Low-Quality Data. Retrieved from http://www.information-management.com/issues/19980101/771-1.html

[36] Pratte, D. (2001, May 30). Poor-quality data can rob your company of information. Retrieved from http://www.techrepublic.com/article/poor-quality-data-can-rob-your-company-of-information/

[37] McKnight, W. (2009, June). 7 Sources of Poor Data Quality. Retrieved from https://www.melissadata.com/enews/articles/0611/2.html

[38] Ramakrishnan, R., & Gehrke, J. (1997). Database management system (Second ed.). New York, N.Y.: McGraw-Hill, Page 3 -10

[39] Motro, A., & Smets, P. (1997). Uncertainty management in information systems: From needs to solutions. Boston: Kluwer Academic, Page 35-87

[40] Motro, A. IMPRECISION AND UNCERTAINTY IN DATABASE SYSTEMS (pp. 5-17, Rep.).

[41] THE TRUE COST OF BAD DATA. (n.d.). Retrieved from http://lemonly.com/work/the-cost-of-bad-data/

[42] Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom. Database Systems: The Complete Book. Second ed. Upper Saddle River, NJ: Prentice Hall, 2002. Print. Page1-3

[43] "Max and Min Are Associative." - ProofWiki. Web. <https://proofwiki.org/wiki/Max_and_Min_are_Associative>.

[44] "Max and Min Are Commutative." - ProofWiki. Web. <https://proofwiki.org/wiki/Max_and_Min_are_Commutative>.

[45] "Bar Graph." Smartdraw. Web. <https://www.smartdraw.com/bar-graph/>.

[46] "PostgreSQL." Wikipedia. Wikimedia Foundation. Web. <https://en.wikipedia.org/wiki/PostgreSQL>.

[47] "Backend Flowchart." PostgreSQL: Backend Flowchart. The PostgreSQL Global Development Group. Web. <http://www.postgresql.org/developer/backend/>.

Appendix – Implemented Functions' Code

src/backend/parser/parser.c:

```
char *str_replace (const char *source, char *find,  char *rep){
  int find_L=strlen(find);
  int rep_L=strlen(rep);
  int length=strlen(source)+1;
  int gap=0;
  const char *former=source;
  char *result = (char*)malloc(sizeof(char) * length);
  char *location= strstr(former, find);
  strcpy(result, source);
     gap+=(location - former);
     result[gap]='\0';
     length+=(rep_L-find_L);
            result = (char*)realloc(result, length * sizeof(char));
     strcat(result, rep);
     gap+=rep_L;
     former=location+find_L;
     strcat(result, former);
  return result;
}

int useseqscan(const char* string){
 char *sub1 = "where";
 char *sub2 = "group by";
 char *sub3 = "order by";
 char *sub4 = ",";
 int length=strlen(string)+1;
 char *fromsentense = (char*)malloc(sizeof(char) * length);
 const char *total = str_replace(string, "from","from /");

if (strstr(string, sub1) != NULL) {
const char *final = str_replace(total, "where","@ where");
printf("%s\n", final);
sscanf(final, "%*[^/]/%[^@]", fromsentense);
printf("%s\n", fromsentense);
}
else if (strstr(string, sub2) != NULL) {
const char *final = str_replace(total, "group","@ group");
printf("%s\n", final);
sscanf(final, "%*[^/]/%[^@]", fromsentense);
printf("%s\n", fromsentense);
}
else if (strstr(string, sub3) != NULL) {
const char *final = str_replace(total, "order","@ order");
printf("%s\n", final);
sscanf(final, "%*[^/]/%[^@]", fromsentense);
printf("%s\n", fromsentense);
}
else {
```

```
const char *final = str_replace(total, ";","@ ;");
printf("%s\n", final);
sscanf(final, "%*[^/]/%[^@]", fromsentense);
printf("%s\n", fromsentense);
}

if (strstr(fromsentense, sub4) != NULL) {
return 1;
}
return 0;
}

char *findfrom(const char* string){
  char *sub1 = "where";
  char *sub2 = "group by";
  char *sub3 = "order by";
  char *sub4 = ",";
  int length=strlen(string)+1;
  char *fromsentense = (char*)malloc(sizeof(char) * length);
  const char *total = str_replace(string, "from","from /");

if (strstr(string, sub1) != NULL) {
const char *final = str_replace(total, "where","@ where");
sscanf(final, "%*[^/]/%[^@]", fromsentense);
}
else if (strstr(string, sub2) != NULL) {
const char *final = str_replace(total, "group","@ group");
sscanf(final, "%*[^/]/%[^@]", fromsentense);
}
else if (strstr(string, sub3) != NULL) {
const char *final = str_replace(total, "order","@ order");
sscanf(final, "%*[^/]/%[^@]", fromsentense);
}
else {
const char *final = str_replace(total, ";","@ ;");
sscanf(final, "%*[^/]/%[^@]", fromsentense);
}

return fromsentense;
}

void split(char **arr, char *str, const char *del) {
        char *sub1 = "where";
        char *sub2 = "group";
        char *sub3 = "order";
        char *sub4 = ";";
  char *s = strtok(str, del);

  while((s != NULL) && (strstr(s, sub1) == NULL) && (strstr(s, sub2) == NULL) && (strstr(s, sub3) == NULL)
&& (strstr(s, sub4) == NULL)) {
    *arr++ = s;
    s = strtok(NULL, del);
  }
```

```
}

char *helpmerge(char *str){
			int i = 0;
			int j = 0;
			char* tnum;
			const char *delim = ",";
			char *tablename[5];
			while(i<5){
			tablename[i] = "null";
			i++;
			}
			char *total;
			const char *s1 = ", merge";
			const char *s2 = " (" ;
			const char *s3 = ".Quality";
			const char *s4 = ") as quality from" ;
			const char *s5 = ",";
			split(tablename, str, delim);
			i=0;
			while (strcmp(tablename[i], "null")!=0){
			i++;}
			if (i==1){
			tnum = "1";}
			if (i==2){
			tnum = "2";}
			if (i==3){
			tnum = "3";}
			if (i==4){
			tnum = "4";}
			if (i==5){
			tnum = "5";}
			total =
(char*)malloc(strlen(s1)*sizeof(char)+strlen(s2)*sizeof(char)+strlen(s3)*9*sizeof(char)+strlen(s4)*sizeof(c
har)+strlen(s5)*sizeof(char)+strlen(str)*sizeof(char)+sizeof(char));
			strcat(total, s1);
			strcat(total, tnum);
			strcat(total, s2);
			while(j<i-1){
			strcat(total, tablename[j]);
			strcat(total, s3);
			strcat(total, s5);
			j++;
			}
			strcat(total, tablename[i-1]);
			strcat(total, s3);
			strcat(total, s4);
			return total;
}

/*
 * raw_parser
 *			Given a query in string form, do lexical and grammatical analysis.
```

```
 *
 * Returns a list of raw (un-analyzed) parse trees.
 */
List *
raw_parser(const char *str)
{
	core_yyscan_t yyscanner;
	base_yy_extra_type yyextra;
	int			yyresult;
	char *substr = "create table";
	char *substr2 = "select";
	if (strstr(str, substr) != NULL) {
	const char *total = str_replace(str, ");",", Quality QNode[]);");
		/* initialize the flex scanner using modified string*/
		yyscanner = scanner_init((const char *)total, &yyextra.core_yy_extra,ScanKeywords,
NumScanKeywords);
		}
	else if (strstr(str, substr2) != NULL && useseqscan(str) == 0) {
	const char *total = str_replace(str, "from",", Quality from");
		/* initialize the flex scanner using modified string*/
		yyscanner = scanner_init((const char *)total, &yyextra.core_yy_extra,ScanKeywords,
NumScanKeywords);
		}
	else if (strstr(str, substr2) != NULL && useseqscan(str) != 0) {
	char *from = findfrom(str);
	char *merge = helpmerge(from);
	const char *total = str_replace(str, "from", merge);
		/* initialize the flex scanner using modified string*/
		yyscanner = scanner_init((const char *)total, &yyextra.core_yy_extra,ScanKeywords,
NumScanKeywords);
		}
	else {
	/* initialize the flex scanner */
	yyscanner = scanner_init(str, &yyextra.core_yy_extra,ScanKeywords, NumScanKeywords); }

	/* base_yylex() only needs this much initialization */
	yyextra.have_lookahead = false;

	/* initialize the bison parser */
	parser_init(&yyextra);

	/* Parse! */
	yyresult = base_yyparse(yyscanner);

	/* Clean up (release memory) */
	scanner_finish(yyscanner);

	if (yyresult)				/* error */
		return NIL;

	return yyextra.parsetree;
}
```

src/include/parser/parser.h:

```
extern char *str_replace (const char *source, char *find,  char *rep);
extern int useseqscan(const char* string);
extern char *findfrom(const char* string);
extern void split(char **arr, char *str, const char *del);
extern char *helpmerge(char *str);
```

JoinHelper.sql:

```
CREATE TYPE QNode AS (QScore int, QTime timestamp, TriggerEvent text);


CREATE OR REPLACE FUNCTION merge2 (arr1 QNode[], arr2 QNode[]) RETURNS QNode[] AS $$
        DECLARE
                q1 int;
                q2 int;
                size1 int;
                size2 int;
                sizet int;
                time timestamp;
                i1    int;
                i2        int;
                ir        int;
                i11       int;
                i22       int;
                res       QNode[];
                temp QNode;
    BEGIN
                i1 := 1;
                i2 := 1;
                ir := 1;
                i11 := 1;
                i22 := 1;
                size1 := array_length(arr1, 1);
                size2 := array_length(arr2, 1);

                while i1<=size1 and i2<=size2 loop
                        if arr1[i1].QTime < arr2[i2].QTime then
                                time := arr1[i1].QTime;
                                i1:=i1+1;
                        elsif arr1[i1].QTime = arr2[i2].QTime then
                                time := arr1[i1].QTime;

                                i1:=i1+1;
                                i2:=i2+1;
                        else time:= arr2[i2].QTime;

                                i2:=i2+1;
                        end if;
```

```
		if arr1[i11].QTime=time then
			q1 := arr1[i11].Qscore;
			i11 := i11+1;
			if i11>size1 then
				i11:=i11-1;
			end if;
		elsif arr1[i11].QTime>time then
			if i11=1 then
				q1 := 6;
			else	q1 := arr1[i11-1].Qscore;

			end if;
		else q1 := arr1[i11].Qscore;
		end if;

		if arr2[i22].QTime=time then
			q2 := arr2[i22].QScore;
			i22 := i22+1;
			if i22>size2 then
				i22:=i22-1;
			end if;
		elsif arr2[i22].QTime>time then
			if i22=1 then
				q2 := 6;

			else	q2 := arr2[i22-1].Qscore;

			end if;
		else q2 := arr2[i22].Qscore;
		end if;

		temp.Qscore := LEAST(q1, q2);
		temp.QTime := time;
		temp.TriggerEvent := null;
		res[ir] := temp;
		ir :=ir+1;
	end loop;

	if i1-1=size1 and i2-1<size2 then
		while i2<=size2 loop
			time := arr2[i2].QTime;
			i2:=i2+1;
		if arr1[i11].QTime=time then
			q1 := arr1[i11].Qscore;
			i11 := i11+1;
			if i11>size1 then
				i11:=i11-1;
			end if;
		elsif arr1[i11].QTime>time then
			if i11=1 then
				q1 := 6;
			else	q1 := arr1[i11-1].Qscore;
```

```
				end if;
			else q1 := arr1[i11].Qscore;
			end if;

			if arr2[i22].QTime=time then
				q2 := arr2[i22].QScore;
				i22 := i22+1;
				if i22>size2 then
					i22:=i22-1;
				end if;
			elsif arr2[i22].QTime>time then
				if i22=1 then
					q2 := 6;


				else	q2 := arr2[i22-1].Qscore;

				end if;
			else q2 := arr2[i22].Qscore;
			end if;

			temp.Qscore := LEAST(q1, q2);
			temp.QTime := time;
			temp.TriggerEvent := null;
			res[ir] := temp;
			ir :=ir+1;
			end loop;
		elsif i2-1=size2 and i1-1<size1 then
			while i1<=size1 loop
				time := arr1[i1].QTime;
				i1:=i1+1;


			if arr1[i11].QTime=time then
				q1 := arr1[i11].Qscore;
				i11 := i11+1;
				if i11>size1 then
					i11:=i11-1;
				end if;
			elsif arr1[i11].QTime>time then
				if i11=1 then
					q1 := 6;
				else	q1 := arr1[i11-1].Qscore;

				end if;
			else q1 := arr1[i11].Qscore;
			end if;

			if arr2[i22].QTime=time then
				q2 := arr2[i22].QScore;
				i22 := i22+1;
				if i22>size2 then
					i22:=i22-1;
```

```
                        end if;
                elsif arr2[i22].QTime>time then
                        if i22=1 then
                                q2 := 6;

                        else    q2 := arr2[i22-1].Qscore;

                        end if;
                else q2 := arr2[i22].Qscore;
                end if;

                temp.Qscore := LEAST(q1, q2);
                temp.QTime := time;
                temp.TriggerEvent := null;
                res[ir] := temp;
                ir :=ir+1;
                end loop;
        end if;


    RETURN res;
  END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION merge3 (arr1 QNode[], arr2 QNode[], arr3 QNode[]) RETURNS QNode[]
AS $$
    DECLARE
        res QNode[];
  BEGIN
        res := merge2(arr1, arr2);
        res := merge2(arr3, res);
    RETURN res;
  END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION merge4 (arr1 QNode[], arr2 QNode[], arr3 QNode[]) RETURNS QNode[]
AS $$
    DECLARE
        res QNode[];
  BEGIN
        res := merge3(arr1, arr2);
        res := merge2(arr3, res);
    RETURN res;
  END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION merge5 (arr1 QNode[], arr2 QNode[], arr3 QNode[]) RETURNS QNode[]
AS $$
    DECLARE
        res QNode[];
  BEGIN
        res := merge4(arr1, arr2);
        res := merge2(arr3, res);
```

```
		RETURN res;
	END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION merge1 (arr1 QNode[]) RETURNS QNode[] AS $$
	BEGIN
		RETURN arr1;
	END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION merge (arr QNode[][]) RETURNS QNode[] AS $$
		DECLARE
				ct int;
				res QNode[];
	BEGIN
				ct := 3;
				res := merge2(arr[1], arr[2]);
				WHILE ct<array_length(arr,1) LOOP
						res := merge2(arr[ct], res);
						ct := ct+1;
				END LOOP;
		RETURN res;
	END;
$$ LANGUAGE plpgsql;
```