Chapter 2

Design and Development of Zero-Information Loss Databases

Conventional/Snapshot database systems capture only present state of data without preserving any data updates. But in some application domains such as audit trail, investigations etc., the snapshot databases are ill-suited as these applications cannot afford to lose the past state of data as well as the execution environment (i.e., derivation process/execution flow, query information,database state etc.). In these kind of applications, time values are much needed to be tied up with data values which can indicate the time interval of their existence. The databases those impose time values with data and levying sequential order within database are designated as a ***Temporal Database***. ***Zero-Information Loss Database (ZILD)*** [3] is a special kind of database based on temporal database that maintains historical data along with information about queries (i.e., query statement, user who executed the query, time of execution) in query store and information about updates in update store. ZILD is well suited for designing the provenance framework specially for capturing the provenance for data updates, historical queries, and querying on historical data.

2.1 Zero-Information Loss Database (ZILD)

Database systems largely contain three type of transactions/operations, i.e., insertion (inserting new record), deletion (delete the existing record), and updation (update the existing record), those are performed very frequently. In a ***Conventional Database***, update operation is disastrous in nature, i.e., deleting the old state and preserving only the new state of data after update. Delete operation also deletes the data forever from database while insert operation add new record in the database. Thus, in conventional database, these operations eradicate the environment for historical queries, i.e., previously executed retrieval query may see now some spurious records, or updated data values, or missing data in its result set in case any insert/update/delete operation respectively has been carried out after earlier execution of the query. Accordingly, the conventional database systems are suitable for capturing the present snapshot of a data and its relationship with other data values, but changing or losing the past state whenever database is updated. Thus, conventional databases are also cited as the ***Snapshot Databases***.

In some applications such as tracing origin, audit trails, data diagnostics, accountability, security etc., the conventional/snapshot databases are ill-suited as these applications cannot afford to lose the past state of data as well as the execution environment. Although, a transaction log is maintained in conventional databases that contains a large volume of meaningful information about the transactions/operations occurred including historical data also, but have an ad-hoc structure, and is not readily available for effectively querying. Consider an application of auditing system, where audit trail restore all the operations happened against database to produce the same result. This audit trail requires who has accessed which data, and at what time. It is also required to capture which operations were executed, what was the result of operations at that time, state of database at that time and even after the operations execution. In these kinds of applications, time values are much needed to be tied up with data values which can indicate the time interval of their existence. The databases those impose time values with data and levying sequential order within database are designated as a ***Temporal Database***. These databases generally store data values by associating timestamp values with them which indicate time interval of their validity, and maintain history of data objects based on its existence in real-word or in database. Thus, temporal databases are competent to store various versions of the data based on time, which allow users to audit complete history of data object. Accordingly, we can say that the temporal databases maintain history of database as a series of snapshot copies in a time cube, and a snapshot or conventional

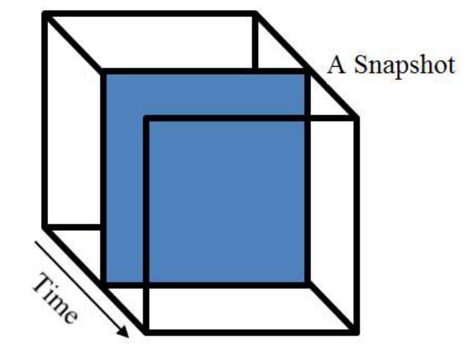


Figure 2.1: Temporal Database as time cube with Snapshot (Conventional) Database as time slice in cube

database is just one time slice of this time cube as shown in Figure 2.1

The time aspect in temporal databases is commonly in the form of "***valid time***", i.e., the time from when any value is true in real-word, or "***transaction time***", i.e., the time when any value was captured in database, or "***decision time***", i.e., the time when any decision was made about the factual information. The databases those associate valid time range with data object are classified as *single-dimension* or *uni-temporal* databases. But in some applications, it is required to audit the historical data value and allowing corrections to that also, which should always be available for historical data queries. This leads to requirement of multi-dimensional databases having atleast two time dimensions (i.e., two-dimensional) associated with data object, i.e., transaction time along with valid time. These two-dimensional databases are classified as *bi-temporal* databases those maintain transaction time at which information was captured in the database as well as valid time for the data when it is valid in the real world. Thus, designing temporal relations by associating transaction time for time stamping with all data values helps in evolution of historical database which maintains history of any data values whenever it is captured in the database with each transaction. This historical database forms the basis of ***Zero-Information Loss Database (ZILD)*** [3].

Zero-Information Loss Database (ZILD) uses time range as [0, Now] to design the previous as well as current (Now) for modeling historical data. Here [0,Now] is universe of time instants which is {0,1,2,3,.....,Now} with linear/sequential ordering ($\leq$) between times. It extends the conventional/snapshot database model that maintains all update operations/transactions in such a way that effect of transaction can be measured at any time in the future and logs can be restored any time without any lose of information, thus named as Zero-Information Loss. To achieve this, whenever any update operation/-transaction occurs, it is divided into two parts viz., 1. updated data value that needs to be captured in temporal database, 2. circumstantial/environment information about the update such as authorized user who perform update, reason and time of update are also required to be captured in corresponding update/shadow store. Along with update store, a query store (single centralized query log) is maintained to store information about all the queries which may helps not only in re-executing any historical query but querying the queries also like what are the different queries executed so far, when and who have performed any query etc.

2.2 Generalized Architecture for Zero-Information Loss Database (ZILD)

ZILD aims to maintain all data update operations (i.e., insert, delete, and update) without any loss of information along with query information. The generalized architecture of ZILD is shown in Figure 2.2 which mainly contains three components viz., "*Temporal Database*", "*Query Store*", and "*Update Store*". *Temporal database* contains time-stamped data, *query store* contains information about all queries executed, and *update store* contains information about all data update operations performed on the database. If the issued query type is an "*Insert Query*" then the new record is inserted in the database and the corresponding execution environment such as time of execution, user who issued the query, reason of new record insertion is captured in update store. If the issued query type is an "*Update Query*" then the corresponding record is updated in database and execution environment such as time of execution, user who issued the query, reason of update is captured in update store. If the issued query type is a "*Delete Query*" then the corresponding record is updated with time of expiry as current time in temporal database and corresponding execution environment such as time of execution, user who issued the query, reason of delete is captured in update store. Finally, if the issued query type is a "*Select Query*" then the results are retrieved from temporal database along with update store for historical data query. Information about all queries such as query statement,

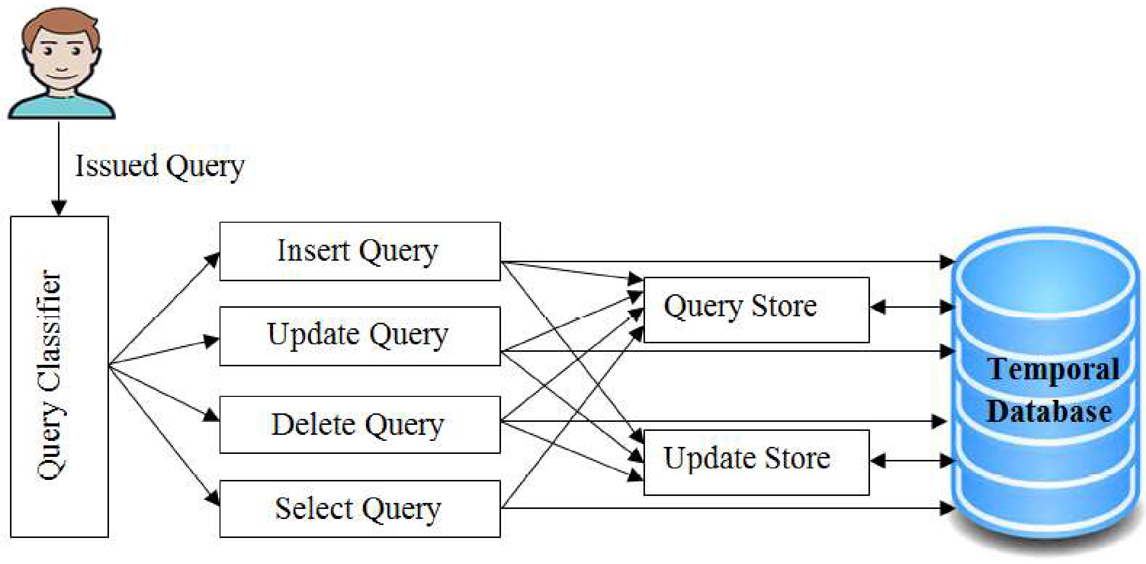


Figure 2.2: Generalized Architecture of ZILD

time, and user who issued the query are captured in query store. ZILD allows the historical queries and querying on historical data. Now, we will briefly explain its operation with an example below:

Example: Consider the time-stamped temporal relation shown in Table 2.1. Here, Twitter_Id: '21256220217241600' and User_Screen_Name: 'Ajweiner3' are recorded in database at transaction time '13-01-2016' and it is valid till now, i.e., at present. Whereas location of user i.e. User_Location with Twitter_ID '21256220217241600' is recorded as 'New York' at transaction time '13-01-2016' and valid till transaction time '21-08-2018'. The location of same user is updated from 'New York' to 'England' at transaction time '22-08-2018' and this is valid till now, i.e., at present. Similarly, other user with Twitter_ID: '721256187350630400' and User_Screen_Name: 'Cesarvaj' and User_Location: 'Canada' is recorded in database at transaction time '25-12-2017'. Twitter_Id and User_ Screen_Name of second user is valid till now i.e. no update but User_Location is changed from 'Canada' to 'Ontario' at transaction time '25-12-2019', thus attribute value 'Canada' is valid from '25-12-2017' to '24-12-2019' and attribute value 'Ontario' is valid from '25-12-2019' till now, i.e., at present. One more row with User_Screen_Name: 'Williom' with Twitter_Id: '721256192572193800' and User_Location: 'Australia' is recorded in database at transaction time '25-12-2017' but row is deleted from database after that. But the row is not permanently deleted from database instead database maintains transaction time, i.e., '29-

Twitter_Id	User_Screen_Name	User_Location
721256220217241600 [13-01-2016 , Now]	Ajweiner3 [13-01-2016 , Now]	New York [13-01-2016, 21-08-2018] England [22-08-2018, Now]
721256187350630400 [25-12-2017 , Now]	Cesarvaj [25-12-2017 , Now]	Canada [25-12-2017, 24-12-2019] Ontario [25-12-2019, Now]
721256192572193800 [25-12-2017 , 29-01-2020]	Williom [25-12-2017 , 29-01-2020]	Australia [25-12-2017 , 29-01-2020]

Table 2.1: A Time Stamped Temporal Relation

01-2020′ when the row has expired. Thus, the temporal range with attributes of this row interprets that the values are valid from '25-12-2017' to '29-01-2020' but now expired. This exhibits the approach how temporal relations can store the insert, update, and delete operations. See the example queries below based on temporal relation shown in Table 2.1.

***Example Query Q1*:** Display the details of all twitter users.

Now the question is what difference is observed in the result of above example query Q1 at Time 'Now' and '01/01/2020'. Here, we can see that the later one, i.e., Query Q1 at Time= '01/01/2020' is an example of *historical query*, i.e., what was the result of above query when it was executed at time '01/01/2020'. Table 2.2 and Table 2.3 shows the result of above query at Time= 'Now' and at Time= '01/02/2020'. Table 2.3 (Historical Query Result) has one extra row as compared to Table 2.2 (Current Query) as the last row with Twitter_Id: '721256192572193800' is valid from [25-12-2017 to 29-01-2020] and at time='29-01-2020' row is deleted from database so currently expired that's why not deriving in result set of current query.

***Example Query Q2*:** What was the location of Twitter User with Twitter_Id: '721256187350 630400' on '01/06/2018'?

The above query is an example of *historical data query* to display the location of user anytime in the past. The query generates result as '*Canada*' as this value is valid from

Twitter_Id	User_Screen_Name	User_Location
721256220217241600	Ajweiner3	England
721256187350630400	Cesarvaj	Ontario

Table 2.2: Example Query 1 Result (Now)

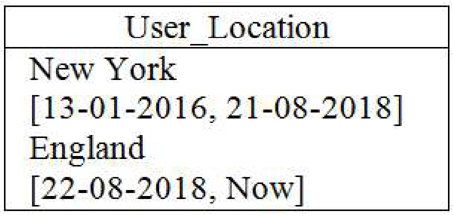
Twitter_Id	User_Screen_Name	User_Location
721256220217241600	Ajweiner3	England
721256187350630400	Cesarvaj	Ontario
721256192572193800	Williom	Australia

Table 2.3: Example Query 1 Result (01/01/2020)

[25-12-2017 to 24-12-2019] and at time='25-12-2019' this value is updated from 'Canada' to 'Ontario'.

***Example Query Q3*:** Display all the location updates of Twitter User with Twitter_Id: '721256220217241600'.

The above query is an example of *historical data query* to display all the updates happened on location attribute of a specific user. Query result includes all location updates along with transaction time and validity time as shown in Figure 2.3.



User_Location
New York [13-01-2016, 21-08-2018] England [22-08-2018, Now]

Figure 2.3: Example Query 3 Result

2.3 Database Schema Design

To develop a provenance framework that is capable of capturing provenance for historical queries, data update queries, and performing historical data queries, we initially design zero-information loss databases on top of Relational Database, Graph Database, and Key-Value Pair Database. Overview of zero-information loss schema design in three databases are given below:

2.3.1 Relational Database

We first develop the Zero-Information Loss Relational Database (ZILRDB) on top of relational database. In general, a Relational database $\boldsymbol{D}$ consists of a collection of relations $\{\boldsymbol{R_1},\boldsymbol{R_2},\ldots\ldots,\boldsymbol{R_n}\}$ with a pre-defined schema of each relation. Further, each relation schema consists of a list of attributes $\{A_1,A_2,\ldots\ldots,A_n\}$ 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.* A complete flowchart for creating ZILRDB schema from a given relational schema is shown in Figure 3.4 of Chapter 3. Process initiates with identifying all the tables in database in which data is updatable, and historical data needs to be maintained as provenance information. Afterwards, identifying all the columns of tables (identified in previous step) those are updatable. If the column is not updatable then add it simply for ZILRDB schema. Otherwise, for every identified updatable column, first create a new type, i.e., *TYPE 1* which consist of *column name*, *valid_from* and *valid_to* members. Afterwards, create another type, i.e., *TYPE2* which is a table of type *TYPE1*, i.e., consisting a table of column name, valid_from and valid_to to stores attribute value, its time of existence, and its time of expiry of attribute value in that column respectively. If the value is currently existing then valid_to stores 'Null' value. Finally, create Column of TYPE2 and add it to ZILRDB schema. In the last, add *tupleid*, *valid_from* and *valid_to* columns to table schema and create table accordingly. Tupleid store unique value in every tuple for generating provenance polynomial for query results later. Valid_from and valid_to in every tuple capture the time when a tuple is created (i.e., time of existence) and expiry time of tuple (i.e., upto which time the tuple is valid in database) respectively. For all the currently existing/valid tuples, their valid_to column stores 'Null' value. Thus, all updates with valid time are maintained in nested tables. Also, information about all the queries/operations executed is stored in "*Query Table*" so that any previous state can be derived anytime. Detailed explanation of ZILRDB design is presented later in Chapter 3 Section 3.2 with suitable examples.

2.3.2 Graph Database

We design a Zero-Information Loss Database on top of graph database, i.e., Zero-Information Loss Graph Database (ZILGDB) especially for capturing provenance for updates, inserts, deletes, historical queries, and to perform historical data queries, along with the provenance for select and aggregate queries in social media environment.

Graph Database is a kind of NoSQL database that uses graph structure comprises of nodes, explicitly defined edges, unique labels on nodes and edges, and properties of nodes and edges for storing data, for semantic queries. Graph databases addresses the shortcoming of relational database for efficient querying specially for linked data. In relational database, relationships between data are implicitly defined that needs to be evolved at the time of query execution,whereas the relationships in the form of edges in graph database are on priority and explicitly stored in database, so faster query execution as compared to relational database. Thus, for social media environment graph databases are well suited for querying relationships. A number of graph databases are available like CODASYL, Neo4j, OrientDB, ArangoDB etc. For our work, we have chosen Neo4j, a key-value pair, labeled graph database.

We are developing Zero-Information Loss Graph Database (ZILGDB) on top of Neo4j database using the concept presented in [3]. ZILGDB maintains all insert, update, and delete operations without any information loss as the provenance information, which aids in capturing provenance information for historical queries as well as querying historical data. To support historical data queries and historical queries, ZILGDB maintains all the updates instead of updating in-place. Whenever any update operation is performed on graph G(V,E), the proposed algorithm first find v, a_v, $G_v(v_a,e_a)$ such that $v \in V$ where v is the vertex to be updated, a_v is the attribute of v to be updated and $G_v(v_a,e_a)$ a subgraph of G such that $v_a \in V$, $e_a \in E$ where v_a is set of vertices associated with v via edges e_a in G. After that, vertex v_c, which is clone of v but with updated value of a_v, is added in G with its "*valid_from*" attribute as "*current date/time*" that will be associated with all v_a via edges e_a respectively, and v is updated with its "*valid_to*" attribute as "*current date/time*". Whenever any insert operation is performed, "*valid_from*" attribute of corresponding new node is set as "*current date/time*". "*valid_to*" attribute of a node is set to "*current date/time*", whenever any delete operation is performed. Information about all the queries executed including time of executions is also stored in "*QUERYNODE*". Detailed explanation for designing ZILGDB is presented later in Chapter 4 Section 4.3 with suitable example queries.

2.3.3 Key-Value Pair (KVP) Database

We also develop Zero-Information Loss Database on top of key-value pair database, i.e., Zero-Information Loss Key-Value Pair Database (ZILKVD) for capturing provenance for updates, inserts, and deletes along with the provenance for select, aggregate, and historical queries, and to perform historical data queries in key-value pair databases.

Key-Value pair database is a kind of non-relational, NoSQL database which uses key-value approach for storing the data. Here, data is represented as a set of key-value pairs where key uniquely identifies every row of database. Key-Value pair databases can be partitioned on a number of nodes in a cluster and can be scaled horizontally which relational database cannot attain. A number of key-value pair databases available are Amazon Dynamo DB, CDB, Gigaspaces, Oracle NoSQL Databases, Keyspaces (Cassandra) etc. For our work, we have chosen Keyspaces, i.e., Cassandra for our implementation. Elementary components of information in Cassandra are Keyspace, Column Family, Row, Super Column and Column. Cassandra is a schema-less database that do not store null values of the columns thus consume much less memory space as compared to relational database. As data are stored into the column in a sorted order of primary/row key of the column, thus it is efficient for search operations including range queries.

Data provenance framework for key-value pair database is proposed in Chapter 5. We first design ZILKVD to maintain all the updates without any information loss as provenance information. The provenance information about update, insert, and delete operations helps in historical data queries as well as provenance for historical queries which can be used for further analysis. Whenever any update operation is issued, it is parsed and corresponding select query is generated to retrieve the old value of column to be updated along with its "*writetime*" and "*rowkey*". Provenance path expression is also generated in the form of "*keyspace/columnfamily/rowkey/updatecolumn*". Finally, the update query is executed and captured provenance information about all the updates including provenance path expression, old column value and its writetime, new value and its type along with current date/time as transaction time is stored in "*update_provenance*" column family. A "*valid_from*" column is added to every row whenever an insert operation is performed. This column stores creation time of row, i.e., "*current date/time*" as a value. Whenever a delete operation is performed, a "*valid_to*" column is then added to the corresponding row with "*current date/time*" as value. It shows that the row has already expired but still exists in the database, which can help in retrieving the same result of historical queries as in its previous executions before delete operation occurs. Information about all the queries executed is also stored in "*query_table*" column family. Detailed explanation of designing ZILKVD is presented later in Chapter 5 Section 5.3. Extended Cassandra Query Language (CQL) constructs are also proposed to support historical data queries.

2.4 Role of Zero-Information Loss Database in Data Provenance

As ZILD, maintains complete information of operational activities in database, it is well-suited for designing the provenance framework specially for capturing provenance for update, insert, delete, and historical queries. This kind of provenance framework is beneficial in number of applications like audit trail, error tracing, fact investigations, tracing source of any historical data etc. Thus, in our proposed provenance framework, we first design zero-information loss database on top of different databases such as relational database, graph database, and key-value pair database as explained briefly in previous section and detailed explanation is given in subsequent chapters, i.e., Chapter 3, 4, & 5 respectively, for capturing provenance for updates, inserts, deletes, and historical queries along with the current queries. Besides recording all the operational and update information, it is also required to query this useful information for auditing, investigation purposes etc. Thus, the proposed provenance framework designed using the concept of zero-information loss database also provides support for querying the stored provenance information for different reasons like tracing the origin of a result, justifying query result, fact investigation etc.

To perform historical data queries and capturing provenance for the historical queries, we basically need to capture following information viz., *query/operation information*, *time of transaction*, *state of the database before and after the operation* i.e., in case of update operation retaining old and new value, in case of delete operation, retaining the value but setting valid_to time as operation time, and in case of insert operation store the value but setting

Operation	Operation Time (OT) & Query (Query_id: Query)
O1	OT = '13-01-2016' Q1: insert (Twitter_Id: 721256220217241600, User_Screen_Name: 'Ajweiner3', User_Location: 'New York')
O2	OT = '25-12-2017' Q2: insert (Twitter_Id: 721256187350630400, User_Screen_Name: 'Cesarvaj', User_Location: 'Canada')
O3	OT = '25-12-2017' Q3: insert (Twitter_Id: 721256192572193800, User_Screen_Name: 'Williom', User_Location: 'Australia')
O4	OT = '01/01/2018' Q4: Display all twitter user details.
O5	OT = '01/01/2018' Q5: What is location of twitter user 'Ajweiner3'?
O6	OT = '22-08-2018' Q6: Update Location of user 'Ajweiner3' to 'England'.
O7	OT = '01/09/2018' Q7: What is location of user 'Ajweiner3'?
O8	OT = '01/09/2018' Q5: What is location of user 'Ajweiner3'?
O9	OT = '25-12-2019' Q8: Update Location of user 'Cesarvaj' to 'Ontario'.
O10	OT = '29-01-2020' Q9: Delete record of user 'Williom'.
O11	OT = '01-01-2021' Q10: Display all twitter user details.
O12	OT = '01-01-2021' Q4: Display all twitter user details.

Table 2.4: Example Operation Sequence

valid_from time as operation time. For example, consider the operation sequence as shown in Table 2.4.

Table 2.1 shows temporal relation corresponding to these operations. Information about all the queries/operations executed is stored in query table as shown in Table 2.5. Accordingly, we are maintaining each activity in database in these two relations (i.e., Table 2.1 and Table 2.5) in case of relational database, and in a different manner in other two databases. This aids in restoring all the operations, i.e., knowing the state before the operations, re-execution of past data retrieval queries (historical queries) and retrieving the same result as their previous executions etc.

Query_id	**Query**	**Time of Execution**
Q1	insert (Twitter_Id: 721256220217241600, User_Screen_Name: 'Ajweiner3', User_Location: 'New York')	13-01-2016
Q2	insert (Twitter_Id: 721256187350630400, User_Screen_Name: 'Cesarvaj', User_Location: 'Canada')	25-12-2017
Q3	insert (Twitter_Id: 721256192572193800, User_Screen_Name: 'Williom', User_Location: 'Australia')	25-12-2017
Q4	Display all twitter user details.	01-01-2018
Q5	What is location of twitter user 'Ajweiner3'?	01-01-2018
Q6	Update Location of user 'Ajweiner3' to 'England'.	22-08-2018
Q7	What is location of user 'Ajweiner3'?	01-09-2018
Q8	Update Location of user 'Cesarvaj' to 'Ontario'.?	25-12-2019
Q9	Delete record of user 'Williom'.	29-01-2020
Q10	Display all twitter user details.	01-01-2021

Table 2.5: Example Query Table

For example, consider operation O4 in Table 2.4 where query Q4 is issued on temporal relation shown in Table 2.1 to display all the twitter users details on 01/01/2018 as shown in Table 2.5. After execution of query Q4, the query result is shown in Table 2.6.

Afterwards, following data update operations viz., Operations O6, O9 and O10 are performed in respective manner as shown in Table 2.4. Then, Operation O11 is performed to execute the query Q10 to display all twitter user details on 01/01/2021 as shown in Table 2.5. The result of issued query Q10 is shown in Table 2.7.

Further, Operation O12 is performed to execute query Q4 on 01/01/2021 as shown

Twitter_Id	User_Screen_Name	User_Location
721256220217241600	Ajweiner3	New York
721256187350630400	Cesarvaj	Canada
721256192572193800	Williom	Australia

Table 2.6: Result of Query Q4

Twitter_Id	User_Screen_Name	User_Location
721256220217241600	Ajweiner3	England
721256187350630400	Cesarvaj	Ontario

Table 2.7: Result of Query Q10

in Table 2.4. However, query Q4 is a historical query that was originally executed on 01/01/2018 as shown in Table 2.5. Therefore, database state is restored again and produce the same result in each subsequent execution as shown in Table 2.6.

In this way, ZILD supports historical queries and also historical data queries which further aid in provenance generation and visualization for various purposes.