Chapter 3

Provenance Framework for Relational Databases

In various application domains such as data warehousing, data collaboration, scientific experiments, curated databases, transactional (OLTP) systems, workflow management etc., information about origin of any information is much needed. Most of the data in these kinds of applications are generated from existing data by applying number of transformations on that. To know about quality of data and some other decision making aspects such as what is the source of resultant data, why this data is generated, how this data is derived, who has created this data etc., fine-grained provenance information i.e. data provenance is required. *Data provenance* provides detailed information about a data element such as owner of data, its direct/indirect sources, history of data,transformations applied on data etc [2, 9, 13, 149, 96]. Provenance information can serves different purposes in relational database system such as incremental maintenance, update propagation in collaborative environment, audit trail, data discovery, and justification of a query result

- Asma Rani, Navneet Goyal, and Shashi K. Gadia. 2015. Data Provenance for Historical Queries in Relational Database. In Proceedings of the 8th Annual ACM India Conference (Compute '15). Association for Computing Machinery, New York, NY, USA, 117–122. DOI:https://doi.org/10.1145/2835043.2835047
- Asma Rani, Navneet Goyal, and Shashi K. Gadia. 2016. Efficient Multi-depth Querying on Provenance of Relational Queries Using Graph Database. In Proceedings of the 9th Annual ACM India Conference (COMPUTE '16). Association for Computing Machinery, New York, NY, USA, 11–20. DOI:https://doi.org/10.1145/2998476.2998480

etc. This can be beneficial in various application domains such as bio-informatics, data warehouse, data retention etc.

3.1 Provenance in Relational Databases

As the data in the world is growing at an unprecedented rate via different means like scientific applications, OLTP systems, data warehousing etc., data provenance has become an important topic of research. Data provenance is used to determine the veracity [103] and the quality [83] of data. Capturing, storing, and querying the provenance data is of paramount importance as it supports to trustworthiness [96], reliability, reputability, accountability, privacy, and quality [83] of data. In the context of scientific experiments, data provenance can be used in reproducing the experiments [124] and to determine the quality of work. It empowers the *auditing* process, and helps in *view maintenance* [12], *update propagation* [47, 48] without executing the query again. In addition to this, it also provides scope for data analysis with less efforts in terms of time and volume of data to be searched [103]. In business domain, it makes easier for the business to trust the data that is transferred between trusted partners [30], and helps in decision making or analyzing data. One of the most common examples of provenance information is *data citation*, where a reference to a previous publication is mentioned [9]. In the context of relational database, data lineage [12, 14] consists of relations, tuples, and attributes of source data along with operators used in transformations i.e. *Transformation Provenance* [86]. For large database applications, this life cycle can be quite tedious as data flows from various files, to database tables, views, and curated databases, while going through various transformation processes. So, tracking how a particular piece of data is derived becomes very challenging. Therefore, an efficient provenance capturing and querying framework in database systems plays a vital role in answering some decision making aspects such as *"What is the source of resultant data"*?, *"Why this data is generated"*?, *"How this data is derived"*?, *"Who has created this data"*?, *"History of data"*? etc.

Granularity of provenance information can be defined at the following two levels viz., *coarse-grained* and *fine-grained*. The first level tells us about which sequence of activities or operators are executed to generate a result set, and the second level gives more detailed information about which source tuples contributed to a piece of data in result set [30] respectively. In databases, data provenance is captured at fine-grained level as it is more significant and explanatory. Fine-grained provenance is further classified into three categories viz., *Where-Provenance* [13], *Why-Provenance* [13], and *How-Provenance* [46]. *Where-Provenance* identify the sources from where a value is copied into the result set as shown in Figure 1.6 in Chapter 1. It only tells us about the cells from where the value is coming, but not about the source tuples which are sufficient enough to generate the result tuple by executing the query again on provenance information. DB-Notes [27, 28], MONDRIAN [41], BDBMS [40, 62, 79], and MMS [53, 54, 55] relational provenance frameworks capture where-provenance via annotation propagation approach [30]. *Why-Provenance* captures, why a result tuple has been derived. It generates the source tuples which contributed to produce the result tuple, and are sufficient enough to produce the result tuple by executing the same query again on provenance information. But, it does not provide any information about the derivation process of a result tuple i.e. how these tuples in provenance information have contributed towards the result tuple generation. PERM [86] model captures why-provenance using query inversion approach [30]. *How-Provenance* captures the complete derivation history of a result tuple in a form of provenance polynomial. ORCHESTRA [29, 64, 47] framework captures how-provenance in collaborative environment using provenance semiring approach [46, 104]. *How-Provenance* is a superset of *Where* and *Why* Provenance. Figure 3.1 shows an example of Why and How Provenance. Instance of database shown in Figure 3.1 contains two tables i.e. *Part* and *Partsupp*. *Part* table contains information about all the Parts supplied, and *Partsupp* table contains the Supplier Id of every part supplied. An example query is issued on database to retrieve all the part names supplied by supplier with Supplier Id 2. Result of query is retrieved as "*Mouse*". We are curious to know why this value appears in result set and how it is derived? To answer these questions, we need to know about the provenance information. Figure 3.1 shows Why-Provenance as <P1,PS3> and How-Provenance as <P1*PS3>. Why-Provenance here shows tuple P1 and tuple PS3 are contributed towards result tuple generation. How-Provenance here shows that the join operation on tuple P1 and tuple PS3 is performed to generate the result tuple.

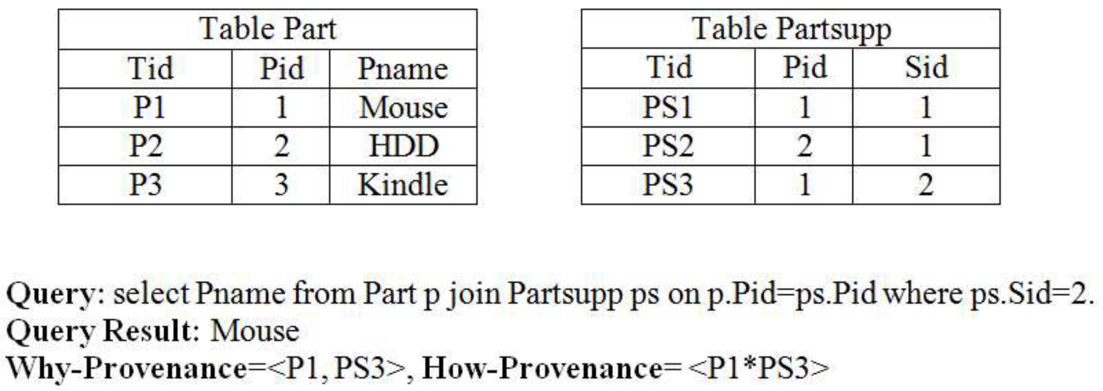


Table Part		
Tid	Pid	Pname
P1	1	Mouse
P2	2	HDD
P3	3	Kindle

Table Partsupp		
Tid	Pid	Sid
PS1	1	1
PS2	2	1
PS3	1	2

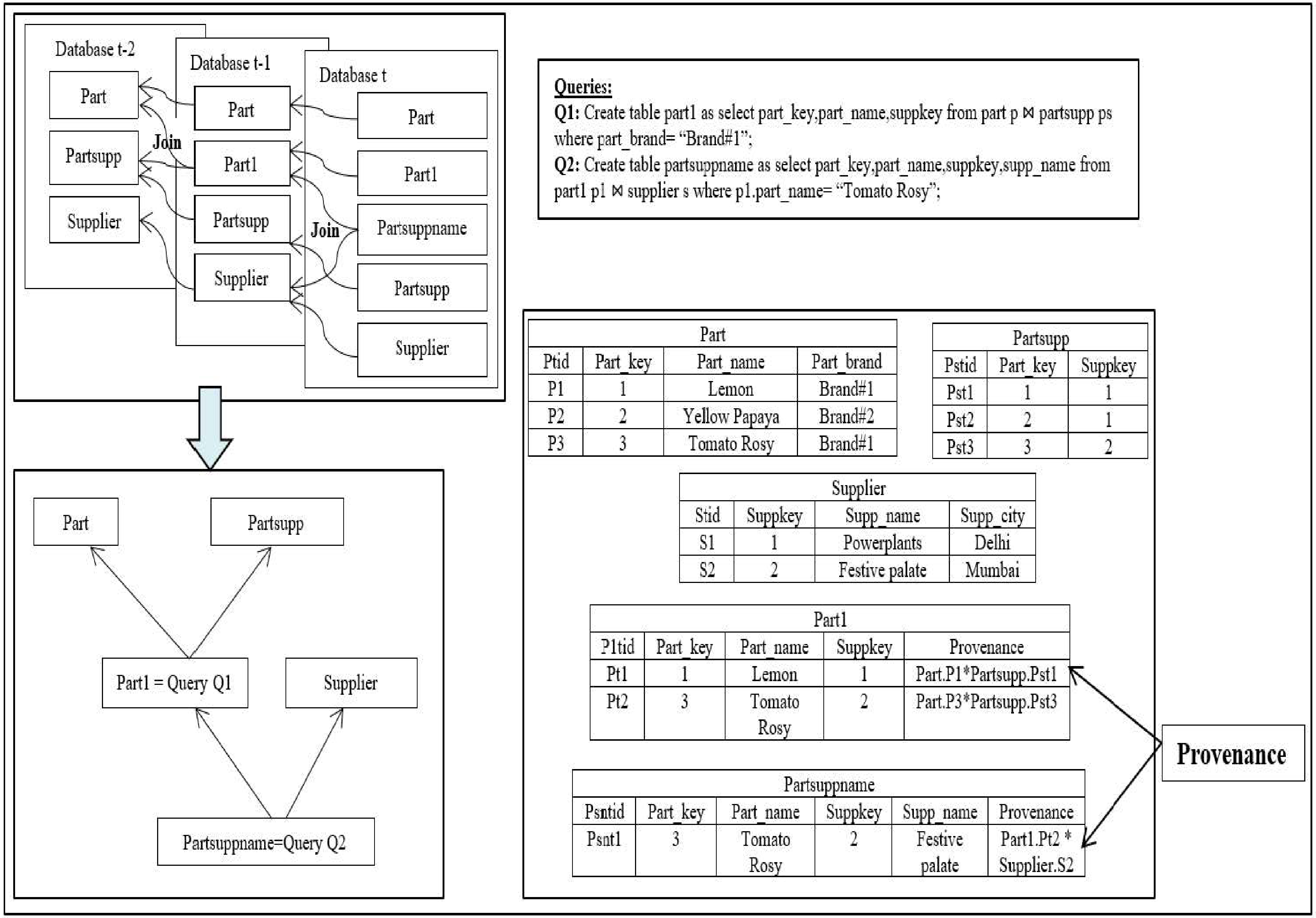
Query: select Pname from Part p join Partsupp ps on p.Pid=ps.Pid where ps.Sid=2.
Query Result: Mouse
Why-Provenance=<P1, PS3>, **How-Provenance**= <P1*PS3>

Figure 3.1: Example : Why and How Provenance

3.1.1 Multi-Layer/Multi-Depth Provenance

Tracking how a particular piece of data is derived either directly or indirectly becomes very challenging. It requires provenance capturing at various stages of the data life cycle. This necessitates the need for *Multi-Layer Provenance Capture* [31, 73] and efficient *Multi-Depth Provenance Querying*. Multi-Layer provenance captures more useful provenance information, such as immediate as well as intermediate sources and origin of data. It also provides information about where a piece of data has been used over time, either directly or indirectly. This is possible by querying on provenance data with varying traversal depths i.e. *Multi-Depth Provenance* querying. Multi-Depth provenance querying is useful in various database applications such as auditing, debugging [96, 103], error tracing, trustworthiness [48], quality assurance [9] etc.

Figure 3.2 shows an example of multi-layer provenance in relational database. Database instances at different times i.e. at time t-2, t-1, and t are shown. Queries Q1 and Q2 are executed on existing source tables to create new tables in database at different times. Initially, our database consists of three tables (source tables) viz., *Part*, *Partsupp*, and *Supplier* at time t-2. Instances of all these *source tables* are also shown in Figure 3.2. Afterwards, query Q1 is executed on source tables *Part* and *Partsupp* at database instance t-1, and in result of that result table *Part1* is generated. Further, at database instance t, another query Q2 is executed on tables *Part1* and *Supplier*, and corresponding result table *Partsuppname* is generated. Here, we can see that the table *Partsuppname* is created from table *Part1* which is further generated from tables *Part* and *Partsupp*. This shows Multi-Layer Prove-



Queries:
Q1: Create table part1 as select part_key,part_name,suppkey from part p ⋈ partsupp ps where part_brand= "Brand#1";
Q2: Create table partsuppname as select part_key,part_name,suppkey,supp_name from part1 p1 ⋈ supplier s where p1.part_name= "Tomato Rosy";

Part

Ptid	Part_key	Part_name	Part_brand
P1	1	Lemon	Brand#1
P2	2	Yellow Papaya	Brand#2
P3	3	Tomato Rosy	Brand#1

Partsupp

Pstid	Part_key	Suppkey
Pst1	1	1
Pst2	2	1
Pst3	3	2

Supplier

Stid	Suppkey	Supp_name	Supp_city
S1	1	Powerplants	Delhi
S2	2	Festive palate	Mumbai

Part1

P1tid	Part_key	Part_name	Suppkey	Provenance
Pt1	1	Lemon	1	Part.P1*Partsupp.Pst1
Pt2	3	Tomato Rosy	2	Part.P3*Partsupp.Pst3

Partsuppname

Psntid	Part_key	Part_name	Suppkey	Supp_name	Provenance
Psnt1	3	Tomato Rosy	2	Festive palate	Part1.Pt2 * Supplier.S2

Figure 3.2: Multi-layer Provenance Example

nance. Instances of all the *derived tables* i.e. *Part1* and *Partsuppname* including *Provenance Information*, i.e., which and how tuples from source tables are contributing to generate the resultant tuples, are shown in Figure 3.2. Here, we can see that the provenance of a tuple (Psnt1) in Partsuppname table is Part1.Pt2*Supplier.S2, i.e., tuple Pt2 of Part1 table and tuple S2 of Supplier table are contributed to generate it using Join operation. But, the tuple Pt2 of Part1 table is further derived from tuple P3 of Part Table and tuple Pst3 of Partsupp table as shown in Part1 table. This shows, tuple P3 of Part Table and tuple Pst3 of Partsupp table are indirectly contributing towards a tuple (Psnt1) in Partsuppname table. Figure 3.3 shows the complete multi-layer provenance graph of example shown in Figure 3.2. Querying on this multi-layer provenance graph with varying traversal depths i.e. *Multi-Depth provenance querying* is useful in various applications. This leads to the requirement of an efficient multi-layer provenance framework. Furthermore, for the applications which can not afford to lose the data such as auditing, data discovery, error tracing etc., provenance for historical queries are beneficial. TRIO [31, 73] and GProM [121] captures the provenance for updates which further supports to capture provenance for historical queries.

Provenance storage and querying are also critical requirements for all database applications, and must be supported by a provenance framework. Directed Acyclic Graph (DAG) [93, 104] is a common way to represent provenance information, as this information consists of various intermediate sources and/or origins of data in the form of a tree

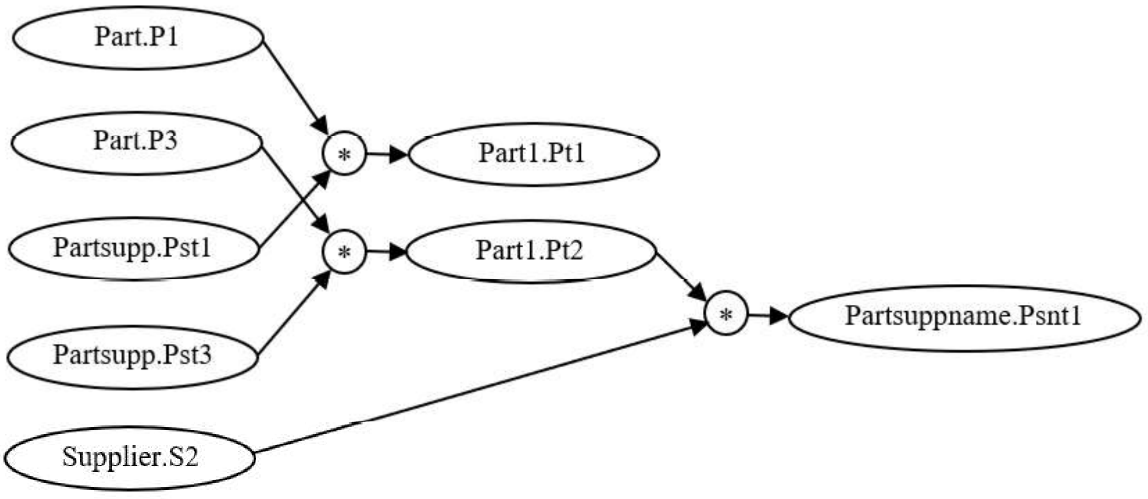


Figure 3.3: Multi-layer Provenance Graph of Example in Figure 3.2

like structure as shown in Figure 3.2 and 3.3. Graphs can be nested, i.e. each data item may be further described by a separate subgraph. Although, storing a graph in a relational database is possible, but querying this is very inefficient as it requires several join operations [93]. Further, as the size of graph increases, execution time for queries increases exponentially [93, 123]. Moreover, SQL queries for any graph operation is very complex and prone to errors. On the other hand, graph databases allow easy expression of queries on graphs which can be executed efficiently as compared to relational databases. Graph databases can also scale to billions of nodes and can traverse thousands of relationships efficiently. This leads to requirement of a provenance framework that can efficiently store and query provenance information especially multi-depth provenance query.

From the available literature on provenance frameworks in relational databases, it is evident that most of the existing data provenance frameworks either do not store the provenance information at all (support querying provenance while generation), and even if they store the provenance information, they store it in relational database for querying and suffers from query performance issues. Few of the existing provenance frameworks such as TRIO and ORCHESTRA supports multi-depth provenance queries but store the captured provenance in relational database only. Although, relational databases can store graph data such as provenance graph, but they are not efficient for multi-depth querying on provenance. Because, with increase in data queries, the volume of provenance information also increases, leading to increase in number of join operations for provenance queries with varying depth. As discussed earlier, provenance for historical queries are also very much needed for applications such as auditing and error tracing, but only TRIO and GProM support provenance for historical queries.

3.1.2 DPHQ

To bridge the above identified gaps, we propose a provenance framework, **D**ata **P**rovenance for **H**istorical **Q**ueries (DPHQ), which supports multi-layer provenance generation and multi-depth provenance querying. Qualitative analysis of existing provenance solutions and our proposed DPHQ framework based on an evaluation matrix which includes type and level of provenance generation, type of queries supported for provenance generation, provenance storage, and provenance visualization support is shown in Table 1.2 of Chapter 1 (page no. 33). DPHQ is capable of capturing the detailed provenance information using proposed extended relational algebra i.e. **P**rovenance **R**elational **A**lgebra (PRA) for every result tuple of ASPJU (Aggregate, Select, Project, Join, Union) queries, including that for historical queries. To capture provenance information for historical queries, we need to efficiently maintain the complete history of all updates in the database. For this purpose, we propose and implement **Z**ero-**I**nformation **L**oss **R**elational **D**atabase (ZILRDB) on top of a relational database. ZILRDB is based on the concept of Zero-Information Loss Database (ZILD) [3]. ZILRDB is space efficient as for every update it only stores updated value in nested table without storing the values of other attributes in new row, unlike relational databases which use Type II changes (new record approach). Further, a result tuple for a query can be generated in multiple ways, thus the proposed DPHQ framework captures the detailed fine-grained provenance information in the form of provenance polynomial, which includes all the derivations to generate a given result tuple. Further, to provide efficient multi-depth provenance query support, captured provenance information about result tuples of queries as well as information about queries are stored in both relational database and in Neo4j graph database. The salient features of DPHQ framework are:

- ***Zero-Information Loss Relational Database (ZILRDB)***: ZILRDB is implemented on top of relational database using object-relational database concepts like user-defined data types and nested tables to maintain all updates efficiently without any loss of information. ZILRDB supports data versioning to maintain history of all data updates (i.e., insert, update, and delete query) as provenance information. It also stores information about all queries, and can generate result of any query that was executed in the past, i.e., historical query. The result will be the same as seen in the past.

- ***Provenance Relational Algebra (PRA)***: Provenance Relational Algebra (PRA) which is an extension of conventional relational algebra is proposed for capturing provenance information for all queries executing on ZILRDB. PRA supports provenance for ASPJU (Aggregate, Select, Project, Join, Union) queries. The proposed DPHQ framework is implemented on top of ZILRDB and captures the detailed provenance information using proposed PRA. Framework also supports Multi-Layer provenance capture.

- ***Querying Current State***: Proposed framework allows querying the current data version and captures the provenance for the same. The captured provenance is stored in both relational and graph database for further analysis.

- ***Querying Historical Data***: It enables queries on archived historical data in nested tables by using proposed extended SQL Constructs viz., "instance", "all", "validon now", and "validon 'date'" respectively. Framework also support for querying data any time in the past as well as some time range as predicate specified in query.

- ***Provenance for Historical Queries***: Proposed framework enables to trace the provenance of each result tuple of query executed in the past i.e. historical query. Framework supports to produce same result of a historical query executed any time later even after the data is modified, which further helps in auditing applications.

- ***Provenance Visualization***: Captured provenance information can be easily visualized in both relational and graph database, i.e., how it is represented, what is provenance for each result tuple of every query executed with varying traversal depths i.e. multi-depth provenance querying for different purposes.

3.2 ZILD Architecture for Relational Database (ZILRDB)

In general, a Relational database $\boldsymbol{D}$ consists of a collection of relations $\{R_1,R_2,.....,R_n\}$ with a pre-defined schema of each relation. Further, each relation schema consists of a list of attributes $\{A_1,A_2,.....,A_n\}$ that contains a set of elements from their attribute domains A_d of particular data type (in-built or user-defined data types). In our work, we first implement Zero-Information Loss Relational Database (ZILRDB) on top of a conventional relational database using object-relational database concepts like user-defined data types and nested tables. Salient features of ZILRDB are as follows:

- ZILRDB maintains all updates efficiently without any loss of information.

- ZILRDB provide data version support to maintain the history of all data updates (i.e., insert, delete, and update operation) in form of provenance data.
- It enables querying on historical data, i.e., retrieving different versions of data.
- It also supports provenance generation for historical queries.

A complete flowchart for creating ZILRDB schema from a given relational schema is shown in Figure 3.4. Process initiates with identifying all the tables in database in which data is updatable and historical data needs to be maintained as provenance information.

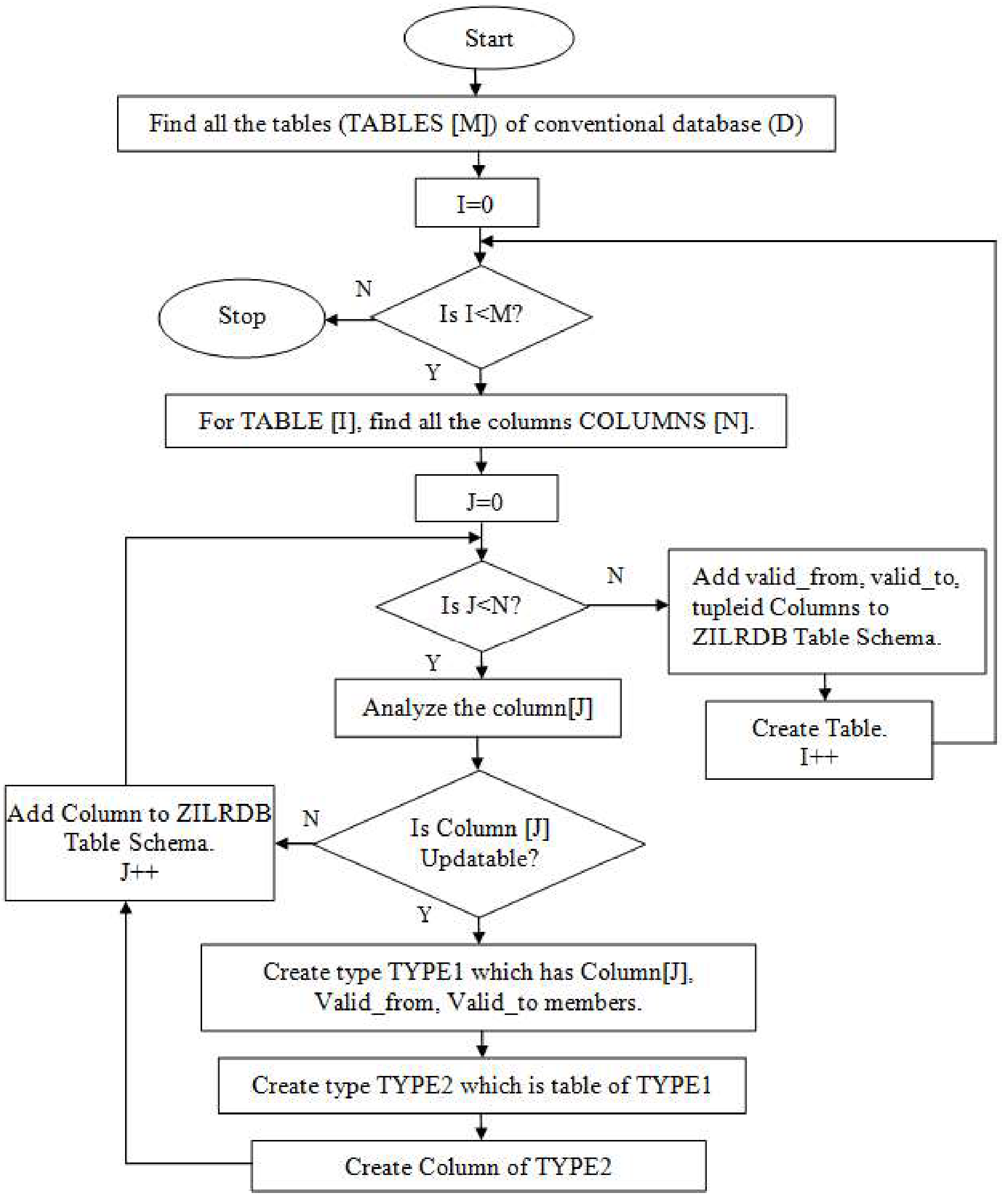

Start
Find all the tables (TABLES [M]) of conventional database (D)
I=0
Is I<M?
N
Stop
Y
For TABLE [I], find all the columns COLUMNS [N].
J=0
Is J<N?
N
Add valid_from, valid_to, tupleid Columns to ZILRDB Table Schema.
Create Table. I++
Y
Analyze the column[J]
Is Column [J] Updatable?
N
Add Column to ZILRDB Table Schema. J++
Y
Create type TYPE1 which has Column[J], Valid_from, Valid_to members.
Create type TYPE2 which is table of TYPE1
Create Column of TYPE2

Figure 3.4: ZILRDB Schema Design Flowchart

Afterwards, identifying all the columns of tables (identified in previous step) which are updatable. If the identified column is not updatable then the column with corresponding data type is added into ZILRDB schema, otherwise for every identified updatable column, new type i.e. "*TYPE1*" is created that consists of "*column name*", "*valid_from*" and "*valid_to*" attributes. Here, "*column name*" stores corresponding "*column value*", "*valid_from*" stores "*date/time*" of existence of column value in database, and "*valid_to*" stores "*date/time*" of expiry of corresponding column value in database respectively. For, the currently existing column value "*valid_to*" stores "*null*". Afterwards, an another type i.e. "*TYPE2*" which is a table of type "*TYPE1*" is created. Thereafter, a corresponding updatable column with user-defined data type "*TYPE2*" is added into ZILRDB Table Schema. Thus, a column of user-defined data type "*TYPE2*" is a nested table in database which can store all updates on corresponding column with its time of existence and time of expiry. Finally, "*Tupleid*", "*vaid_from*", and "*valid_to*" attributes are added in Table Schema. "*Tupleid*" stores unique value in every tuple which can identify the table and tuple with ease, that is further used for generating provenance polynomial. Attributes "*valid_from*" and "*valid_to*" in every tuple capture the "*date/time*" when a tuple is created (i.e. time of existence), and "*date/time*" when a tuple is deleted (i.e. time of expiry). For all the currently valid tuples, their "*valid_to*" attribute stores "*null*" value. Pseudo-code for designing ZILRDB schema is given in Algorithm 1. The algorithm is explained using an example of TPC-H Schema [122] as given below:

PART (partkey, name, mfgr, brand, type, size, container, retailprice, comment)

SUPPLIER (suppkey, name, address, nationkey, phone, acctbal, comment)

PARTSUPP (partkey, suppkey, availqty, supplycost, comment)

NATION (nationkey, name, regionkey, comment)

REGION (regionkey, name, comment)

CUSTOMER (custkey, name, address, nationkey, phone, acctbal, mktsegment, comment)

ORDERS (orderkey, custkey, orderstatus, totalprice, orderdate, orderpriority, clerk, shippriority, comment)

LINEITEM (orderkey, partkey, suppkey, linenumber, quantity, extendedprice, discount, tax, returnflag, linestatus, shipdate, commitdate, receiptdate, shipinstruct, ship-

Algorithm 1 ***ZILRDB_Design***: Design ZILRDB (Zero Information Loss Relational Database) schema

Input: Conventional Relational Database Schema (D)

Output: Corresponding ZILRDB Schema

```
 1: R ← Retrieve all relations in D
 2: for all r ∈ R do
 3:    A ← Retrieve all Attributes of r
 4:    ZILS ← NULL              //Zero-Information Loss Schema (ZILS) corresponds to r
 5:    ZILS ← Add r_tid         //r_tid is an attribute for unique tuple id
 6:    for all a ∈ A do
 7:       if a is Updatable then
 8:          Create $T_N$       //$T_N$ is a Nested Table Type with attributes a, valid_from, valid_to
 9:          Create ZILA        //ZILA is a Zero-Information Loss Attribute of type $T_N$
10:          ZILS ← Add ZILA
11:       else
12:          ZILS ← Add a
13:       end if
14:    end for
15:    ZILS ← Add Valid_from, Valid_to
16:    Generate $Q_c$           //$Q_c$ is a Create Query corresponds to ZILS
17:    Execute $Q_c$
18: end for
19: End
```

mode, comment)

Nested tables are created for all the updatable attributes in the ZILRDB schema to capture all the data update information, as shown in Algorithm 1. Let us consider an example of *Region* table in above TPC-H schema, which consists of *regionkey*, *name*, and *comment* attributes, where *regionkey* is the primary key of table which is not updatable. On the other side, values of attributes *name*, and *comment* are updatable. To store all the update related to attribute *name*, we first create a *nested table* type i.e. *Reg_Name* for *name* as shown below in Step 1 and Step 2:

Step 1:

```
create or replace type R_Name as object
    (eR_Name varchar2(60),
    valid_from timestamp,
    valid_to timestamp);
```

Step 2:

create or replace type Reg_Name as table of R_Name;

In a similar fashion, nested table type i.e. *Reg_Comment* for attribute *comment* is also created. Finally, in Step 3, we are creating Zero-Information Loss relational table corresponds to Region table which consists of nested tables for attributes name, and comment in our database (ZILRDB) as below:

Step 3:

```
create table Zero_Region
    (R_RegionKey number(38) primary key,
    r_Name Reg_Name,
    r_Comment Reg_Comment,
    r_valid_from timestamp,
    r_valid_to timestamp,
    R_tID varchar2(10))
    NESTED TABLE r_Name store as rReg_Name_tab,
    NESTED TABLE r_Comment store as rReg_Comment_tab;
```

In the same manner, all the tables are created in ZILRDB using nested table approach. Here, the complexity of proposed Algorithm 1 depends on the following two factors; First, how many relations 'r' is being designed as zero-information loss in the database. Second, how many attributes 'a' of a relation 'r' are being designed as a nested table for capturing update information. Hence, theoretical complexity of Algorithm 1 is O(r * a) i.e. O(n^2). However, number of relations 'r' in a database are usually very less and a very few attributes of relation are required to maintain updates in database. Therefore, in practical this algorithm runs faster despite O(n^2) complexity.

Now, whenever any insert, update, or delete operation is performed on the database, it maintains complete history of data in ZILRDB, which supports querying current state, querying historical data, querying through time, capturing provenance for historical queries, and obtaining same result of historical query as in its previous executions. Algorithm 2 shows pseudo-code for capturing all necessary changes in ZILRDB whenever any data update query is issued on database. Initially, the issued query is parsed to retrieve parsed results along with the type of query (insert/update/delete) in line 1. If the issued query

Algorithm 2 ***ZILRDB_Gen***: Generate ZILRDB (Zero Information Loss Relational Database)

Input: ZILRDB (D), Query Q (Insert/Update/Delete Query)

Output: Updated ZILRDB (D) with history maintained

```
1: Parsed Result (P), Query Type ($Q_t$) $\leftarrow$ Query_Parser (Q)
2: if $Q_t$ is Insert-Query then
3:     R, PK, $V_{pk}$, A, V $\leftarrow$ Retrieve(P)
       //R=Relation, PK=Primary Key of R, $V_{pk}$=Value of primary Key of row to be inserted,
       //A=Attributes, V=values of attributes A
4:     $Q_i$ $\leftarrow$ Query_Rewriter(R,PK,$V_{pk}$,A,V)
       //$Q_i$ is a rewritten insert query to insert values V of attributes A with "valid_from" of all
       //attributes, and "valid_from" of row to be inserted as "current date/time"
5:     Execute $Q_i$
6: else if $Q_t$ is Delete-Query then
7:     R, PK, $V_{pk}$ $\leftarrow$ Retrieve(P)
       //R=Relation, PK=Primary Key of Relation R, $V_{pk}$=Value of primary Key of row to be deleted
8:     $Q_d$ $\leftarrow$ Query_Generator(R,PK,$V_{pk}$)
       //$Q_d$ is a corresponding update query to set "valid_to" of all attributes and "valid_to" of
       //row to be deleted as "current date/time"
9:     Execute $Q_d$
10: else
11:     R, PK, $V_{pk}$, A, U $\leftarrow$ Retrieve(P)
        //R=relation, PK=Primary Key of R in update query Q, $V_{pk}$=Value of primary Key for row
        //to be updated, A=Attribute to be updated, U=Updated value
12:     $Q_u$ $\leftarrow$ Query_Rewriter(R,PK,$V_{pk}$,A)
        //$Q_u$ is rewritten update query to set "valid_to" of currently valid value of A as "current
        date/time"
13:     Execute $Q_u$
14:     $Q_i$ $\leftarrow$ Query_Generator(R,PK,$V_{pk}$,A,U)
        //$Q_i$ is corresponding insert query to insert U (updated value of A) with "valid_from" and
        //"valid_to" of Attribute A set as "current date/time" and "null" respectively
15:     Execute $Q_i$
16: end if
17: End
```

type is an "*Insert Query*", then a rewritten insert query is generated to insert a new row in the corresponding table with "*valid_from*" and "*valid_to*" attributes of all columns as well as row set as "*current date/time*" and *null* respectively and executed further (refer to lines 2 to 5). If the issued query type is a "*Delete Query*", then a corresponding update query is generated to set "*valid_to*" attributes of all columns as well as row as "*current date/time*" as time of expiry of row and executed further (refer to lines 6 to 9). Thus, in this way, whenever a delete operation occurs, its time of validity (i.e., time of expiry) is set as "*current date/time*", rather than to delete a row permanently from database. Now, if the issued query type is an "*Update Query*", then a rewritten update query is generated first to set "*valid_to*" attribute of currently existing value in corresponding column as "*current date/time*" and executed further (refer to lines 11 to 13). After this, an insert query is generated to insert a new value of column i.e. updated value in the corresponding nested table, and to set values of "*valid_from*" and "*valid_to*" attributes of updated value as "*current date/time*" and *null* respectively and executed further (refer to lines 14 and 15). Thus, in ZILRDB, whenever any update happens, it only stores updated value in nested table of corresponding column without storing a complete row as a new tuple unlike relational databases Type II changes (new record approach). In this algorithm, insert operation takes O(1) time to insert a new row in database, while delete or update operation takes O(n) times to search a row to be deleted or updated in database. Therefore, overall complexity of algorithm 2 is O(n).

3.3 Data Provenance Framework using ZILRDB

In this section, we present the proposed *DPHQ (Data Provenance for Historical Queries)* framework that is designed on top of ZILRDB. DPHQ framework supports to capture provenance for ASPJU (Aggregate, Select, Project, Join, Union) queries along with historical queries, and data update (insert, delete, and update) queries. Figure 3.5 shows the architecture of proposed provenance framework. The architecture mainly consists of following components viz., *Relational Database*, *Query and Provenance Capturing Module (QPCM)*, *ZILRDB*, and *Graph Database*. Initially a user issues a data retrieval query on the conventional relational database in the form of simple SQL query, which is further sent to QPCM layer. The major tasks of provenance generation are performed in QPCM layer. First, QPCM layer rewrites the issued SQL query into a query which is to be executed on ZILRDB. The query on ZILRDB requires join operations between nested tables of attributes and relation mentioned in issued query statement. Second, query is required to be rewritten as per the proposed *Provenance Relational Algebra (PRA)* (explained in next subsection 3.3.1). PRA captures provenance information in the form of provenance polynomial for all the result tuples of a query. The task of query rewriting as per the proposed PRA is also performed in QPCM layer. Therefore, in the whole process, QPCM hides the

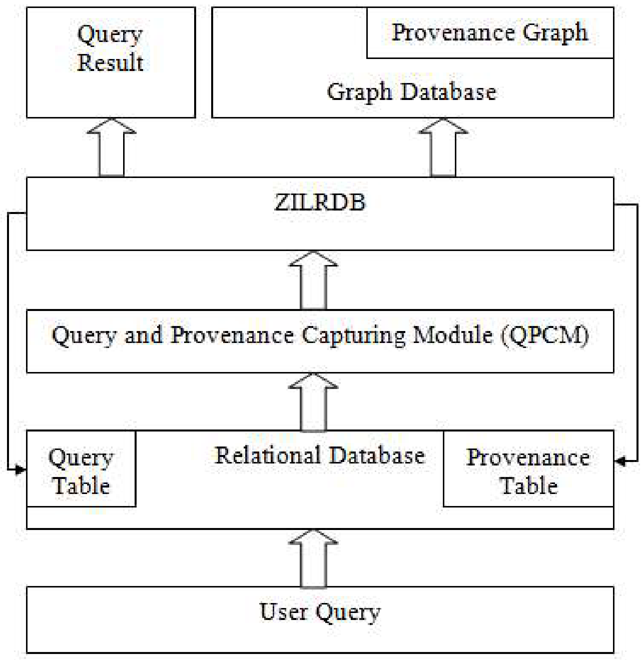


Figure 3.5: Proposed DPHQ Framework

complexity of the system by abstracting ZILRDB layer and provenance generation using PRA from the user. In the end, the query information and the captured provenance polynomials are stored in relational database in *query table (querytabletpch)* and *provenance table (provtbl1)*, respectively, and also in a Neo4j graph database (*provenance graph*) for efficient multi-depth provenance queries as shown in Figure 3.3.

Unlike any conventional relational database, DPHQ supports querying of historical data that has been updated or deleted by using the attributes "*valid_from*" and "*valid_to*" of nested tables as well as rows in ZILRDB. DPHQ also supports historical queries i.e. the queries which are executed in the past and generates the same result in every subsequent executions. This helps in auditing and data diagnostic applications. To support historical data queries and historical queries, we have extended SQL constructs with four new user-defined constructs viz., "*instance*", "*all*", "*validon now*", and "*validon 'date'*" to retrieve currently valid data and historical data. The concept of historical data query is explained with an example query below:

Example Query: Display the partname supplied by supplier 'Supplier#000000001' on 15/04/2015.

User Query: select instance p_name from part p, partsupp ps, supplier s where p.p_ partkey = ps.ps_partkey and s.s_suppkey= ps.ps_suppkey and s.s_name=′ Supplier#00000 001′ validon ′15-Apr-2015′.

In the above example query, user issues the query using extended SQL constructs "*instance*" and "*validon date*" to get historical data of any particular time in the past. The issued query is further passed to QPCM that will automatically rewrite the query that can be executed on ZILRDB.

3.3.1 Provenance Relational Algebra (PRA)

We introduce *Provenance Relational Algebra (PRA)*, which is an extended relational algebra, for capturing provenance information for all queries executing on ZILRDB. PRA is defined in a way that every rule has corresponding natural Relational Algebra (RA) expression, and can be easily translated into SQL queries. PRA comprises of 7 rules, as presented in Table 3.1, which are required for provenance capturing in the proposed DPHQ framework.

Explanation of all rules in PRA with suitable example queries are given next. Instance of relations for example queries Q1 to Q6 are shown in Tables 3.2 to 3.6.

Rule 1 (R1) **:** Conventional relational algebra gives complete relation (R) as a result, but PRA gives complete relation (R) along with unique tuple id's (Tid) to each tuple in result set as provenance information.

Rule 2 (R2) **:** The tuples satisfying the predicate (c) on relation R are retrieved as part of the result set using conventional RA expression, but PRA generates all result tuple's satisfying the predicate (c) on relation R along with the unique tupleid (Tid) for every row as provenance information. Illustration of Rule 2 is shown with Example Query Q1 below:

Example Query (Q1): Display all the parts supplied by manufacturer name "Manufacturer#2".

SQL Query (with construct): select instance * from part where P_mfgr= "Manufacturer#2" validon now;

Here, the above SQL query is first rewritten using rule R2 of PRA, and then executed on ZILRDB to retrieve the result set. Result set includes all result tuples along with tuple

Rule	**Relational Operator**	**Original RA Expression**	**Provenance RA Expression**
R1	Scan/Access	R	$\Pi_{(R,Tid \rightarrow Provenance)}(R')$
R2	Select	$\sigma_{(c)}(R)$	$\sigma_{(c)}(\Pi_{(R,Tid \rightarrow Provenance)}(R'))$
R3	Project	$\Pi_A(R)$	$\Pi_{(A,agg(Tid) \rightarrow Provenance)}(R')$
R4	Join	$R \bowtie_c S$	$\Pi_{R,S,agg(P_i) \rightarrow Provenance}(\gamma_{G,agg(P_i)}(\Pi_{R,S,Tid_{R'} * Tid_{S'} \rightarrow P_i}(R' \bowtie_c S')))$
R5	Left Join	$R ⟕_c S$	$\Pi_{R,S,agg(P_i) \rightarrow Provenance}(\gamma_{G,agg(P_i)}(\Pi_{R,S,Tid_{R'} * Tid_{S'} \rightarrow P_i}(R' ⟕_c S')))$
R6	Right Join	$R ⟖_c S$	$\Pi_{R,S,agg(P_i) \rightarrow Provenance}(\gamma_{G,agg(P_i)}(\Pi_{R,S,Tid_{R'} * Tid_{S'} \rightarrow P_i}(R' ⟖_c S')))$
R7	Aggregate	$\gamma_{G,agg}(R)$	$\Pi_{G,agg,Provenance}(\gamma_{G,agg}(R') \bowtie_{G=X} \Pi_{G \rightarrow X,agg(Tid) \rightarrow Provenance}(\gamma_{G,agg(Tid)}(R')))$

Table 3.1: Provenance Relation Algebra (PRA)

P_tid	P_partkey	P_name	P_mfgr	P_brand	P_type	P_size	P_container	P_retailprice
P52	52	lemon midnight lace sky deep	Manufacturer#3	Brand#35	STANDARD BURNISHED TIN	25	WRAP CASE	952
P53	53	bisque rose cornsilk seashell purple	Manufacturer#2	Brand#23	ECONOMY BURNISHED NICKEL	32	MED BAG	953
P54	54	blanched mint yellow papaya cyan	Manufacturer#2	Brand#21	LARGE BURNISHED COPPER	19	WRAP CASE	954
P55	55	sky cream deep tomato rosy	Manufacturer#2	Brand#23	ECONOMY BRUSHED COPPER	9	MED BAG	955
P56	56	antique beige brown deep dodger	Manufacturer#1	Brand#12	MEDIUM PLATED STEEL	20	WRAP DRUM	956

Table 3.2: Sample Part Table

PS_tid	PS_partkey	PS_suppkey	PS_availqty	PS_supplycost
PS216	54	7555	536	259
PS217	55	56	7874	611
PS218	55	2556	8460	236
PS221	56	57	241#2	855
PS220	55	5056	8278	134
PS219	55	7556	1289	130

Table 3.3: Sample Partsupp Table

R_tid	R_regionkey	R_name
R1	1	Africa
R2	2	America
R3	3	ASIA
R4	4	Europe
R5	5	Middleeast

Table 3.4: Sample Region Table

S_tid	S_suppkey	S_name	S_address	S_acctbal	S_nationkey	S_phone
S6	6	Supplier# 000000006	tQxuVm7s7CnK	1365	14	24-696-997-4969
S56	56	Supplier# 000000056	fUVtlUVal Gi-HBOuYoUQ XQ9NfNLQR3Gl	-632	16	26-471-195-5486
S2556	2556	Supplier# 000002556	285GS9P,eiB9kow jp yjhvEtIx-taR4FplLGuUj0y	5211	21	31-134-422-5382
S5056	5056	Supplier# 000005056	jNR,eLOeczR3Q4 xuq3aW3K	2017	16	26-945-772-6739
S7556	7556	Supplier# 000007556	iI1FclAmBLde PCl6d	-748	23	33-974-496-5278

Table 3.5: Sample Supplier Table

N_tid	N_naionkey	N_name	N_regionkey
N1	1	South Africa	1
N2	2	Canada	2
N3	3	New York	2
N4	4	Toronto	2
N5	5	Jordon	5
N6	6	Nigeria	1
N7	7	Sweden	4
N8	8	Norway	4
N9	9	Sri Lanka	3
N10	10	India	3

Table 3.6: Sample Nation Table

id's of source tuples which contributed to produce the corresponding result tuple as a provenance information under "*Provenance*" attribute as shown in Table 3.7.

Result tupleid	P_ partkey	P_ name	P_ mfgr	P_ brand	P_ type	P_ size	P_ container	P_retail price	Provenance
q1t0	53	bisque rose cornsilk seashell purple	Manufacturer#2	Brand #23	ECONOMY BURNISHED NICKEL	32	MED BAG	953	P53
q1t1	54	blanched mint yellow papaya cyan	Manufacturer#2	Brand #21	LARGE BURNISHED COPPER	19	WRAP CASE	954	P54
q1t2	55	sky cream deep tomato rosy	Manufacturer#2	Brand #23	ECONOMY BRUSHED COPPER	9	MED BAG	955	P55

Table 3.7: Example Query Q1 Result

Rule** 3 **(R3) **:** In Projection operation, a subset of attributes (A) of relation (R) are part of the result set using conventional RA expression. But with PRA, Rule 3 generates Attributes (A) as well as unique tuple id (Tids) of rows which are contributing towards A in the result set of project query. If one result tuple is derived from more than one rows (source tuples), then aggregation on all tupleid's of source tuples are performed which are contributing towards it using '+' operator between tuple id's to generate the corresponding provenance polynomial. It means that there are multiple derivations of a resultant value. Illustration of Rule 3 is shown with Example Query Q2 & Q3 below:

***Example Query (Q2)*:** Display all the supplier names from nation with nationkey 16.

***SQL Query (with construct)*:** select instance s_name from supplier where s_nationkey=16 validon now;

Q2 is a project query to retrieve a selected list of attributes in its result set. Query is first rewritten using Rule R3 of PRA expression, and then executed on ZILRDB to produce the result set. Result set includes all the projected attributes from source table along with tuple id's of source tuples which contributed to produce result tuple as provenance information under "*Provenance*" attribute as shown in Table 3.8.

Resulttupleid	S_name	Provenance
q2t0	Supplier#000000056	S56
q2t1	Supplier#000005056	S5056

Table 3.8: Example Query Q2 Result

***Example Query (Q3)*:** Display different brands of all parts.

***SQL Query (with construct)*:** select instance distinct p_brand from parts validon now;

Q3 is also a project query, but with distinct values in a row. Query will remove the duplicate rows from its result set. Thus, according to Rule R3, if a result tuple is derived from more than one rows in the result set, then the tuple id's of all source rows are aggregated using '+' operator. Result of the above query is shown in Table 3.9 along with provenance information of each result tuple. Here, we can see that result tuple q3t1 has multiple derivations and is generated from two source tuples of Part relation individually i.e. tuples with id P53 and P55. Thus, the provenance polynomial for corresponding result tuple is "*P53+P55*" i.e. aggregation of all source tuples using '+' operator.

Resulttupleid	P_brand	Provenance
q3t0	Brand#35	P52
q3t1	Brand#23	P53+P55
q3t2	Brand#21	P54
q3t3	Brand#12	P56

Table 3.9: Example Query Q3 Result

Rule 4 (R4) **:** If a query is a join query on two or more tables then using Rule R4, result tuples are generated after executing the query on ZILRDB, along with provenance polynomial of each result tuple. Provenance polynomial aggregates all source tuple id's from different tables which are contributing towards each result tuple of query using '*' operator. Example Query Q4 below illustrates the join query using R4. Further, if any result tuple is duplicated in result set then aggregation using '+' operator is performed on provenance polynomials of all duplicated result tuples generated after join operation, as in case of Rule R3. Illustration of removing duplicates in join query using Rule R4 is shown in Example Query Q5.

***Example Query (Q4)*:** Find the supplier names currently supplying part 'sky cream deep tomato rosy'.

***SQL Query (with construct)*:** select instance s_name from part p, supplier s, partsupp ps where p_partkey=ps_partkey and s_suppkey= ps_suppkey and p_name="sky cream deep tomato rosy" validon now;

Result of the above query using PRA Rule R4 is shown below in Table 3.10. Here, we can see that the first result tuple with id q4t0 is generated when source tuples with id's S56, P55 and PS217 of Supplier table, Part table and PartSupp table respectively, are joined together. Here, provenance polynomial represents the join operation with '*' operator.

Resulttupleid	S_name	Provenance
q4t0	Supplier#000000056	S56*P55*PS217
q4t1	Supplier#000002556	S2556*P55*PS218
q4t2	Supplier#000005056	S5056*P55*PS219
q4t3	Supplier#000007556	S7556*P55*PS220

Table 3.10: Example Query Q4 Result

***Example Query (Q5)*:** Display all the region names of all nations.

***SQL Query (with construct)*:** select instance distinct r_name from nation n join region r on r.r_regionkey= n.n_regionkey validon now;

Result of the above query using PRA Rule R4 is shown in Table 3.11. Here, we can see that the first result tuple with id q5t0 is generated when source tuples with id's R2 and N3, R2 and N2, R2 and N4 of region table and nation table respectively are joined

Resulttupleid	R_name	Provenance
q5t0	America	R2*N3+R2*N2+R2*N4
q5t1	ASIA	R3*N9+R3*N10
q5t2	Europe	R4*N7+R4*N8
q5t3	Middleeast	R5*N5
q5t4	Africa	R1*N6+R1*N1

Table 3.11: Example Query Q5 Result

together individually. Thus, region name "*America*" is having three different derivations with provenance polynomials *R2*N3*, *R2*N2*, and *R2*N4*, so after removing duplicates all these individual provenance polynomials of duplicated values are aggregated using '+' operator, and generates complete provenance polynomial as *R2*N3+R2*N2+R2*N4*.

***Rule 5, Rule 6 (R5, R6)* :** For left outer join and right outer join, Rule 5 and Rule 6 are executed in same way as Rule 4 but with modified join operator.

***Rule 7 (R7)* :** If a query is an aggregate query containing any aggregate function such as count, sum, min, max etc., then the provenance polynomial of tuple id's of all source rows which contributing towards one aggregated value in result set is generated along with the result tuple. To generate provenance polynomial, another aggregate function is applied to aggregate source tuple ids. Aggregation on tuple id's are represented using operator $\otimes$. Illustration of provenance for aggregate query is shown in Example Query Q6 below:

***Example Query (Q6)*:** Display the number of suppliers of every part.

***SQL Query (with construct)*:** select instance ps_partkey, count(ps_suppkey) as number_of_suppliers from partsupp group by ps_partkey validon now;

The above query is an aggregate query to count total number of suppliers for each part. The result of above aggregate query is shown below in Table 3.12. Here, result tuple with id q6t1 is generated by aggregating four source tuples of Partsupp table i.e. PS217, PS218, PS219, and PS220. Thus, provenance polynomial for this result tuple is generated as $PS217 \otimes PS218 \otimes PS219 \otimes PS220$.

Resulttupleid	PS_partkey	number_of_suppliers	Provenance
q6t0	54	1	PS216
q6t1	55	4	$PS217 \otimes PS218 \otimes PS219 \otimes PS220$
q6t2	56	1	PS221

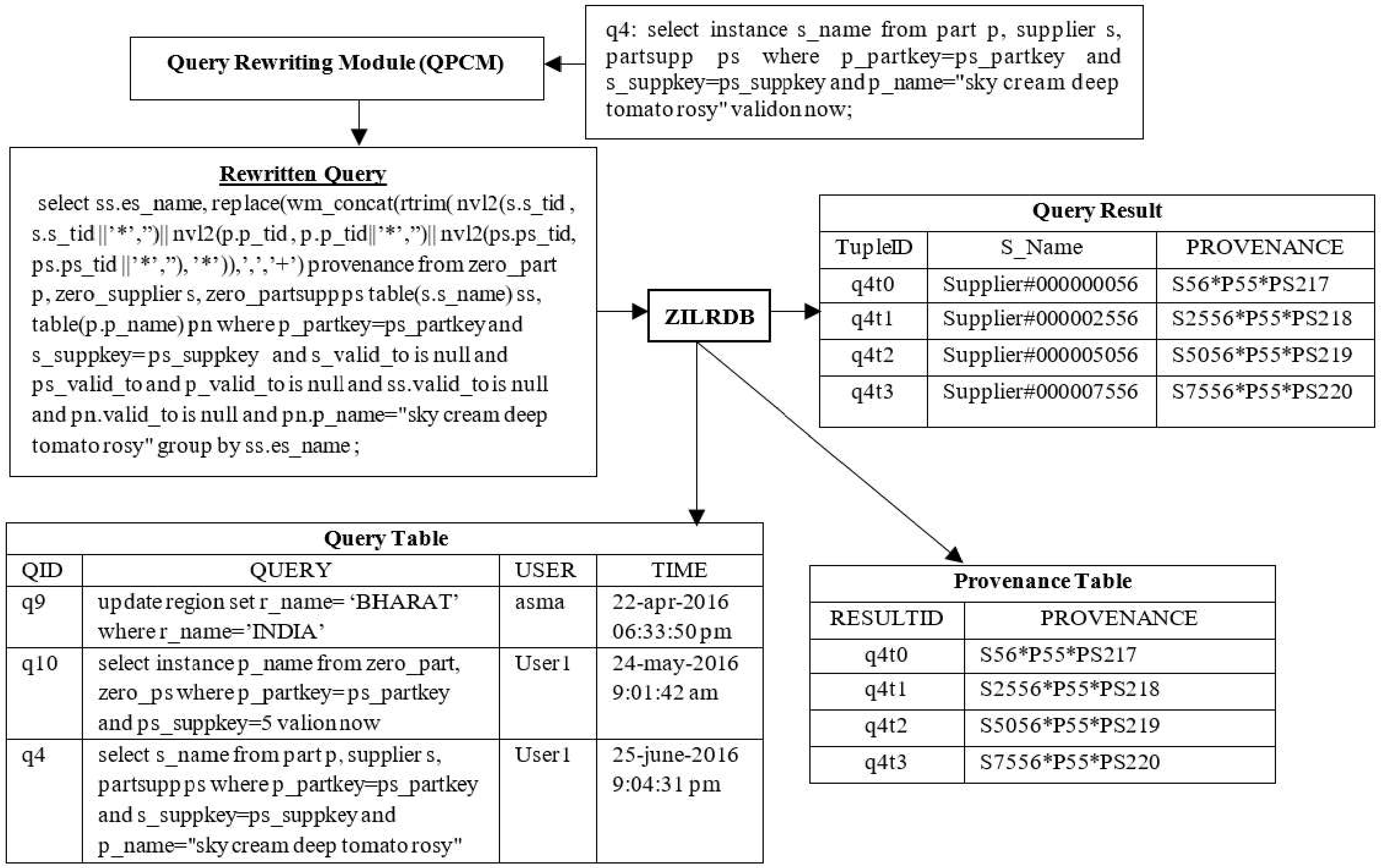
Table 3.12: Example Query Q6 Result

3.3.2 Provenance Generation

Proposed DPHQ framework captures the provenance information using proposed rules in Provenance Relational Algebra (PRA) as explained in section 3.3.1. We will illustrate the provenance generation in DPHQ framework for above Example Query Q4 with a complete flow diagram as shown in Figure 3.6. The user issues a SQL query using user-defined extended SQL constructs viz. "instance", "all", "validon now", and "validon date", which is then automatically rewritten by *Query and Provenance Capturing Module (QPCM)* for executing it on ZILRDB, and capturing the provenance polynomial as per Provenance Relation Algebra. The rewritten query generated from QPCM layer executes on ZILRDB, and query result set along with provenance information as provenance polynomials for each result tuple in the result set are generated. In the whole process, the complexity of rewriting the query as per PRA and ZILRDB schema is completely abstracted from the user by the QPCM layer. ZILRDB returns the query result to the user which consists of result tuples and corresponding provenance polynomial for each result tuple. In Figure 3.6, we can see that the query result displays all the supplier names which are currently supplying the part with name ="sky cream deep tomato rosy". The corresponding provenance in the form of provenance polynomial is also generated as shown in "*Query Result*". Captured provenance is then stored in relational as well as graph database for later analysis. In a similar fashion, provenance for historical queries can also be captured easily using extended SQL operators "instance" and "validon date" as shown in example query below:

***Example Query (Q3)*:** Display different brands of all parts.

Let us consider the above query which was initially executed on 01/05/ 2021. This



q4: select instance s_name from part p, supplier s, partsupp ps where p_partkey=ps_partkey and s_suppkey=ps_suppkey and p_name="sky cream deep tomato rosy" validon now;

Rewritten Query

select ss.es_name, replace(wm_concat(rtrim(nvl2(s.s_tid , s.s_tid ||'*',''))|| nvl2(p.p_tid , p.p_tid||'*',''))|| nvl2(ps.ps_tid, ps.ps_tid ||'*',''), '*')),',','+') provenance from zero_part p, zero_supplier s, zero_partsupp ps table(s.s_name) ss, table(p.p_name) pn where p_partkey=ps_partkey and s_suppkey= ps_suppkey and s_valid_to is null and ps_valid_to and p_valid_to is null and ss.valid_to is null and pn.valid_to is null and pn.p_name="sky cream deep tomato rosy" group by ss.es_name ;

Query Result

TupleID	S_Name	PROVENANCE
q4t0	Supplier#000000056	S56*P55*PS217
q4t1	Supplier#000002556	S2556*P55*PS218
q4t2	Supplier#000005056	S5056*P55*PS219
q4t3	Supplier#000007556	S7556*P55*PS220

Query Table

QID	QUERY	USER	TIME
q9	update region set r_name= 'BHARAT' where r_name='INDIA'	asma	22-apr-2016 06:33:50 pm
q10	select instance p_name from zero_part, zero_ps where p_partkey= ps_partkey and ps_suppkey=5 valion now	User1	24-may-2016 9:01:42 am
q4	select s_name from part p, supplier s, partsupp ps where p_partkey=ps_partkey and s_suppkey=ps_suppkey and p_name="sky cream deep tomato rosy"	User1	25-june-2016 9:04:31 pm

Provenance Table

RESULTID	PROVENANCE
q4t0	S56*P55*PS217
q4t1	S2556*P55*PS218
q4t2	S5056*P55*PS219
q4t3	S7556*P55*PS220

Figure 3.6: DPHQ : Querying and Provenance Generation for Example Query 4

is an example of historical query. Afterwards, some updates are performed on data. But now, we desire to re-run the query for any reason such as auditing. To retrieve the same result as in its previous executions (Example Query Q3 shown in section 3.3.1) as shown in Table 3.9, query is issued using extended SQL operators "instance" and "validon date" as shown below:

***Extended SQL Query*:** select instance distinct p_brand from parts validon '01-May-2021';

The above extended SQL query is initially passed to QPCM layer, and rewritten as per PRA and ZILRDB schema. Rewritten SQL query is passed further to ZILRDB and retrieves the same results as in its previous executions and every subsequent executions also. In this way, DPHQ framework supports to capture provenance for current as well as historical queries.

The framework also supports multi-layer provenance generation as shown in Figures 3.2 and 3.3. For example, consider any create-select query (Q) which first retrieves the result set as per select sub-query along with provenance polynomial of each result tuple, and then creates a table (T) for the result set of select sub-query. Now, when we execute any query Q1 on table T, then the provenance for result tuples of query Q1 will be the tuples from table T because these are the source tuples directly contributing to result of Q1. But, the tuples in Table T are further derived from other sources as per initial Query Q. Provenance polynomial of all the contributed sources was generated and stored in T while executing Q. Therefore, all the sources in provenance polynomial which are contributing to tuples in T via Query Q, are indirectly contributing to result tuples of Query Q1. Suppose, we execute a query Q1 on part table, and result of this query can be stored in a new table, i.e., table1, if required. The corresponding provenance information is stored in provenance table. This is the *Layer 1* provenance. Now, if we further execute a query Q2 on table1, it will generate the result tuples and their corresponding provenance polynomial. Thus, in this way, the immediate source tuples which contributed to produce the result set of query Q2 are from table1 (i.e. traversal depth 1). But, the tuples in table1 are generated from part table via query Q1. So, the tuples of part table are at traversal depth 2 for result of query Q2. Thus, this is *Layer 2* provenance. In the same way, framework supports to capture the provenance up to multiple layers and supports multi-depth provenance querying.

3.3.3 Provenance Storage

The framework stores complete provenance information of each result tuple of a query along with query statement and its related information. This provenance information is used for further analysis, and to perform historical query execution as explained in earlier section. As the provenance is a graph like structure, we are storing the captured provenance information in both relational and graph database for efficient visualization as explained below:

3.3.3.1 Provenance Storage in Relational Database

In relational database, provenance information is stored in two tables viz., "*Query Table*" named "*querytabletpch*" and "*Provenance Table*" named "*provtbl1*" as shown in Figure 3.6. Information about all the queries are captured in a "*Query Table*", which contains QID, Query, User, and Time attributes. A snapshot of "*querytabletpch*" in shown in Table 3.13. This provenance information about queries is beneficial in re-execution of a query, and in auditing such as by whom and when the query is executed.

Provenance information about all result tuples of a query is stored in "*Provenance Table*" as shown in Figure 3.6. This provenance table contains Resultid (concatenation of QID and TupleID), and provenance (provenance polynomial). A snapshot of "*provtbl1*" is shown in Table 3.14. This information allows tracing the origin of any data derived upto any depth, error tracing, data diagnostics etc.

3.3.3.2 Provenance Storage in Graph Database

To provide efficient provenance query support, captured provenance information of all the queries along with each result tuple is also stored as a provenance graph in Neo4j graph database. This allows us to query the provenance in graph database in all possible ways, without toggling between relational and graph databases. Provenance graph of Example Query Q4 is shown in Figure 3.7. In provenance graph, edges are directed from

QID	Query	Username	Validon
q63	select instance p_name from part,partsupp where p_partkey = ps_partkey and ps_suppkey=1	System	02-APR-16 12.00.00.000000000 AM
q78	select instance p_name from part, supplier, partsupp where p_partkey = ps_partkey and ps_suppkey = s_suppkey and s_name = 'Supplier#000000007' and p_retailprice>100	System	04-APR-16 12.00.00.000000000 AM
q83	select instance p_name,s_name from part, supplier, partsupp where p_partkey = ps_partkey and s_suppkey = ps_suppkey and ps_availqty>9900	Tinu	26-APR-16 12.00.00.000000000 AM
q84	select all c_name from customer, lineitem, orders, part where c_custkey = o_custkey and o_orderkey = l_orderkey and l_partkey = p_partkey and p_name like '%chocolate%' and p_mfgr='Manufacturer#2'	Tinu	26-APR-16 12.00.00.000000000 AM
q86	select instance s_name from supplier where s_nationkey=6	System	05-SEP-16 12.00.00.000000000 AM

Table 3.13: Sample Query Table (querytabletpch) in Relational Database

Resultid	Provenance
q78t0	S7*P90006*PS3600216
q78t1	S7*P10006*PS40021
q84t20021	P94484*LI3161453*C65053*O790491
q84t20022	P100514*LI2290360*C65054*O572627
q83t0	S354*P42841*PS171364
q83t1	S5491*P87966*PS351864
q86t0	S70
q86t1	S90

Table 3.14: Sample Provenance Table (provtbl1) in Relational Database

source as well as query node to result tuple. Source tuple nodes are annotated with label "*table name*", and have two properties viz. "*tupleid*" and "*table name*". Result tuple nodes

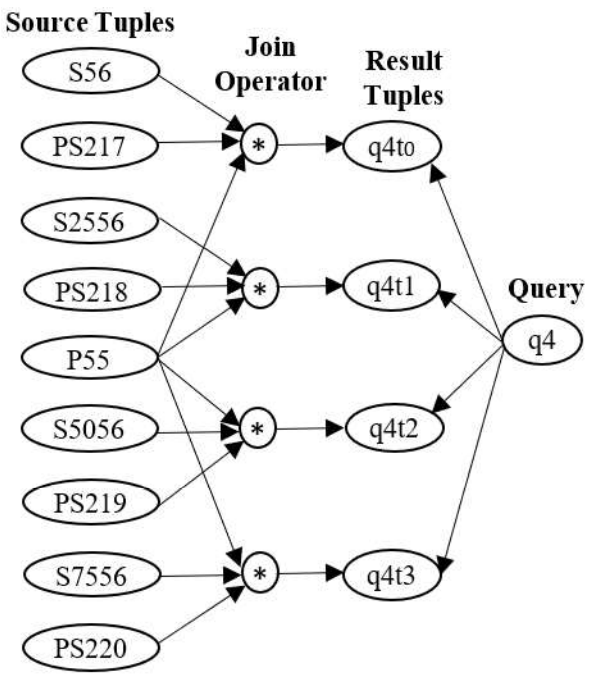


Figure 3.7: Provenance Graph of Example Query 4

in provenance graph are annotated with label "*resultuple*", and have two properties viz. "*resultid*" as in relational database and "*query execution time*" that gives validity time of source tuples. Query nodes in graph are also annotated with label "*querytable*", and have following properties viz. "*QID*", "*query*", "*user*", and "*time*". Edges from source to result tuple via operator node signify the sources which are contributing in the generation of this result tuple and also explains how they are contributing. And edges from query node to result tuple nodes signify that these are the tuples which are generated from this query.

It is observed that graph database incurred high storage overhead as compared to relational database, because it creates a separate node for every source, result, query, and operator to store provenance information. In spite of high storage overhead, graph database is very efficient for queries on provenance data, especially in multi-depth queries as compared to relational database, as it organizes the relevant data using their relationships and quickly respond to complex queries. The relational databases are generally efficient for queries, where data is not highly connected. The performance of provenance queries in relational database degrades with increase in number of join operations in highly interconnected provenance data. A detailed storage and provenance querying analysis in both the databases are given later in section 3.4.2.

3.3.4 Querying Provenance

The DPHQ framework provides support for querying provenance in both relational and graph databases using SQL and Cypher query language, respectively. Sample query table ("*querytabletpch*") and provenance table ("*provtbl1*") in relational database are shown in Table 3.13 and Table 3.14, respectively. Sample provenance graph in Neo4j is shown in Figure 3.7. In provenance graph, all the nodes viz. *Query Node*, *Resultuple Node*, and *Source Node* have labels and properties as explained in section 3.3.3.2. Provenance information can be queried in two perspectives viz. 1. *provenance queries for justifying query results*, and 2. *provenance queries for historical data.* In perspective 1, provenance information is queried for justifying answers of a query result that explains how any information is generated (single-depth or multi-depth provenance query from destination to source). It also includes querying provenance information to explore any result tuples derived from a particular source (single-depth or multi-depth provenance query from source to destination). It may further include the provenance queries like which data has been accessed by any user for any purpose. In perspective 2, provenance information is queried to know about different updates of data, or instance of any data any time in the past i.e. historical data. Illustration of provenance querying from both perspectives in relational as well as graph database are explained with example provenance queries given below:

Example Provenance Query (PQ1) **:** Display all the data accessed by the system user.

Provenance Query in Relational Database using SQL **:**

select distinct qt.qid, pt.resultid , pt.provenance, from provtbl1 pt, querytabletpch qt where qt.username='system' and SUBSTR(pt.resultid, 0, INSTR(pt.resultid, 't')-1) like qt.qid;

In the above query, we want to find all the data which has been accessed by a particular user such as "*system*" in this query, for any specific purpose such as authorization issue, auditing etc. To retrieve the required result, there is a need to perform the join operation between provenance table ("provtbl1") and query table ("querytabletpch") in relational database. The above query requires search in the entire provenance data.

Provenance Query in Neo4j using Cypher **:**

MATCH (node: querytable {user:'system'})-[r]->(b)-[*..2]<-(a) RETURN node.qid,b,a;

The corresponding cypher query in Neo4j for the provenance query PQ1 is shown above. This cypher query locates the starting node with label "querytable" having value of property "user" as "system". It localizes the search space to query nodes only where property "user" has value "system" instead of whole provenance database as in case of relational database. After retrieving the starting node, it traces all the outgoing edges from starting node upto depth 2 to get all result tuple nodes at depth 1, and then all source nodes contributing towards result tuple nodes at depth 2.

This cypher query is efficient as compared to corresponding SQL query, as query is localized to a subgraph of the complete provenance graph, by using any starting point algorithm to find the start node in the graph.

Example Provenance Query (PQ2) **:** Find all the data derived from tuple p1 of part table after 15/may/2016, assuming there is an error occurred.

Provenance Query in Relational Database using SQL **:**

select resultid from provtbl1 pt, querytabletpch qt where SUBSTR(pt.resultid, 0, INSTR(pt. resultid, 't')-1) like qt.qid and provenance like '%P1%' and time=>'15-may-2016';

Provenance Query in Neo4j using Cypher **:**

MATCH (node: part {tupleid:'p1'})-[*]-> (node1:resulttuple {time>=date('15-may-2016')}) RETURN node1;

The above cypher query is efficient as compared to the corresponding SQL query, as it is localized to a subgraph of the complete provenance graph.

Example Provenance Query (PQ3) **:** Display all the tuples generated from tuple PS10 of partsupp table up to traversal depth 2.

Provenance Query in Relational Database using SQL **:**

select resultid from provtbl1 pt where provenance like '%PS10%' union (select resultid from provtbl1 , (select resulted rt from provtbl1 pt where provenance like '%PS10%') depth1 where provenance like '%'+depth1.rt+'%';

Provenance Query in Neo4j using Cypher **:**

MATCH (node: part {tupleid:'p1'})-[*..4]-> (node1:resulttuple) RETURN node1;

Example Provenance Query (PQ4) **:** Lets us consider the provenance graph shown in Figure 3.7, find all source tuples which contribute to derive result tuple with resultid 'q4t1' (To know about quality or trustworthiness of this Result).

Provenance Query in Relational Database using SQL **:**

select provenance from provtbl1 pt where resultid= 'q4t1';

The above query retrieves the provenance polynomial of result tuple with id 'q4t1', which is further parsed to get source tuple id's which directly contributed to generate it. To further retrieve all the source tuple which are indirectly contributing to generate result tuple with id 'q4t1', then the above query executes again on provtbl1 table for each tupleid retrieved in previous step as resultid in this query. Thus, the query becomes quite complex for performing multi-depth provenance query.

Provenance Query in Neo4j using Cypher **:**

match (tupleid: resulttuple {name:'q4t1'}) <-[*]-(b) RETURN tupleid, b;

The above cypher query performs backward tracing in provenance graph upto any depth and retrieves all the source tuples which contributed to produce it either directly or indirectly. This query is much efficient as compared to corresponding provenance query in relational database.

The above provenance queries (from perspective 1) i.e. PQ1 to PQ4, explain about data derivations via backward and forward tracing in provenance data to justify result tuples, and exploring all the data dependent on any particular source tuple. These queries are helpful in auditing, security, knowing trustworthiness of any derived data, and exploring error tracing if any etc. Further, in perspective 2, the proposed DPHQ framework provides support for historical data queries i.e. provenance querying on historical data, by extending SQL query with four user-defined constructs, viz., "*instance*", "*all*", "*validon now*", and "*validon 'date'*", to get instance of data at current time or any particular time in the past, and to retrieve all updates on an attribute till now or any specific time range. Now, we will illustrate the historical data queries with some example provenance queries given below:

Example Provenance Query (PQ5) : Display the current region name for nation key=1.

User Query : select instance r_name from region r join nation n on n.n_regionkey=r.r_ regionkey where n.n_nationkey=1 validon now;

In the above example provenance query, extended SQL construct *instance* and *validon now* are used to get currently valid data.

Example Provenance Query (PQ6) : Display the region name for nation key=1 on date '15-Apr-2016'.

User Query : select instance r_name from region r join nation n on n.n_regionkey=r.r_ regionkey where n.n_nationkey=1 validon '15-Apr-2016';

In the above example provenance query, extended SQL construct *instance* and *validon 'date'* are used to get instance of data at any particular time in the past.

Example Provenance Query (PQ7) : Display all partnames supplied by supplier 'Supplier#000000001' till '15-Apr-2015'.

User Query : select all p_name from part p, partsupp ps, supplier s where p.p_partkey= ps.ps_partkey and s.s_suppkey= ps.ps_suppkey and s.s_name=' Supplier#00000001' valid_ on '15-Apr-2015';

In the above example provenance query, extended SQL construct *all* and *validon 'date'* are used to retrieve all updates on attribute partname supplied by supplier 'Supplier#000000001' within some particular time duration, since its existence.

Example Provenance Query (PQ8) : Display all partnames supplied by supplier 'Supplier#000000001 till now.

User Query : select all p_name from part p, partsupp ps, supplier s where p.p_partkey= ps.ps_partkey and s.s_suppkey= ps.ps_suppkey and s.s_name=' Supplier#00000001' valid_ on now;

In the above example provenance query, extended SQL construct *all* and *validon now* are used to retrieve complete history of all updates on attribute partname supplied by supplier 'Supplier#000000001' till now, since its existence.

3.4 Experimental Setup and Results

3.4.1 Experimental Setup

All the experiments are performed on Windows Machine with Intel i7 processor and 16GB RAM. Oracle 11g as the relational database management system and Neo4j graph database has been used to perform the experiments. Data set of TPC-H benchmark [122] is used and stored in relational database for executing queries, and capturing the provenance for the same. ZILRDB is implemented on top of the relational database. Captured provenance information for relational queries is stored in both relational as well as graph database. Further, to perform queries on provenance, we used SQL for relational database

QID	Query
Q1	Display all the orders placed by customer 'Customer#000078002' before '01-May-2014'.
Q2	Display all the part names whose available quantity is more than 9900. Display the corresponding supplier name also.
Q3	Display the product names which are returned and corresponding customer name also who returned it.
Q4	Display the part names which are shipped on highest priority basis.
Q5	Display the partname for products whose supplycost is more than 1000.
Q6	Display the part names which are shipped on lowest priority basis.
Q7	Update region name of region 'Asia'.
Q8	Update price of Part named " sky cream deep tomato rosy" to 512 by supplier Supplier#000007556.
Q9	Create table part1 as select part_key,part_name,suppkey from part p $\bowtie$ partsupp ps where part_brand= "Brand#1";
Q10	Create table partsuppname as select part_key,part_name,suppkey, supp_name from part1 p1 $\bowtie$ supplier s where p1.part_name= "Tomato Rosy";

Table 3.15: Sample Data Queries for Provenance Capture

and Cypher for Neo4j. We executed a set of twenty five different data retrieval queries in the proposed DPHQ framework to capture the provenance of the query results. As the framework supports Multi-Layer provenance, we captured provenance up to five layers (as explained in previous section) for these experiments. A sample set of data retrieval queries for provenance capture are shown in Table 3.15. We created five different provenance data sets, which store the provenance of 5, 10, 15, 20, and 25 data retrieval queries, respectively. We analyzed the proposed framework on different perspectives viz. 1. Storage overheads of provenance data in relational and graph database, and 2. The average execution time for queries on provenance in relational and graph databases. The queries on provenance data are considered at different traversal depths in all five provenance data sets created during provenance capturing.

3.4.2 Results and Discussions

3.4.2.1 Provenance Storage Analysis

As the number of data retrieval queries increase from 5 to 25, the volume of captured data provenance is also increases from 12 to 284 MB in relational databases, and in graph database it increases from 72MB to 1GB (refer to Figure 3.8). In relational database, provenance for every result tuple is stored as provenance polynomial in a provenance table (provtbl1), and provenance information about the data queries is stored in query table (querytabletpch). Corresponding provenance graph in Neo4j consists of 1529514 nodes and 1913654 relationships among the nodes. It is clear from Figure 3.8, that the storage requirement in relational database is consistently less than the graph database. A graph database (i) creates the node viz., source tuple node, result tuple node, operator node, and query node, (ii) stores all the information about relationship i.e. an edge in explicit relationship store, and (iii) has properties and labels assigned with nodes and relationship which are stored separately. Thus, storage requirement in Neo4j increases with the increase in provenance data. Still, some optimization on storage requirement in graph database is done, by not creating a source node, if it already exists. This optimization further improves the performance of provenance query in graph database.

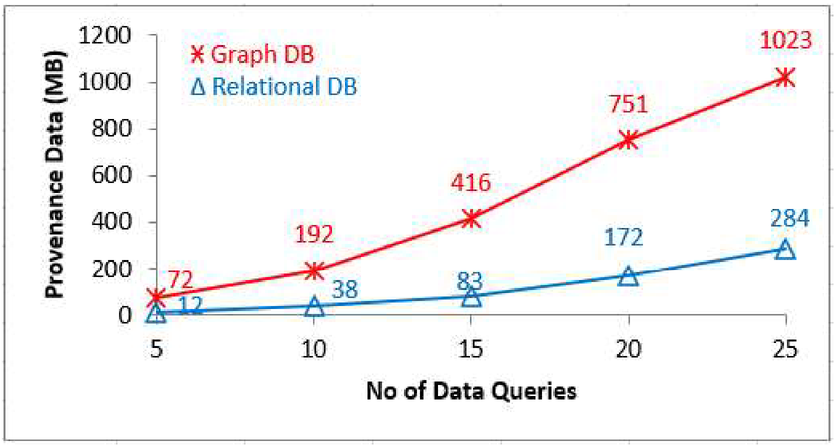


Figure 3.8: Storage Requirement for Provenance Data

QID	Query
PQ1	Display all the source data accessed by system user between dates "01-mar-2016" and "18-apr-2016".
PQ2	Display all the tuples derived from tuple P1 of part table on "20-apr-2016".
PQ3	Display all the queries executed by system user on part table after "15-mar-2016".
PQ4	Display the time when Q76t3 tuple was generated.
PQ5	Display all the sources who contributed to tuple q74t0.
PQ6	Display all the sources who contributed to tuple q74t0 upto depth 2.
PQ7	Display the current price of product named " sky cream deep tomato rosy" supplied by supplier "Supplier#000007556".
PQ8	Display the price of product named " sky cream deep tomato rosy" supplied by supplier "Supplier#000007556" on "01-Apr-2016".
PQ9	Display all the price updates of product named " sky cream deep tomato rosy" supplied by supplier "Supplier#000007556" till now since it exists.
PQ10	Display all price updates of product named " sky cream deep tomato rosy" supplied by supplier "Supplier#000007556" till "01-Apr-2016" since it exists.

Table 3.16: Sample Queries on Provenance

3.4.2.2 Provenance Querying Analysis

In our experiments, first we executed sixty provenance queries with traversal depth one, on all five provenance data sets. Sample queries on provenance data are shown in Table 3.16. Afterwards, the same set of provenance queries are executed with different traversal depths i.e. two, three, four, five, and six. Query on provenance with traversal depth one and two are shown in example provenance queries PQ2 and PQ3, respectively in section 3.3.4. Average execution time of all the queries on provenance for both relational and graph database are recorded. Referring to Figure 3.8, when the volume of captured provenance data is small i.e. up to five data queries, provenance queries on relational database are performing better than graph database as shown in Figure 3.9. But, as the volume of provenance data increases, provenance query requires the search on entire provenance data in case of relational database that gradually deteriorates the query performance. Secondly, the provenance query may require join operation between provenance table and query table for its execution, that is a quite expensive operation in terms of execution time. This join operation may take longer execution time with increase in data size. On the other hand, in graph database, the provenance queries are localized to a subgraph in the whole provenance graph by using any starting point algorithm such as label scan, all node scan etc. Queries also use indexes available on node and properties. After finding the starting node using any starting point algorithm, it only requires to traverse the path starting from it. Further, in the graph database, there is no need of join

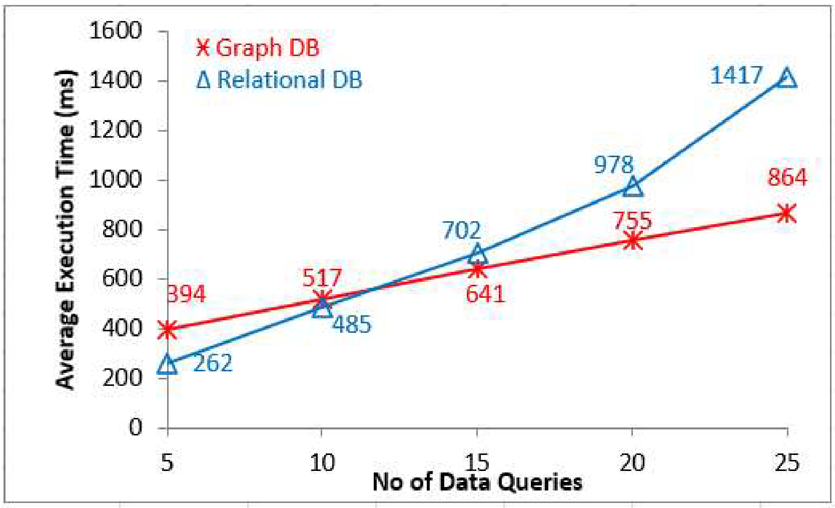


Figure 3.9: Average execution time of queries on provenance : Traversal Depth 1

operations, because predefined relationships in the form of edges already exist. Figure 3.9 shows the average execution time of queries on provenance with traversal depth one in both databases . It is clear from the figure that the relational database performs better when number of data queries are less (up to 10). But, as the number of data queries increase (and consequently the size of provenance data increase), the graph database performs much better. As expected, the performance gains increase with increase in data queries.

Now, as we increase the traversal depth for queries on provenance from one to two (example query shown in Example Provenance Query PQ3 in section 3.3.4), it is found that the average execution time in graph database is almost same as for queries on provenance with traversal depth one. But, in relational database, it increases slightly (refer to Figure 3.10). This is because with increase in traversal depth, number of join operation in relational query increases which degrades its performance.

As we further increase the traversal depth of queries on provenance to three, the gap between graph database and relational database increases as can be seen in Figure 3.11. Although, there is a minor increase in average execution time in graph database as compared to traversal depth two, but there is a significant increase in average execution time of provenance queries in case of relational database.

In the same manner, as we further increase the traversal depth of queries on prove-

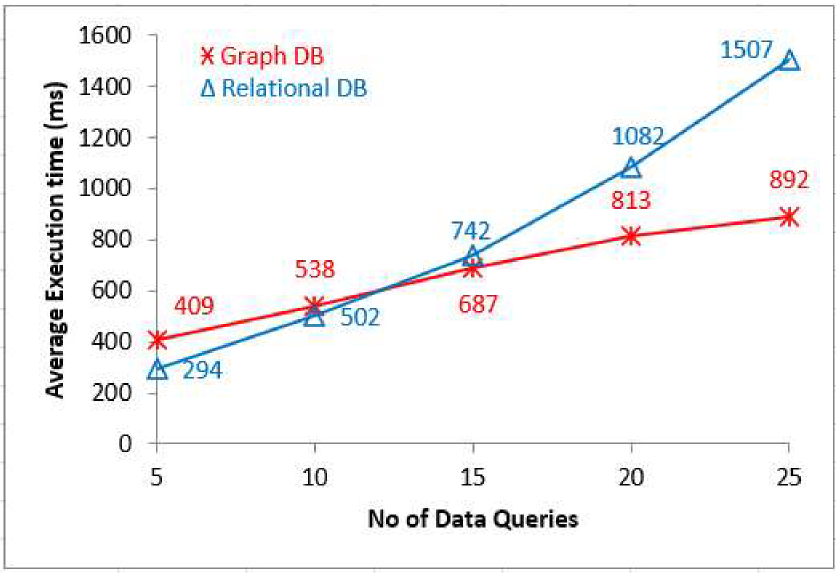


Figure 3.10: Average execution time of queries on provenance : Traversal Depth 2

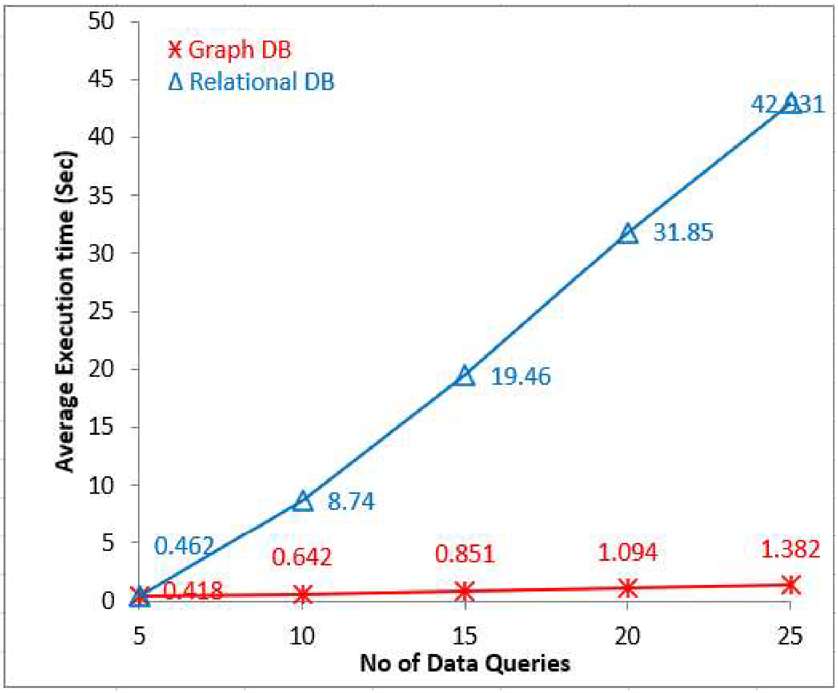


Figure 3.11: Average execution time of queries on provenance : Traversal Depth 3

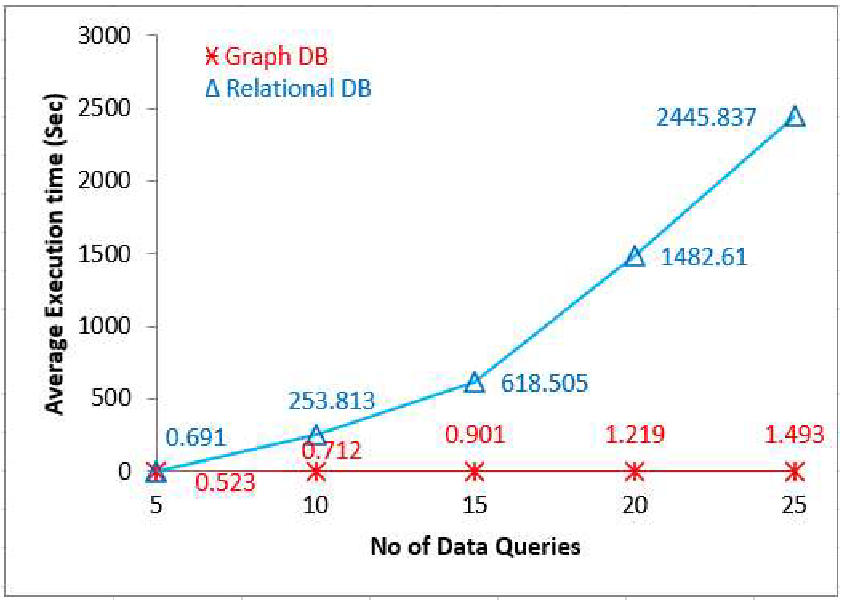


Figure 3.12: Average execution time of queries on provenance : Traversal Depth 4

nance data to 4 and 5, there is a minor increase in average execution time of queries on provenance in graph database, but there is exponential increase in relational database as can be seen in Figure 3.12 and Figure 3.13 respectively.

As the provenance information is captured up to five layers, the average execution time of queries on provenance in graph database at traversal depth six is same as traver-

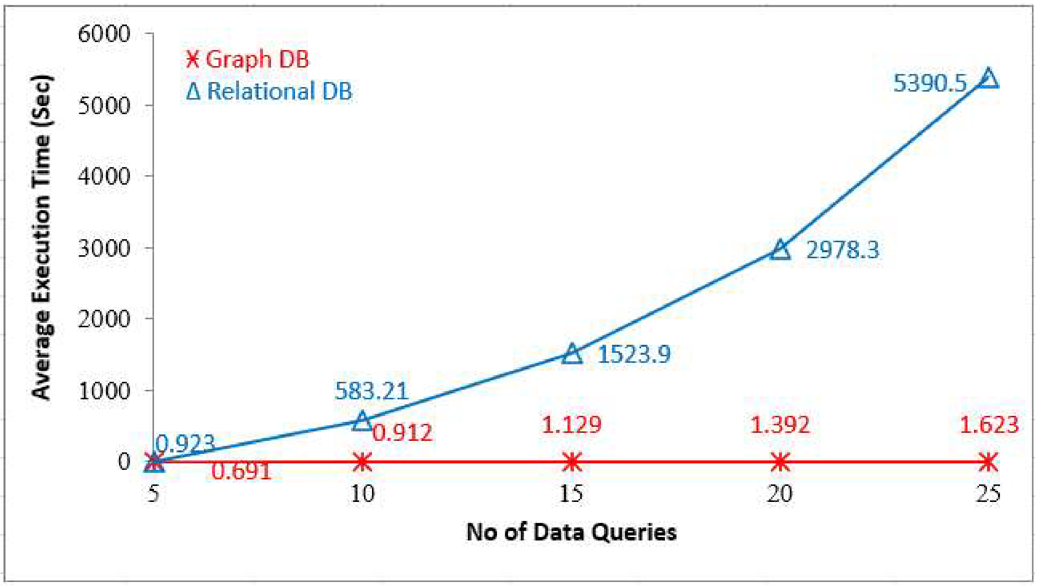


Figure 3.13: Average execution time of queries on provenance : Traversal Depth 5

sal depth five. This is because the graph does not traverse further. But, in case of relational database, a sixth join operation will anyway be carried out, resulting in deteriorated performance. This is evident from the results presented in Figure 3.14.

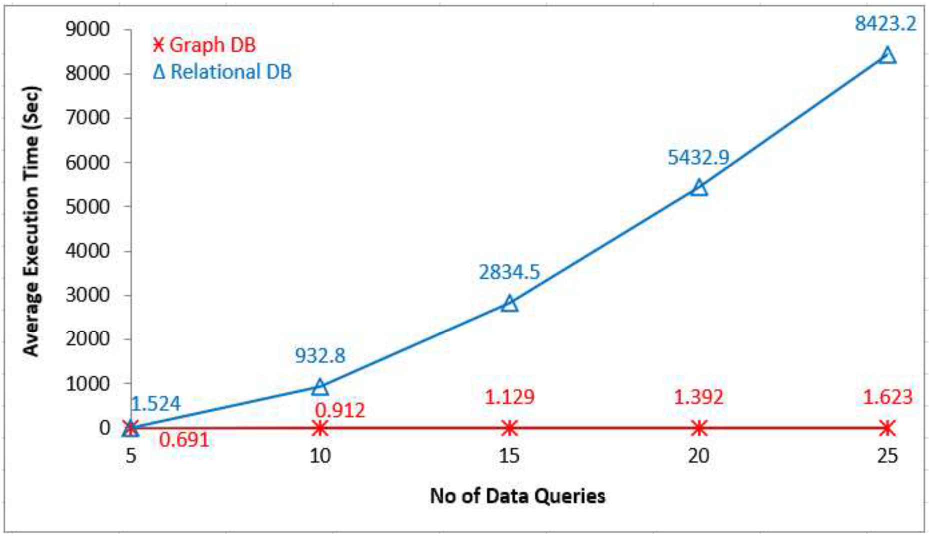


Figure 3.14: Average execution time of queries on provenance : Traversal Depth 6

It can be seen from the results that although the relational database has less storage overheads, but its performance for queries on provenance data is far inferior as compared to graph databases. The performance gains in graph database become more pronounced as the traversal depth of queries on provenance increases. It can also be seen that the average execution time in case of relational databases increases exponentially with increase in traversal depth, and also with increase in provenance data.

3.5 Application Scenarios

Many datasets like genomic data in bioinformatics, scientific experimental data, transactional data etc. grows at an unprecedented scale over time. It is extremely ineffective to process the whole dataset. By visualizing the provenance information, one can easily process the data increments instead of processing the entire dataset. Our proposed framework is capable of generating and visualizing provenance information in different scenarios such as auditing applications, bio-informatics applications, measuring trustworthiness of any data for reliability purpose, data retention applications, data diagnostics etc. Some of the application scenarios are discussed below:

1. **Bio-Informatics:** In bio-informatics applications different databases such as SWIS-PROT, UniProt, Genome, and recent SARS-COV-2 Genome are available. For meaningful analysis, it is required to exploit all the data resources altogether, to help answer research questions which cannot be answered by any single data resource alone. Also, scientists and researchers are generating new datasets from the published data resources. These new databases may contain some results and analysis derived by them. Here, provenance information will be very beneficial to analyse the results of experiments. By applying the proposed framework, provenance information can be captured with ease while generating the new datasets for experiments from all the published data resources. The captured information can later be visualized to know the direct or indirect source of any information with varying depths using provenance graph.

2. **Data Diagnostics:** Using provenance information, data diagnostics can be done in a systematic and efficient manner. One can easily map the result data of a query with input data contributed towards it either directly or indirectly. This mapping identifies how and when any data is generated, who has produced it, etc. Our proposed framework is applicable for such kind of applications.

3. **Data Retention:** In some applications such as auditing or traceability, we desire to

retain the data that is expired/updated. Our framework efficiently maintains the historical data, and is capable to capture and visualize the provenance for historical queries along with current queries. Thus, proposed provenance framework is suitable for auditing applications also.

4. **Information Discovery:** Provenance information is very valuable in information discovery. By visualizing provenance information, one can easily discover the sources of any derived data or discard some of the data originated from some erroneous/-suspicious sources. The proposed provenance framework can be efficiently applied in information discovery applications by visualizing provenance graph from source to destination (forward tracing) or destination to source (backward tracing) with varying depths.

3.6 Conclusions and Future Work

In this work, we designed and implemented DPHQ (Data Provenance for Historical Queries) framework for capturing, storing and querying the provenance data on top of ZILRDB (Zero-Information Loss Relational Database). Provenance relational algebra for ASPJ queries is proposed for capturing how-provenance for queries in the form of provenance polynomial. Framework supports to capture provenance for ASPJU (Aggregate, Select, Project, Join, Union) queries along with the queries executed in the past (historical queries), and data update (insert, delete, and update) queries. The proposed framework supports to capture provenance up to any layers (Multi-Layer provenance). The information about the queries i.e. what was the query, when it was executed, and by whom, are also stored in relational as well as graph database, this helps in executing the past queries. The captured provenance information for query results is stored in both relational and graph databases for further analysis for different purposes like justifying the result tuples via backward tracing, for auditing or identifying any error propagation's via forward tracing. We found that the graph databases offer significant performance gains over relational databases for executing multi-depth queries on provenance. The performance gains in graph database become much more pronounced with increase in traversal depth and also with increase in provenance data. Querying on historical data is also supported in relational database using extended SQL user-defined constructs. In future, we plan to extend it further for capturing the provenance information for complex queries including nested queries and sub-queries.