# Chapter 5

# Provenance Framework for Key-Value Pair (KVP) Databases

The term "Big Data" is characterised by 7 V's viz., Volume, Velocity, Veracity, Variety, Variability, Visualization, and Value. Veracity of big data that is defined as quality, accuracy and truthfulness of source of data is directly linked with data provenance. Currently, big data and social media are deeply interconnected, as a major portion (over 90%) of the total data in the world is produced through several social media platforms in an unstructured format. Social media is the best example of volume, variety, velocity, and complexity; those are usually concerned with big data [161]. In Big Social Data Analytics, the credibility of an analysis generally depends upon the quality [172] and truthiness of input data which can be assured by the Big Social Data Provenance [153]. In big data applications, provenance information can be used to provide justification for query results.

---

- Asma Rani, Navneet Goyal, and Shashi K. Gadia. 2021. Twitter Data Modelling and Provenance Support for Key-Value Pair Databases. In: Qiao M., Vossen G., Wang S., Li L. (eds) Databases Theory and Applications. ADC 2021. Lecture Notes in Computer Science, vol 12610. Springer, Cham. https://doi.org/10.1007/978-3-030-69377-0_8.

## 5.1 Introduction to Key-Value Pair (KVP) Database

Nowadays, NoSQL databases are frequently used to provide the solutions for various problems related to big data analytics, as these databases efficiently supports to a low latency, horizontal scalability, efficient storage, high availability, high concurrency, and reduced operational costs [45, 59, 140]. Although, there are over 150 different database products that belong to the NoSQL community, yet an increasing attention is being paid to the Key-value pair (KVP) databases. The KVP databases are not only flexible, but efficient also. The key strengths of KVP databases lies in simplicity, scalability, and a very efficiently streamlined architecture. They have the capability to perform extremely fast read and write operations, but querying is very limited. The basic architecture of a KVP database consists of a two column hash table in which each row contains a unique id known as a "key", and a "value" associated with this key. A Key-Value Pair (KVP) database is just like a dictionary in which each key is a word entry and corresponding definition of that word is a value. KVP database is also indexed by the key that directly points to the corresponding value without performing any search, no matter how much data exists in the database.

In KVP databases, an associated matrix is represented by row and column indices to values (i.e., row keys and column label associated with value indexed). Let 'A' is a 'm x n' order matrix, such as:

$$A: \{1,..,m\} \times \{1,..,n\} \rightarrow V$$

Here, set of integers {1,..,m} represents row indices, set of integers {1,..,n} represents column indices and set of values 'V' form a semiring $(V, \oplus, \otimes, 0, 1)$. with addition $\oplus$, multiplication $\otimes$, additive identity 0, and multiplicative identity 1. The dimensions of above matrix are 'm.n'.

Associative Array is an abstract data type that includes a collection of key identifiers and a set of values such as a hash table, dictionary or symbol table. It is just like a map that associates keys to values (i.e. a mapping with a finite domain). An associative array is generalized representation of an array, which maps a key identifier to its associated data value.

The KVP databases are a good choice to handle extremely high volumes of data in a

distributed processing environment as they have a built-in redundancy, which is capable of handling loss of storage nodes. Apache Cassandra [85] is one of the most popular KVP database that comes under the ambit of NoSQL databases. It is a schema-free, horizontally scalable, distributed, column-family store in which each column is a data structure that contains a key, value, and a timestamp, thus it is also named as a key-value pair column-oriented data store (refer to Figure 5.1). It is used in application development by Facebook, Twitter, Cloudkick, and Mahalo etc [105]. The brief introduction of elementary components of information in Apache Cassandra are given below:

**Column**: Column is a smallest unit of information that contains a key, value, and timestamp.

**Super Column**: Super Column or Composite Column is a group of similar columns, or columns likely to query together with common name. Like name which consist of first name, last name, middle name.

**Row**: A Row is a group of orderable columns i.e. columns are stored in sorted order by their column names, with a unique row key/primary key that can uniquely identify data.

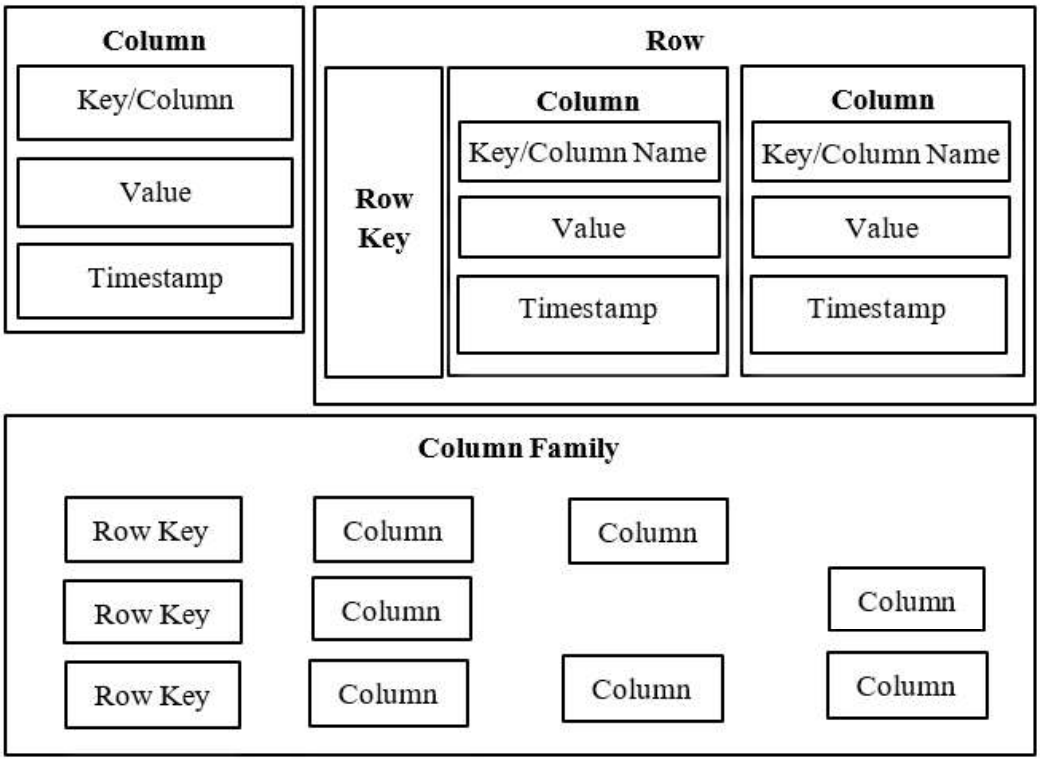


**Figure 5.1:** Cassandra Column, Row & Column-Family Structure

**Column Family**: Column Family is similar to the table in relational database but no pre-defined schema, and also provides flexibility to have different number of columns in different rows. Column families are stored in separate files on the disk.

**Keyspace**: Keyspace is the highest level of information in Apache Cassandra, analogues to the database in relational database, which is the set of related column families. It also maintains the information about data replication, and replication strategy on nodes.

Apache Cassandra is a schema-less database which does not store null values of the columns, thus consume much less memory space as compared to relational database. As data is stored in columns in a sorted order of primary/row key of the column, it is efficient for search operations including range queries. Unlike SQL queries, Cassandra does not support complex constraints and join operations on column families.

## 5.2 Provenance in KVP Databases

Capturing provenance for big data applications is very challenging because of the high volume and unstructured nature of data. A number of challenges are presented for provenance support in big data application by different authors [103, 128, 139, 146] like automatic provenance capture, different granularity levels at which provenance needs to be captured, provenance capturing overhead, and analysing data via querying provenance etc.

A number of provenance models for key-value pair databases exist in the literature those capture the whole system provenance [120, 109, 111], and provenance for workflows [106, 95, 150, 98, 100]. To the best of our knowledge, there is only one data provenance model for a key-value pair system [116], which captures fine-grained provenance information for Cassandra database. Existing provenance model for Cassandra is suitable for a small database, and is application specific. It is suitable for provenance of update queries only. There is no support for provenance of select and aggregate queries. Moreover, it uses the Thrift API, an older version of Cassandra Query, which makes expressing queries quite tedious.

To bridge the above identified gaps, we propose a *Big Social Data Provenance (BSDP)* framework build upon *Zero-Information Loss Key-Value Pair Database (ZILKVD)*. Qualita-

tive analysis of existing provenance solutions and our proposed BSDP framework based on an evaluation matrix which includes data modelling, provenance granularity level, type of queries supported for provenance generation, provenance visualization, and its applicability is shown in Table 1.4 of Chapter 1 (page no. 37). Our proposed provenance framework efficiently captures provenance for all queries including select, aggregate, historical, and data update queries with insert, delete and update operations. ZILKVD is developed based on the concept of Zero-Information Loss Database ZILD [3]. The proposed framework facilitates tracing out the origin and derivation history of a query result. It also supports provenance querying for historical data. The salient features of the BSDP framework are:

- ***Streaming real-time social data***: Fetching a huge volume of real life social data such as Twitter's network through live streaming by using Twitter Streaming API's.

- ***Key-Value Pair (KVP) data model design***: Efficient KVP data model is designed based upon a query driven approach to correlate large size data through relationships and dependencies in appropriate formats so that it makes sense for further analysis.

- ***Zero-Information Loss Key-Value Pair Database (ZILKVD)***: ZILKVD supports data versioning to maintain history of all the updates as a provenance information along with the provenance of insertion and deletion operations.

- ***Provenance for Current Queries***: Proposed provenance framework enables querying the current snapshot of data and captures the provenance for all the results of queries including *select*, *aggregate* and data update queries with *insert*, *delete* and *update* operation. The captured provenance information is stored in respective column families for visualization.

- ***Provenance for Historical Queries***: Proposed framework supports past or historical queries i.e. it generates same results of historical query in every subsequent execution and traces the provenance for the same.

- ***Querying Historical Data***: Proposed framework also allows querying historical data by introducing four new query constructs viz. *"instance"*, *"all"*, *"validon now"*, and

"*validon 'date'*" as extended Cassandra Query Constructs. It supports querying a data element with a given time in the past and with a time range specified in the query statement.

- ***Provenance Visualization***: Stored provenance information can be further analyzed for various purposes such as justifying the result tuple of a query, querying historical data, audit-trail etc.

## 5.3 ZILD Architecture for Key-Value Pair Database (ZILKVD)

ZILKVD is designed using the concept of Zero-Information Loss Database [3], to maintain all the insert, delete, and update operations without losing any information as a provenance data. The architecture of ZILKVD consists of following components viz., *Query Parser*, *Query Rewriter*, *Query Generator*, *Processing Module*, and *KVP Database* (refer to Figure 5.2). When user issues a query, it is sent to the Query Parser to parse the query and to identify the type of that query i.e. Insert (I), Update (U), or Delete (D) query. If issued query type is an "*Insert Query*" (i.e. To insert a new row in database) then the parsed results are sent to the Query Rewriter as mentioned in step $I_1$ and corresponding Rewritten Insert Query ($Q_i$) is generated in step $I_2$. Here, "*valid_from*" column of this new row in corresponding column family is being set to the "*current date/time*" and then it is sent to the KVP database for further execution. If issued query type is a "*Delete*

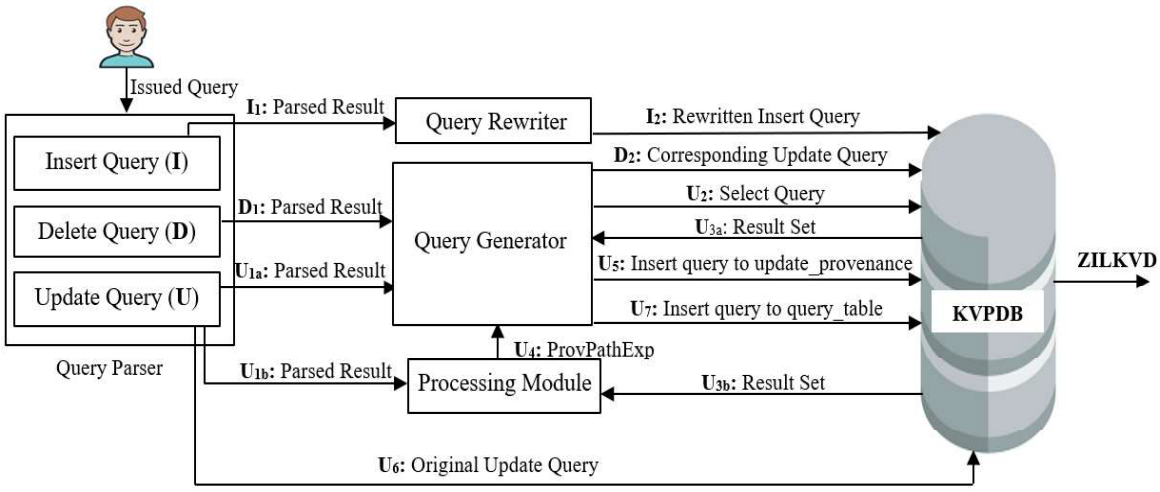


**Figure 5.2:** ZILKVD Architecture

*Query*" (i.e. To delete an existing row from the database) then the parsed results are sent to the Query Generator as mentioned in step $D_1$ and corresponding Update Query ($Q_u$) is generated in step $D_2$. Here, the value of "*valid_to*" column of the row to be deleted from the corresponding column family is being set to the "*current date/time*" and then it is sent to the KVP database for further execution. If issued query type is an "*Update Query*" (i.e. To update an existing row in database) then the parsed results are sent to both Query Generator and Processing Module in steps $U_{1a}$ and $U_{1b}$ respectively. Then, in step $U_2$, corresponding Select Query ($Q_s$) generated from Query Generator is executed on KVP database, to retrieve the following information viz., value of primary key columns of the row to be updated, old value of column before performing update, and its write time in database. This information is sent to the Processing Module in step $U_{3b}$, to generate corresponding Provenance Path Expression (*ProvPathExp*) in the following format i.e. "*Key- space/Column_Family/RowKey/Update_Column_ Name*", and then sent back to the Query Generator in step $U_4$. Now, in step $U_5$, Query Generator generates an insert query ($Q_i$) to insert the following information in "*update_provenance*" column family viz., *Query statement*, *ProvPathExp*, *old_value*, *old_value writetime* (i.e. its valid_from time), *new_value*, *current Date/Time* etc., for further execution on KVP database. After this, both the queries (i.e. generated insert query $Q_i$ and issued update query U) are executed on KVP database in step $U_5$ and $U_6$ respectively, to maintain the complete history of data update operations. Finally, the following information viz., *Query Id*, *Query Statement*, its *time of execution* etc., are also inserted in "*query_table*" column family through an insert query executed on KVP database in step $U_7$.

The implementation code for ZILKVD is given in Algorithm 5 and 6. Two inputs to Algorithm 5 are 1) A KVP Database ($D_{kv}$) and 2) A query Q ( i.e. insert, delete or update query), and output of the algorithm is a ZILKVD database with complete history maintained. According to Algorithm 5, the issued input query Q is first parsed to retrieve the required information, i.e., parsed result $R_p$ and to identify the query type, i.e., $Q_t$ (refer to line 1). If $Q_t$ is an insert query, then a corresponding rewritten insert query $Q_i$ is generated and executed on $D_{kv}$ (refer to lines 3 and 4). If $Q_t$ is a delete query then a corresponding update query $Q_u$ is generated and executed on $D_{kv}$ (refer to line 6 and 7).

If $Q_t$ is an update query then Algorithm 6 i.e. UpdateCassProv is called (refer to line

9). Following two inputs i.e. query Q and its parsed result $R_p$ are passed to the Algorithm 6 and provenance path expression ($P_p$) of updated columns and updated "*query_table*" and "*update_provenance*" column families are obtained as outputs of the Algorithm 6. According to Algorithm 6, all the required information such as KS, CF, PK, $CN_u$, $CV_u$, $CT_u$ etc., are retrieved form $R_p$ (refer to line 1). If Q contains a "*Where Clause*" in its query statement, then value of $V_{pk}$ is retrieved and assigned to RK to uniquely identify a row (refer to lines 2 to 4). Afterwards a corresponding select query $Q_s$ is generated and executed to retrieve old value of column before update, and its write time in database i.e. $CV_o$ and $CV_{owt}$ respectively (refer to lines 5 and 6). Now, provenance path $P_p$ (i.e. KS/CF/RK/$CN_u$) is generated and column family "*update_provenance*" is updated with updated values of $Q_{id}$, Q, $P_p$, $CV_o$, $CV_{owt}$, $CV_u$, $CT_u$, and current date/time (refer to lines 7 and 8). Similarly, if Q does not contain a "*Where Clause*" then again $Q_s$ is generated and executed to store all the query results in RS (refer to line 11). Now, for each result tuple r of RS, value of following parameters i.e. $V_{pk}$, $CV_o$, $CV_{owt}$ etc., are retrieved and value of $V_{pk}$ is assigned to RK. After this, corresponding provenance path $P_p$ (i.e. KS/CF/RK/CNu) is generated and column family "*update_provenance*" is updated with updated values of $Q_{id}$, Q, $P_p$, $CV_o$, $CV_{owt}$, $CV_u$, $CT_u$, and current date/time (refer to lines 12 to 16). Finally, Q is executed and column family "*query_ table*" is also updated with updated values of following parameters i.e. $Q_{id}$, Q, current date/time etc (refer to lines 19 and 20). In respect of complexities of

**Algorithm 5** ***ZILKVD Design***: Design ZILKVD (Zero Information Loss Key-Value Pair Database)

**Input:** Key-Value Pair Database ($D_{kv}$), Query Q (Insert/Update/Delete Query)
**Output:** Zero Information Loss Key-Value Pair Database with history maintained

1: Parsed Result ($R_p$), Query Type ($Q_t$) ← Query Parser (Q)
2: **if** $Q_t$ is **Insert-Query** to insert a new row **then**
3: **Generate** $Q_i$ *//Where $Q_i$ is rewritten insert query with "valid_from" column set as //"current date/time"*
4: **Execute** ($D_{kv}$, $Q_i$)
5: **else if** $Q_t$ is **Delete-Query** to delete an existing row **then**
6: **Generate** $Q_u$ *//Where $Q_u$ is corresponding update query with "valid_to" column set //as "current date/time"*
7: **Execute** ($D_{kv}$, $Q_u$)
8: **else**
9: UpdateCassProv(Q,$R_p$)
10: **end if**
11: **End**

**Algorithm 6** ***UpdateCassProv(Q,$R_p$)***: ZILKVD with Update Management

**Input:** Query Q (Update Query), $R_p$(Parsed Results)
**Output:** ProvPathExp ($P_p$) of updated tuples, updated column families viz., "query_table", "update_provenance"

1: $KS$,$CF$,$PK$,$CN_u$,$CV_u$,$CT_u$ ← **Retrieve**($R_p$)
   *//Where KS=Keyspace, CF=Column_Family, PK=Primary Key of CF,*
   *//$CN_u$=Update column name, $CV_u$=Update column value, $CT_u$= Update column type*
2: **if** Q contains ***where clause*** **then**
3: **Retrieve** $V_{pk}$ ← **Parse** ***where clause*** *//Where $V_{pk}$ = Values of PK*
4: $RK \leftarrow V_{pk}$ *//Where RK is RowKey*
5: **Generate** $Q_s$ *// $Q_s$ is a select query corresponding to Q for retrieving old value //and write time of $CN_u$*
6: $CV_o$ , $CV_{owt}$ ← **Execute** $Q_s$ *//Where $CV_o$ = Old column value, $CV_{owt}$ =Old //column value WriteTime*
7: $P_p \leftarrow KS/CF/RK/CN_u$ *//Where $P_p$ is provenance path*
8: update_provenance ← **Insert** $Q_{id}$, Q, $P_p$, $CV_o$ , $CV_{owt}$ , $CV_u$,$CT_u$ , current date/time
9: **else**
10: **Generate** $Q_s$ *// $Q_s$ is a select query corresponding to Q for retrieving old value, // write time of $CN_u$ and values of PK*
11: RS ← **Execute** $Q_s$ *//RS is result set of query $Q_s$*
12: **for all** $r \in RS$ **do**
13: **Obtain** $V_{pk}$ , $CV_o$ , $CV_{owt}$ ← $r$
14: $RK \leftarrow V_{pk}$
15: $P_p \leftarrow KS/CF/RK/CN_u$
16: update_provenance ← **Insert** $Q_{id}$, Q, $P_p$, $CV_o$ , $CV_{owt}$ , $CV_u$,$CT_u$ , current date/-time
17: **end for**
18: **end if**
19: **Execute** ← Q
20: query_table ← **Insert** $Q_{id}$, Q, current date/time
21: **End**

Algorithms 5 and 6, search operation is efficient in Cassandra as data are stored into the column in a sorted order of primary key of the column. Therefore, insert operation in algorithm takes O(log(n)) times to find a location where to insert new row. Delete operation also takes O(log(n)) to search a row to be deleted and setting valid_to time as current date/time. Update operation takes O(log(n)) time when primary key is given in query statement to search a row and to perform update operation. Otherwise, it takes O(n) time to search all the rows to be updated. Therefore, overall complexity of Algorithms 5 and 6 are O(log(n)) and O(n), respectively. A demonstration of above algorithms with illustrative example query 1 is given below:

***Example Query 1***: Update location of the user with name "DDNewsAndhra".

***Cassandra Query 1***: update user_details set location= 'Andhra' where screen_name= 'DDNewsAndhra';

Initially, the above Example Query 1 is passed as an input query (Q) to the Algorithm 5. Where, the query is parsed to identify its type (i.e. Update Query) and to retrieve the required information. Now, both query (Q) and its parsed results ($R_p$) are passed as inputs to the Algorithm 6. Here, the provenance path expression (i.e. ProvPathExp) of updated tuples along with the updated column families viz., "*query_table*" and "*update_provenance*" of underlying KVP database is obtained as outputs of above algorithm. A snapshot of "*update_provenance*" column family is shown in Figure 5.3.

| Operation | Query Id | Query | Column_ type | Old_ value | Old_value _writetime | New_ value | Provenance_path_ exp | Row_ key | Time |
|---|---|---|---|---|---|---|---|---|---|
| update | q23 | update user_details set location=' Andhra' where screen_name=' DDNewsAndhra ' | VARCHAR | Andhra Pradesh | 2019-12-17 08:30:01 | Andhra | NewTWitter_Keys pace/user_details/' DDNewsAndhra '/Location | screen _name | 2019-12-17 9:16:4 |
| update | q25 | update user_details set user_name= 'Reality Show' where screen_name= 'myshowtalkies' | VARCHAR | MY SHOW MY TALKS | 2019-10-17 09:07:54 | Reality Show | NewTWitter_Keys pace/user_details/' myshowmytalks'/u ser_name | screen _name | 2019-12-17 9:18:3 |

**Figure 5.3:** A Snapshot of "update_provenance" Column Family

## 5.4 Big Social Data Provenance Framework using ZILKVD: Twitter Case Study

### 5.4.1 Twitter Data Model using Key-Value Pair Database

Over the past few years, more than 90% of total data are contributed by various social media platforms. Several leading social media platforms such as Twitter, Facebook, Instagram, WhatsApp etc., are responsible for this unprecedented growth of data. Out of these social media platforms, Twitter is one of the most popular platform which allows users to share their thoughts with worldwide audience. It is tuned for very fast communications over internet with more than 200 million daily active users publishing approximate 500 million tweets daily. A twitter user can either create its own tweet or can retweet the information that has already been tweeted by some other user. A twitter user can choose to follow other users also. For instance, if a user A follows user B, then user A can see B's tweets in his 'timeline'. Twitter's popularity as a massive source of information has

led to research in various domains [132]. Researchers can obtain this information from twitter through publically available Twitter APIs. These APIs are categorized in following two categorise; first is *REST APIs* for conducting specific searches, reading user profile or posting new tweets, and second is *Streaming APIs* to collect a continuous stream of public information. In our framework, we are using Streaming APIs to continuously stream the tweets and related information whenever the new tweet is published as shown in Figure 5.4.

Twitter provides an open standard for authorization known as Open Authentication (OAuth). This authentication mechanism allows controlled and limited access to protected information. Traditional authentication mechanism is vulnerable to theft, while OAuth mechanism provides a more secure approach without using user's username and password. By using a three-way handshaking, it allows users to grant third party access to their data. As user's password for his/her twitter account is never shared with this third-party application, therefore, user's confidence in the application is also improved. Twitter APIs can only be accessed by a twitter application using OAuth authorization mechanism.

To get the authorization for accessing the protected data, user first creates a twitter

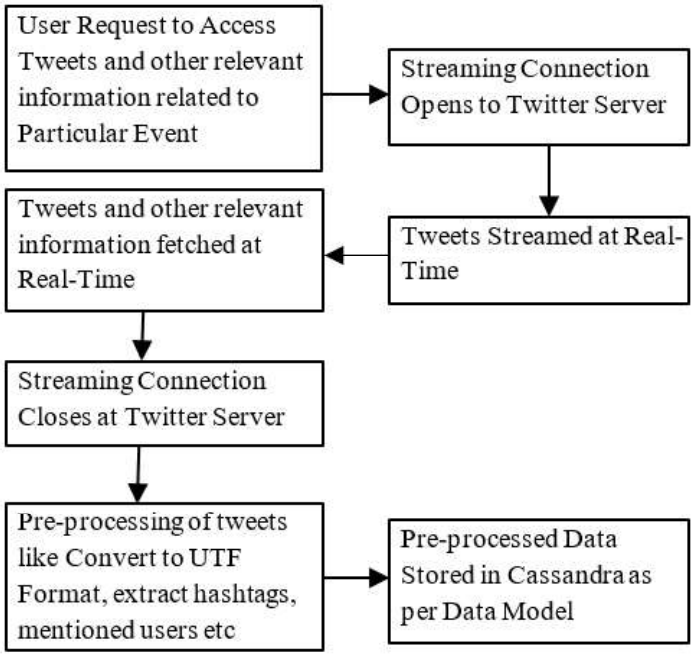


**Figure 5.4:** Twitter Data Streaming

application which is also known as consumer. After registering this application on twitter, a *consumer key* and a *consumer secret key* is issued to the application by twitter that will uniquely identify this application. By using this *consumer key* and *consumer secret key*, application creates a unique twitter link through which user authenticate him/her self to twitter. After verifying the user's identity, twitter issues an *OAuth verifier* to the user. Application uses this OAuth verifier to request an *Access Token* and *Access Token Secret* that is unique to the user. Now, twitter application authenticates the user on twitter by using these Access Token and Access Token Secret, and make API calls on behalf of the user (refer to Figure 5.5). By using these Access Credentials, we fetched all the tweets related to a specific event through live streaming to design an efficient key-value pair data model as explained in Algorithm 7 (i.e. TweetCassandra). The two inputs to the algorithm are 1) Twitter API Access Credentials i.e. Consumer Key ($C_k$), Consumer Secret Key ($C_{sk}$), Access Token ($A_t$), Access Token Secret ($A_{ts}$); and 2) Event name (E) for which related tweets are required to be fetched. While an efficient query driven KVP data model in Apache Cassandra is obtained as an output of this algorithm.

After successful authorization on Twitter's Network using access credentials, tweet set (T) related to input event (E) is fetched through live streaming of social data (refer to line 1). Then, on every fetched tweet (t) of tweet set, pre-processing is performed to extract the following information viz. User (U) who posted the tweet, Hashtags (H) and Mentioned

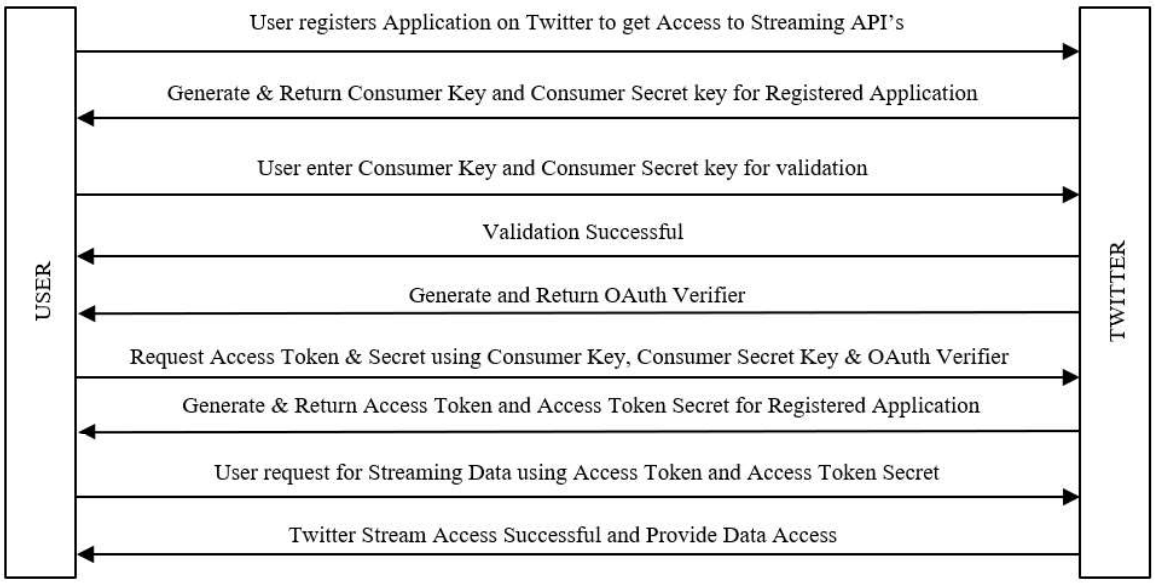


**Figure 5.5:** Open Authentication Process for Twitter

**Algorithm 7** ***TweetCassandra***: Cassandra Data Model creation for Twitter Streaming Data

**Input:** Twitter API Access Credentials ($A_t$, $A_{ts}$, $C_k$, $C_{ks}$) and Twitter Event (E)
*//Where $A_t$=Access Token, $A_{ts}$= Access Token Secret, $C_k$=Consumer Key, $C_{sk}$= Consumer Secret Key*
**Output:** Cassandra Data Model (CDM)

1: **Fetch** Tweet Set T of Event E
2: **for all** t ∈ T **do**
3: U, H, M, Ur, T, $TW_i$ ← **Preprocess**(t)
4: $F_{rl}$, $F_{ol}$, $U_i$← **Fetch** U *//Where $F_{rl}$=User's friend list, $F_{ol}$=User's follower list, $U_i$=other user information*
5: **for all** f ∈ $F_{rl}$ **do**
6: $F_{rld}$← **Fetch** f *//Where $F_{rld}$=User's friend details*
7: **end for**
8: **for all** f ∈ $F_{ol}$ **do**
9: $F_{old}$← **Fetch** f *//Where $F_{old}$=User's follower details*
10: **end for**
11: $C_{cf}$ ← **Insert** (Extracted Data)
12: **Update** $C_{cf}$ *//$C_{cf}$ = Cassandra column-families*
13: **end for**
14: **End**

Users (M) in the Tweet, Tweet Body (T) in UTF-encoding, and other related information etc (refer to lines 2 and 3). Simultaneously, for each User (U) following related information viz., list of user's friends ($F_{rl}$), list of user's followers ($F_{ol}$) and user's profile attributes ($U_i$) such as user_name, screen_name, profile created date, twitter id, location etc., are also extracted from User's Twitter profile (refer to line 4). Similarly, Friend Details ($F_{rld}$) and Follower Details ($F_{old}$) of each user are also extracted (refer to lines 5 to 10). Finally, all the extracted information is stored in corresponding column families of Apache Cassandra in appropriate format. This information is continuously streamed and populated in different column families to build an effective query driven KVP data model for efficient queries.

Although, Apache Cassandra is known for flexible data management to manage world's biggest datasets on clusters of several nodes deployed at different data centres. However, one of the major challenges that big social data applications face when choosing Apache Cassandra is data model design that is significantly different from traditional data model design methodologies. Traditional data model design methodology like the one used in relational databases, is purely a data driven approach. On the contrary, data model design for Cassandra begins with application specific queries and is purely a query driven approach. Several SQL constructs such as data aggregation, table joins etc., are not sup-

ported by Cassandra Query Language (CQL). Therefore, data modelling in Cassandra relies on denormalization of database schema which enable a complex query to execute on a single column family only, to retrieve the required information. In this way, data duplication is common in Cassandra column families to support a variety of queries. Database schema design for big social data in Cassandra requires not only the understanding of relationships and dependencies among social data, but also the understanding of needs to access this data through a query driven approach.

In this work, we applied a query driven methodology in KVP data model design. By a query driven, we mean designing a data model on the basis of what type of queries our database will be required to support. This approach provides not only the sequence of tasks, but also aids in determining what type of data will be needed and when? In our proposed framework, we designed a query driven data model based on frequent queries required to execute on Twitter dataset as shown in Figure 5.6. Initially, all the tweets posted by different Twitter users in the response of a particular event are fetched through Twitter's Streaming APIs. However, all such information is not useful for our data model, therefore, only required information viz., tweet id, tweet text, tweet published date, hashtags, user_name, screen_name, profile created date, twitter id, location, friend list, follower list etc., are extracted from the input list of tweet objects. Simultaneously, pre-processing on extracted data is performed to convert them in a required format. Afterwards, all such pre-processed data are stored in different column families of Apache Cassandra. The structure of such extracted information is given below:

***tweet_id:*** *The tweet identification number assigned by twitter.*

***tweet_body:*** *The text content of the tweet.*

***published_date:*** *Date on which tweet was published on twitter.*

***screen_name:*** *User's screen name on twitter.*

***user_name:*** *User's profile name on twitter.*

***twitter_id:*** *Identification number of user/author of the tweet.*

***created_date:*** *Creation date for the user account.*

***location:*** *The geolocation of the user/author of the tweet.*

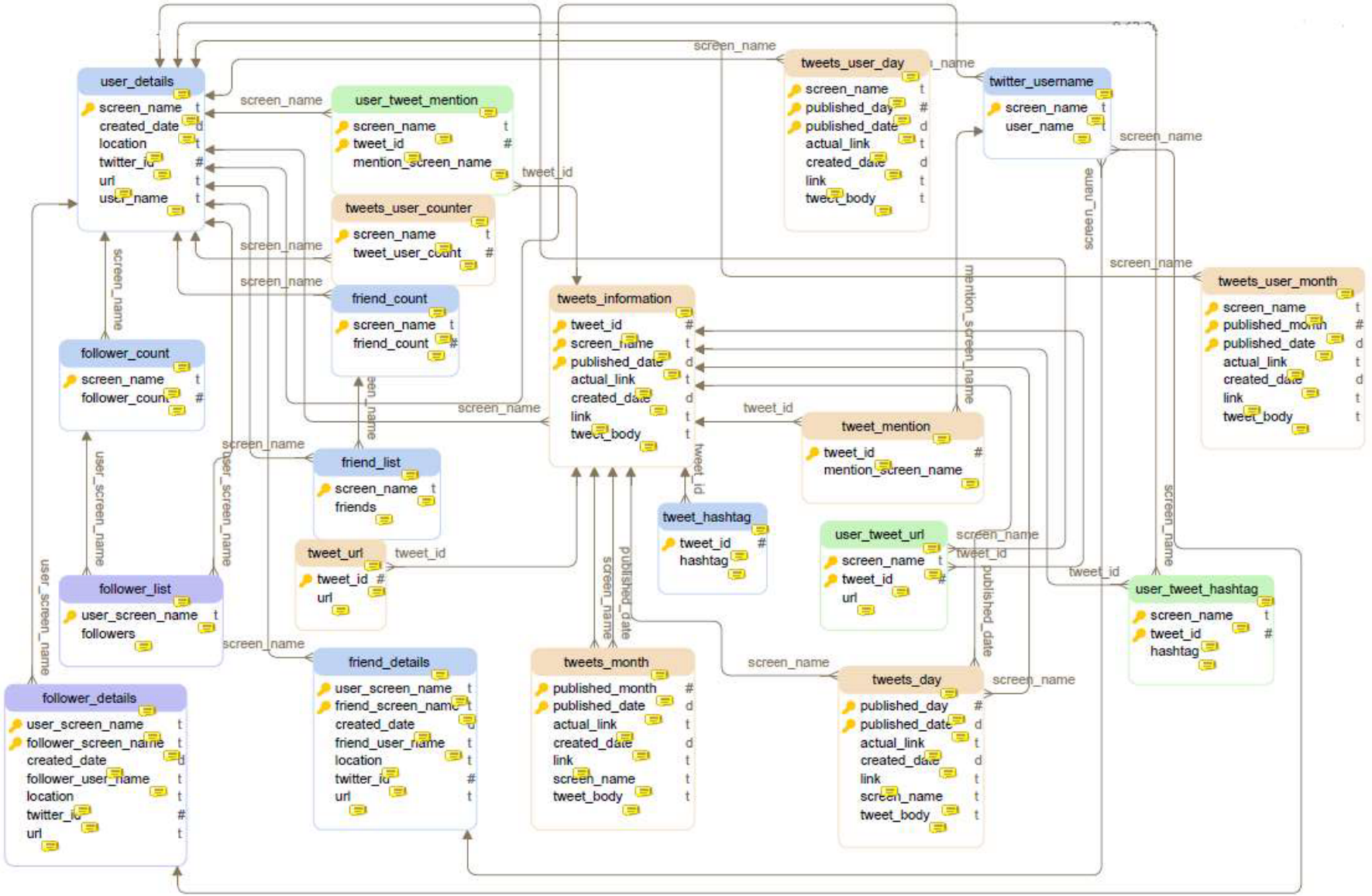


**Figure 5.6:** Cassandra Data Model

***url:*** *The list of urls contained in the tweet.*

***hashtag:*** *The list of hashtags contained in the tweet.*

***mention:*** *The list of mentioned users contained in the tweet.*

***friends:*** *The list of friends that user is following.*

***friends_details:*** *Details of all friends of user i.e. Screen_name, User_name, location etc.*

***friend_count:*** *The number of friends that user is following.*

***followers:*** *The list of followers of the user.*

***followers_details:*** *Details of all followers of user i.e. Screen_name, User_name, location etc.*

***follower_count:*** *The number of followers of the user.*

***tweet_user_count:*** *The total number of tweets that the user has published over time.*

Proposed data model contains a keyspace named "NewTwitter_Keyspace" that consists of 20 Column Families. The various column names of these column families with their row keys are also mentioned in Figure 5.6. All the 20 column families are organized on the basis of social data set fetched from the Twitter's network to support different query sets for capturing, storing and querying provenance. Cassandra Query Language (CQL) is used for querying and to communicate with Apache Cassandra. For example, to retrieve the total number of tweets posted by a given user on a particular day, we can issue the following CQL statement to the proposed data model i.e.

*"select count(*) from tweets_user_day where screen_name='sunilthalia' and published_day=8 and published_date>= '2019-10-08' and published_date< '2019-10-09' group by screen_name;"*

### 5.4.2 Provenance Generation

We designed and implemented three provenance generation algorithms for *select*, *aggregate*, and *historical queries*, respectively. The high level details of all the algorithms along with their illustrative example queries are given in the following subsections:

#### 5.4.2.1 Provenance Generation for Select Queries

Proposed framework supports to capture provenance information for select queries. The high-level details of provenance generation algorithm for select queries i.e. "*Select-Prov*" are given in Algorithm 8. In proposed algorithm, a select query ($Q_s$) and its query id ($Q_{id}$) is passed as inputs and a comma separated list of provenance path expression (P) for each value exists in the result tuple of a query result along with the following updated column families viz., "*select_provenance*", and "*query_table*" are obtained as outputs of the algorithm. Initially, $Q_s$ is parsed and following information viz., KS, CF, PK, CN etc., are retrieved from the query statement in the form of parsed result $R_p$ (refer to lines 1 and 2). Then, a rewritten select query $Q_r$ is generated by appending a predicate (i.e. "valid_to") in the query statement (refer to line 3). The value of this predicate is being set to the *Null* to retrieve currently existing rows. After this, $Q_r$ is executed and all its result tuples are stored in record set (RS) (refer to line 4). Now, for each result tuple r of

**Algorithm 8** ***SelectProv***: Provenance Generation for Current Select Query

**Input:** Query $Q_s$ (Select Query), $Q_{id}$ (Query Id)
**Output:** ProvPathExp ($p_i$) with each value in result set, updated "select_provenance" and "query_table" column families

1: $R_p \leftarrow$ **Parse** $Q_s$
2: *KS,CF,PK,CN* $\leftarrow$ **Retrieve**($R_p$) *//Where KS=Keyspace, CF=Column_Family, //PK=Primary Key of CF, CN=Column names in $Q_s$*
3: **Generate** $Q_r$ *//$Q_r$ is rewritten select query for retrieving values of //Pk and appending predicate with "valid_to" as Null*
4: RS $\leftarrow$ **Execute** $Q_r$
5: k=0
6: **for all** $r \in RS$ **do**
7: $r_{tid} = Q_{id}$+ 't'+ k *//$r_{tid}$ is unique id of each result tuple of $Q_r$*
8: **Set** P = Null
9: **Obtain** $v_{pk}$ *//$v_{pk}$=Values of PK*
10: RK $\leftarrow$ $v_{pk}$ *//RK is Row Key*
11: **for** i=1 to n **do**
12: $p_i \leftarrow$ KS/CF/RK/$C_i$ *//Where n=number of non-key columns in CN, $C_i \in CN$ //and $C_i$ is a non-key column, $p_i$ is provenance path of $C_i$*
13: r $\leftarrow$ **Add** $p_i$ *//Adding provenance of $C_i$'s value in r*
14: P $\leftarrow$ **Append** $p_i$ *//P is comma separated list of provenance paths of all $C_i$'s in r*
15: **end for**
16: select_provenance $\leftarrow$ **Insert** $r_{tid}$, $Q_s$, P, current date/time
17: **Increment** k
18: **end for**
19: query_table $\leftarrow$ **Insert** $Q_{id}$, $Q_s$, current date/time
20: **Return** RS
21: **End**

result set, a unique result tuple id is generated by using $Q_{id}$ (refer to lines 5,7 and 17). Initially, the value of P for all columns of each result tuple is being set to the null (refer to line 8). Then the value of $V_{pk}$ is retrieved from result tuple and assigned to RK (refer to lines 9 and 10). After this, for each non-key column $C_i$ of r, provenance path expression $p_i$ is generated (i.e. KS/CF/RK/$C_i$) and added in the corresponding r and further appended in P (refer to lines 11 to 14). A provenance path expression consists of a keyspace name, column family, row key, and column name in the following form; "*keyspace/columnfamily/rowkey/columnname*". Provenance path expression provides a detailed provenance for each of the result tuple exists in the query result at different granularity levels, i.e., How a value in result tuple is derived? Finally, column families "*select_provenance*" and "query_table" are also updated (refer to lines 14 to 20). With respect to complexity of algorithm, select query takes O(log(n) time to retrieve result set 'RS'. In addition, for each column c (except primary key columns) of result tuple r∈RS, generation of provenance path expression $p_i$ takes O(r*c) time. Therefore, theoretical complexity of Algorithm 8 is O(log(n)) + O(r*c). However, its practical complexity is slightly different as it depends on the number of columns 'c'. In most cases, the proposed algorithm runs comparatively fast as only a small number of columns exist in the result set. Demonstration of Algorithm 8 with illustrative example queries 2 and 3 are given below:

***Example Query 2*:** Display the location of user with Screen_Name 'Gagan4041'.

***Cassandra Query 2*:** select location from user_details where screen_name='Gagan4041';

Query result of the above select query contains following two columns viz. "*LOCATION*" and "*LOCATION_PROVENANCE*" with values "*India*" and "*[ NewTwitter_Keyspace/ user_details/Gagan4041/ location]*" respectively. Here, the value under the column name "*LOCATION_PROVENANCE*" justifies the query result i.e. "*India*". It explains that the value in result set is derived from keyspace: *NewTwitter_Keyspace*, column family: *user_details*, row key: *Gagan40041*, column: *location*.

***Example Query 3*:** Display all the hashtags used in the tweets posted by a user with Screen_Name 'mkzangid'.

***Cassandra Query 3*:** select hashtag from user_tweet _hashtag where screen_name= 'mkzangid';

Query result of the above query is shown in Figure 5.7, which shows that the user

"*mkzangid*" used hashtag "*Vikramlander*" in two of his tweets with tweet id's "*118151081737-7767426*" and "*1181512471518990342*". Provenance path expression under column "*Hashtag_Provennace*" shows the derivation process of the value present in result set i.e. value "*Vikramlander*" in result set is derived from two different rows with row key (composite primary key of screen_name and tweet id) "*mkzangid-118151081737- 7767426*" and "*mkzangid-1181512471518990342*".

| Hashtag | Hashtag_Provenance |
|---|---|
| [Vikramlander] | [NewTWitter_Keyspace/user_tweet_hashtag/ mkzangid-118151081737767426/hashtag] |
| [Vikramlander] | [NewTWitter_Keyspace/user_tweet_hashtag/ mkzangid-1181512471518990342/hashtag] |

**Figure 5.7:** Example Query 3 Result

#### 5.4.2.2 Provenance Generation for Aggregate Queries

Proposed framework supports the capturing of provenance information for aggregate queries too. The high-level details of provenance generation algorithm for aggregate queries i.e. "*AggreProv*" are given in Algorithm 9. According to this algorithm, an Aggregate Query ($Q_a$) with its Query Id ($Q_{id}$) is passed as an input and a comma separated list of Provenance Path Expressions pv[i] for each of its result tuple exists in query result are obtained as an output in Provenance Vector (pv). The provenance path expression consists of all the source rows and column names of a column family in a keyspace that contributed to generate the corresponding result tuple. All the steps of this algorithm are very similar to Algorithm 8, i.e. "*SelectProv*" except the concept of provenance vector. Although, Provenance path is generated in the same way as in Algorithm 8, however the iteration is performed on all source rows which contributed to produce one result row in result set to generate pv[i] of all source rows (refer to lines 13 to 21). Further, provenance of result tuples and corresponding aggregate query are stored in "*select_provenance*" and "*query_table*" column families respectively. In respect of complexity of algorithms, select queries in step 4 and 5 take 2 * O(log(n)) time for execution. In addition, for each column c

**Algorithm 9** *AggreProv*: Provenance Generation for Current Aggregate Query

**Input:** Query $Q_a$ (Aggregate Query), $Q_{id}$ (Query Id)
**Output:** Comma separated list of ProvPathExps (*pv[i]*) with each value in result set, updated "select_provenance" and "query_table" column families

1: $R_p \leftarrow$ **Parse** $Q_a$
2: *KS,CF,PK,CN* $\leftarrow$ **Retrieve**($R_p$) *//Where KS=Keyspace, CF=Column_Family, //PK=Primary Key of CF, CN=Column names in $Q_a$*
3: **Generate** $Q_r$ *//$Q_r$ is rewritten aggregate query for retrieving values of //Pk and appending predicate with "valid_to" as Null*
4: RS $\leftarrow$ **Execute** $Q_a$
5: RS1 $\leftarrow$ **Execute** $Q_r$
6: k=0
7: **while** RS1 != Null **do**
8: r1=**Iterate** over RS1
9: **for all** $r \in RS$ **do**
10: $r_{tid}$ = $Q_{id}$+ 't'+ k *//$r_{tid}$ is unique id of each result tuple of $Q_r$*
11: **Set** P = Null
12: **Vector** pv = Null *//pv is a vector to store all provenance paths of r*
13: **for all** r1 $\in$ RS1 till value of aggregate attribute in r1 is same as in r **do**
14: **Obtain** $v_{pk}$ from r1 *//$v_{pk}$=Values of PK*
15: RK $\leftarrow$ $v_{pk}$ *//RK is Row Key*
16: **for** i=1 to n **do**
17: pr $\leftarrow$ KS/CF/RK/$C_i$ *//Where n=number of non-key columns in CN, $C_i \in$CN //and $C_i$ is a non-key column, pr is provenance path of $C_i$*
18: *pv[i]* $\leftarrow$ **Append** pr *//pv[i] is comma separated list of all provenance path of //$C_i$'s value in r*
19: P $\leftarrow$ **Append** pr *//P is comma separated list of provenance paths of //all $C_i$'s in r*
20: **end for**
21: **end for**
22: r $\leftarrow$ **Add** *pv* *// Adding provenance vector of all $C_i$'s value in r*
23: select_provenance $\leftarrow$ **Insert** $r_{tid}$, $Q_a$, P, current date/time
24: **Increment** k
25: **end for**
26: **end while**
27: query_table $\leftarrow$ **Insert** $Q_{id}$, $Q_a$, current date/time
28: **Return** RS
29: **End**

(except primary key columns) of result tuple r$\in$RS, generation of provenance path expression pv[i] takes O(r*c) time. Therefore, theoretical complexity of Algorithm 9 is O(log(n)) + O(r*c). However, practical complexity is slightly different, as it depends on the number of columns 'c' in the result set 'r'. In case of aggregate queries, the proposed algorithm runs fast as the number of columns is usually one. Demonstration of Algorithm 9 with illustrative example queries 4 and 5 are given below:

***Example Query 4*:** Display the total no of tweets posted by a user "sunilthalia" on "08/10/2019".

***Cassandra Query 4*:** select count(tweet_body) from tweets_user_day where screen_name= 'sunilthalia' and published_day=8 and published_date>= '2019-10-08' and published_date <'2019-10-09' group by screen_name allow filtering;

The above query is an example of aggregate query to retrieve the total number of tweets posted by a specific user on a given day. This aggregate query efficiently executes on "*tweets_user_day*" column family with composite primary key i.e. "*screen_ name*, *published_day*, and *published_date*". Figure 5.8 shows the partial result of above aggregate query where the total number of tweets posted by the given user on 08/10/2019 are 7 (mentioned under the column name "*SYSTEM.COUNT(TWEET_BODY)*") along with a comma separated list of provenance path expressions for all the 7 rows with the name of column families that have contributed to the result set under the column name "*SYSTEM.COUNT( TWEET_BODY )_PROVENANCE*".

| SYSTEM.COUNT (TWEET_BODY) | SYSTEM.COUNT(TWEET_BODY) _PROVENANCE |
|---|---|
| 7 | [NewTwitter_Keyspace/tweets_user_day/sunilthalia-8-Tue Oct 08 11:37:56 IST 2019/tweet_body, NewTwitter_Keyspace/tweets_user_day/sunilthalia-8-Tue Oct 08 11:40:12 IST 2019/tweet_body, NewTwitter_Keyspace/tweets_user_day/sunilthalia-8-Tue Oct 08 11:48:33 IST 2019/tweet_body, NewTwitter_Keyspace/tweets_user_day/....] |

**Figure 5.8:** Example Query 4 Result

***Example Query 5*:** Display the total no of tweets posted on each day in month of October, 2019.

***Cassandra Query 5*:** select published_day, count(tweet _body) from tweets_day where published_date>='2019-10-01' and published_date<'2019-11-01' group by published_day allow filtering;

The above aggregate query executed on "*tweets_day*" column family with composite primary key i.e. "*published_day*, *published_date*", and counts the total number of tweets

posted on each day of October, 2019. Partial result of above aggregate query is shown in Figure 5.9, where the total number of tweets posted on each day is shown under the column name "*SYSTEM.COUNT (TWEET_BODY)*", along with the tweets posted day, and a comma separated list of provenance path expression for all the rows that contributed towards aggregated result under the column name "*SYSTEM.COUNT (TWEET_BODY)_PROVENANCE*".

| PUBLISHED_DAY | SYSTEM.COUNT (TWEET_BODY) | SYSTEM.COUNT(TWEET_BODY)_PROVENANCE |
|---|---|---|
| 23 | 5 | [NewTwitter_Keyspace/tweets_day/23-Wed Oct 23 12:35:17 IST 2019/tweet_body, NewTwitter_Keyspace/tweets_day/23-Wed Oct 23 12:38:14 IST 2019/tweet_body, NewTwitter_Keyspace/tweets_day/...] |
| 5 | 34 | [NewTwitter_Keyspace/tweets_day/5-Sat Oct 05 00:01:18 IST 2019/tweet_body, NewTwitter_Keyspace/tweets_day/5-Sat Oct 05 04:46:01 IST 2019/tweet_body, NewTwitter_Keyspace/tweets_day/...] |
| 10 | 8 | [NewTwitter_Keyspace/tweets_day/10-Thu Oct 10 01:09:32 IST 2019/tweet_body, NewTwitter_Keyspace/tweets_day/10-Thu Oct 10 07:03:30 IST 2019/tweet_body, NewTwitter_Keyspace/tweets_day/...] |
| 30 | 1 | [NewTwitter_Keyspace/tweets_day/30-Wed Oct 30 14:23:14 IST 2019/tweet_body] |
| 13 | 2 | [NewTwitter_Keyspace/tweets_day/13-Sun Oct 13 00:37:02 IST 2019/tweet_body, NewTwitter_Keyspace/tweets_day/13-Sun Oct 13 10:51:20 IST 2019/tweet_body] |

**Figure 5.9:** Example Query 5 Result

#### 5.4.2.3 Provenance Generation for Historical Queries

Proposed framework also supports to capture provenance information for historical queries using data versioning support in ZILKVD. The high-level details of provenance generation algorithm for historical queries i.e. "*HistProv*" are given in Algorithm 10, where a Historical Query ($Q_h$) along with its Time of Execution(t) is passed as an input and a comma separated list of Provenance Path Expressions ($p_i$) for all result tuples are obtained as an output. Initially, query $Q_h$ is parsed to retrieve following information viz., Keyspace, Column Family, Column Names, Primary Key etc (refer to lines 1 and 2). Afterwards, a Rewritten Select Query ($Q_r$) is generated to retrieve Row Key (RK) i.e. values of primary key column of column family, and each result tuple with predicate "valid_to". The value of this predicate is set to time "t" (i.e. given in input) and then query $Q_r$ is executed on

**Algorithm 10** ***HistProv***: Provenance Generation for Historical Query

**Input:** Query $Q_h$ (Historical Query), t (Time of Execution)
**Output:** ProvPathExp ($p_i$) for each value of result set.

```
1: R_p ← Parse Q_h
2: KS,CF,PK,CN ← Retrieve(R_p)   //Where KS=Keyspace, CF=Column_Family,
                                 //PK=Primary Key of CF, CN=Column names in Q_h
3: Generate Q_r                  //Q_r is rewritten select query for retrieving values of Pk and
                                 //appending predicate with "valid_to" as "t"
4: RS ← Execute Q_r
5: for all r ∈ RS do
6:    Obtain v_pk                //v_pk=Values of PK
7:    RK ← v_pk                  //RK is Row Key
8:    for all v in r do
9:       if writetime(v)<=t then
10:         p_i ← KS/CF/RK/C     //Where v is value of column C in r and C ∈ CN,
                                 //p_i is provenance path of C
11:         r ← Add p_i          //Adding provenance of C_i's value in r
12:      else
13:         Generate Q_s         //Q_s is rewritten select query for retrieving value of C (v_c) from
                                 //update_provenance column family with predicate "valid_to" as "t"
14:         RS1 ← Execute Q_s
15:         p_i, v_c ← Retrieve from RS1 //Where p_i is provenance path of value v_c of column C
16:         r ← Update v with v_c in r
17:         r ← Add p_i
18:      end if
19:   end for
20: end for
21: Return RS
22: End
```

database (refer to lines 3 to 7). Now, for every value in result set of $Q_r$, its "*writetime*" (time of existence in database) is compared with "*t*". If "*writetime*" is less than or equal to "*t*" then provenance path expression ($p_i$) is generated with corresponding source row and column contributed towards its generation and further, added in result tuple (refer to lines 9 to 11). But, if "*writetime*" is greater than "*t*" then corresponding column value and provenance path is retrieved from "*update_provenance*" column family (refer to lines 13 to 15). In the end, the value of column and provenance path expression that retrieved from "*update_provenance*" column family are updated in result set and finally, updated result set along with provenance information is obtained (refer to lines 16 to 21). In proposed algorithm, select query takes O(log(n)) time to retrieve required result set 'RS'. In addition, for each column c (except primary key columns) of result tuple r∈RS, generation of provenance

path expression $p_i$ takes O(r*c) or O(r*log(n)) time. Therefore, theoretical complexity of this algorithm is O(log(n)) + O(r*c). However, practical complexity is slightly different, as it depends on the number of updates performed.

### 5.4.3 Provenance Storage

In the proposed framework, all the captured provenance is stored in the following three column families of Apache Cassandra for further analysis viz. "*query_table*", "*select_provenance*", and "*update_provenance*" (refer to Figure 5.10). Provenance information of all the executed queries with their query id and time of executions is stored in "*query_table*" column family. Provenance path expressions for all the result tuples of select/aggregate queries are stored in "*select_provenance*" column family along with their query statement, result tuple id and time of executions as shown in Figure 5.11. Similarly, the column family "*update_provenance*" keeps the provenance information about all the update operations along with following attributes i.e. *query statement*, *provenance path expression*, *old value* and *its write time*, *new value*, *column type*, and *time of update (current date/time)* (refer to Figure 5.3). The captured provenance is used in source tracing, update tracking, and in querying historical data. Further, the visualization of this provenance data is helpful in analysing and determining the truthiness of a query result.

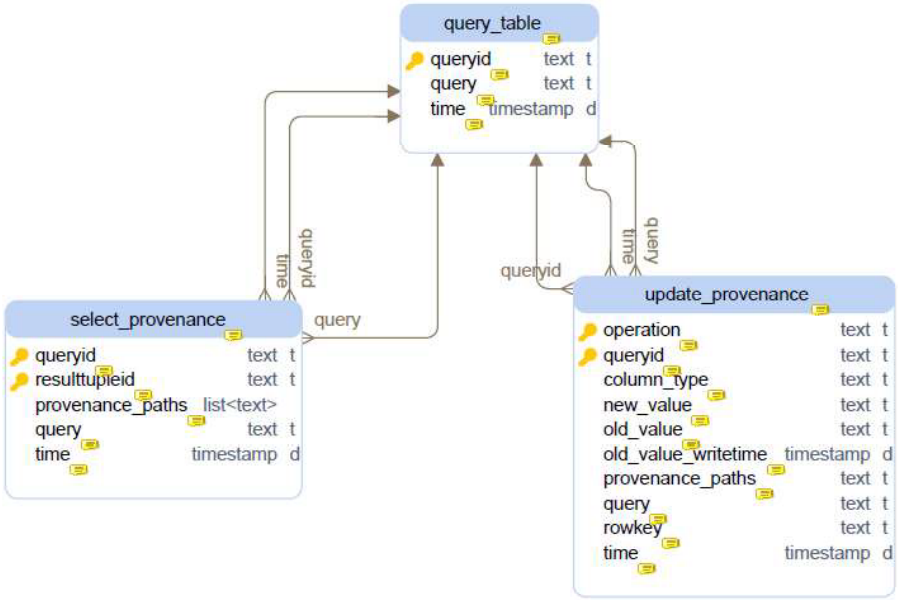


**Figure 5.10:** Provenance Storage

| Query Id | Query | Result tupleid | Provenance_path_exp | Time |
|---|---|---|---|---|
| q6 | select location from user_details where screen_name='Gagan4041' | q6t1 | ['NewTwitter_Keyspace/user_details/Gagan4041/location'] | 2019-12-16 05:02:12 |
| q7 | select screen_name, published_day, count(tweet_body) from tweets_user_day where published_day=8 and published_date>= '2019-10-08' and publishcd_datc< '2019-10-09' group by screen_name allow filtering | q7t1 | ['NewTWitter_Keyspace/tweets_user_day/barrotchetan999-8-Tue Oct 08 01:38:10 IST 2019/tweetbody'] | 2019-12-18 04:54:02 |
| q7 | select screen_name, published_day, count(tweet_body) from tweets_user_day where published_day=8 and published_date>= '2019-10-08' and published_date< '2019-10-09' group by screen_name allow filtering | q7t2 | ['NewTWitter_Keyspace/tweets_user_day/asifawan8080780-8-Tue Oct 08 01:38:10 IST 2019/tweetbody'] | 2019-12-18 04:54:02 |
| q7 | select screen_name, published_day, count(tweet_body) from tweets_user_day where published_day=8 and published_date>= '2019-10-08' and published_date< '2019-10-09' group by screen_name allow filtering | q7t3 | ['NewTWitter_Keyspace/tweets_user_day/ahmedtanzeel105-8-Tue Oct 08 01:38:10 IST 2019/tweetbody'] | 2019-12-18 04:54:02 |

**Figure 5.11:** A Snapshot of "select_provenance" Column Family

### 5.4.4 Querying Provenance

The proposed framework also supports querying of provenance information for various purposes. Provenance querying on captured provenance are carried out to achieve the following two objectives; First, *How any result tuple of select query is derived?* i.e. querying provenance to know about the source of information, and Second, *How to track all the updates performed on a given data?*, i.e., querying provenance for historical data. The framework provides following two column families to accomplish the above tasks viz., "*select_provenance*" and "*update_provenance*". Provenance path expressions for all the result tuples of select/aggregate queries along with their query statement, result tuple id and time of executions are stored in "*select_ provenance*" column family. This provenance information is used in provenance querying to know about the source of information, as shown in Figure 5.11. Similarly, the column family "*update_provenance*" stores the provenance information about all the update operations performed along with the following parameters i.e. query statement, provenance path expression, old value and its write time, new value, column type, and time of update (current date/time). This provenance information is used in provenance querying for historical data (refer to Figure 5.3). In addition to above column families, one more column family i.e. "*query_table*" is also used in provenance querying to obtain the information about all the queries executed till a particular date with their time of execution. The illustrative examples of provenance querying are

given below:

***Example provenance Query PQ1*:** Explain how result tuple q6t1 of query q6 (as shown in Figure 5.11) is derived?

The above query is executed on "*select_provenance*" column family to retrieve provenance path expressions for result tuple q6t1 of query q6 along with its time of execution. Here, provenance path expression of resultant tuple is "*[NewTwitter_Keyspace/ user_details/ Gagan4041/location]*" and time of query execution is "*2019-12-16 05:02:34.266000+0000*". This indicates that the source keyspace name of required tuple is "*NewTwitter_Keyspace*", name of column family is "*user_details*", row key is "*Gagan4041*", column name is "*location*" and time of query execution is "*2019-12-1605:02:34.26600 0+0000*". Now, "user_details" column family is queried with this row key, column name and execution time to retrieve all the rows that contributed to produce the result tuple t1 of query q6 which justify the resultant tuple. However, if the source has been modified after query execution, in that case, the original source can still be devised through querying historical data. To support provenance querying for historical data, we designed following four User-Defined CQL Constructs (UDCs) viz., "*all*", "*instance*", "*validon now*", and "*validon date*". These constructs are further categorized in following two categories viz., T1("all", "instance") and T2 ("validon now", "validon date").

The high level details of provenance querying algorithm for historical data i.e. "*Query-Prov_HistData*" are given in Algorithm 11, in which an Extended Query ($Q_E$) (i.e. a CQL query with UDCs) is passed as an input and a corresponding Result Set (RS) of historical data is obtained as an output. In the beginning, $Q_E$ is sent to the Query Parser to retrieve all the UDCs (T1 and T2) used in $Q_E$ along with the CQL Query Q (i.e. CQL query without UDCs) and parsed result ($R_p$) (refer to lines 1 and 2). In addition to this, some other information such as Keyspace Name (KS), Column Family (CF), Primary Key (PK), and Column Name (CN) associated with $Q_E$ are also extracted from $R_p$ (refer to line 3). Now, query Q executes on the related column families to retrieve required historical data as per the following conditions mentioned in lines 4 to 16. If UDC T1 and T2 are "*instance*" and "*validon now*" type constructs respectively, then query Q executes on the column families mentioned in issued query statement only (refer to lines 4 and 5). If UDC T1 and T2 are "*instance*" and "*validon date*" type constructs respectively, then the "*write time*" of current

**Algorithm 11** ***QueryProv_HistData***: Querying Provenance for Historical Data

**Input:** Query $Q_E$ (Provenance Query with extended user-defined constructs T1,T2)
**Output:** RS(Result Set) of $Q_E$

1: $R_p$, T1, T2 ← **Parse** $Q_E$ //$R_p$ *is parsed query result, T1="instance"/"all", //T2="validon now"/"validon date"*
2: **Generate** Q //*Q is the corresponding query to* $Q_E$ *without user-//defined constructs*
3: *KS,CF,PK,CN* ← **Retrieve** $R_p$ //*Where KS=Keyspace, CF=Column_Family, //PK=Primary Key of CF, CN=Column name in Q*
4: **if** $(T1 = instance) AND (T2 = validonnow)$ **then**
5: RS ← **Execute** Q on *CF* //*Where RS is ResultSet*
6: **else if** $(T1 = instance) AND (T2 = validondate)$ **then**
7: **Generate** $Q_r$ //$Q_r$ *is a rewritten select query corresponding to Q with Time T2 to //retrieve value of CN from update_provenance(UP) Column Family*
8: RS ← **Execute** $Q_r$ on *UP*
9: **if** *RSisNull* **then**
10: RS ← **Execute** Q on *CF*
11: **end if**
12: **else**
13: **Generate** $Q_r$ //$Q_r$ *is a rewritten select query corresponding to Q with Time T2 to //retrieve value of CN from update_provenance(UP) Column Family*
14: RS ← **Execute** $Q_r$ on *UP*
15: RS1 ← **Execute** Q on *CF*
16: RS ← **Append** RS1
17: **end if**
18: **Return** RS
19: **End**

value is first fetched and compared with "*validon date*". If the "*write time*" of current value is lesser than "*validon date*" then query Q executes on the column families mentioned in issued query statement only, otherwise it executes on "*update_proven- ance*" (refer to lines 6 to 10). If UDC T1 and T2 are "*all*" and "*validon now*" type constructs respectively, then query Q executes on both "*update_provenance*" and the column families mentioned in issued query statement to retrieve the complete history of all the updates of a column value (refer to lines 13 to 16). Similarly, If UDC T1 and T2 are "*all*" and "*validon date*" type constructs respectively, then again "*write time*" of current value is fetched and compared with "*validon date*". If the "*write time*" of current value is lesser than "*validon date*" then query Q executes on both "*update_provenance*" and the column families mentioned in issued query statement otherwise it executes only on "*update_provenance*" (refer to lines 13 to 16). In

proposed algorithm, issued select query takes O(log(n)) time to search required information in respective column family as mentioned in its query statement and O(log(n)) time is taken to search in "*update_provenance*" column family. In this way, the theoretical complexity of this algorithm is O(log(n)). Demonstrations of Algorithm 11 with illustrative examples of provenance queries 2, 3, 4 and 5 are given below:

***Example Provenance Query PQ2****: Display all the location updates of a specific user named 'MemeBaaaz' till now.*

***Extended CQL Query $Q_E$:*** *select all location from user_details where screen_name='MemeBaaaz' validon now;*

The above $Q_E$ is parsed first to retrieve all the UDCs used in this extended query i.e. "*all*" and "*validon now*" respectively. Now, CQL query Q is executed on "*user_ details*" and "*update_provenance*" column families to retrieve all the location updates of the given user "*MemeBaaaz*". The query result of above provenance query is shown in Table 5.1.

| LOCATION | VALID_FROM |
|---|---|
| Meme Ki Duniya, India | Wed Oct 02 13:33:27 IST 2019 |
| Kolkata | Wed Oct 23 08:20:18 IST 2019 |
| Mumbai | 2019-12-17 10:22:22.0 |

**Table 5.1:** Example Provenance Query PQ2 Result

***Example Provenance Query PQ3****: Display all the location updates of a specific user named 'MemeBaaaz' till 23/10/2019 9:50AM.*

***Extended CQL Query $Q_E$:*** *select all location from user_details where screen_name='MemeBaaaz' validon 2019-10-23 09:50:16.*

The query result of above provenance query is shown in Table 5.2, i.e. all the location updates till '2019-10-23 09:50:16'.

| LOCATION | VALID_FROM |
|---|---|
| Meme Ki Duniya, India | Wed Oct 02 13:33:27 IST 2019 |
| Kolkata | Wed Oct 23 08:20:18 IST 2019 |

**Table 5.2:** Example Provenance Query PQ3 Result

***Example Provenance Query PQ4****: Display the current location of a specific user named 'Meme-Baaaz'*

***Extended CQL Query $Q_E$:*** *select instance location from user_details where screen_name='MemeBaaaz' validon now.*

The above provenance query generates current location of user as 'Mumbai' that is valid from '2019-12-17 10:22:22.0' as shown in Table 5.3

| LOCATION | VALID_FROM |
|---|---|
| Mumbai | 2019-12-17 10:22:22.0 |

**Table 5.3:** Example Provenance Query PQ4 Result

***Example Provenance Query PQ5****: Display the location of a specific user named 'MemeBaaaz' on date 23/10/2019 8:22:16AM.*

***Extended CQL Query $Q_E$:*** *select instance location from user_details where screen_name='MemeBaaaz' validon 2019-10-23 08:22:16.*

The above query generates location of user at 23/10/2019 8:22:16AM as "Kolkata" which is valid from "Wed Oct 23 08:20:18 IST 2019" as shown in Table 5.4.

| LOCATION | VALID_FROM |
|---|---|
| Kolkata | Wed Oct 23 08:20:18 IST 2019 |

**Table 5.4:** Example Provenance Query PQ5 Result

## 5.5 Experimental Setup and Results

### 5.5.1 Experimental Setup

To evaluate the performance of proposed framework, all the experiments are performed on a single node Apache Cassandra Cluster on Intel i7-8700 processor @ 3.20GHz with 16GB RAM, and 1TB disk. Apache Cassandra version 3.11.3 has been used for the experiments. In the proposed framework, data is fetched from the Twitter's network through live streaming and modelled in Apache Cassandra. This big social data consists of around 2.4 lakh twitter users, 2.1 lakh user's friends, 1.8 lakh user's followers, and their related information such as tweet's body, tweet's id, tweeter's screen name, tweet created date, user's personal information etc. The proposed key value data model contains a keyspace named *"NewTwitter_Keyspace"* that consists of 20 Column Families those are used to store this huge volume of social data. On execution of each query the provenance information is captured and stored in the following three column families viz. *"select_provenance"*, *"update_provenance"*, and *"query table"* that gradually increases the size of database. Java version 8 has been used as front-end programming language to interact with Cassandra, and Twitter's network. Cassandra Query Language (CQL) is used for querying and to communicate with Apache Cassandra.

### 5.5.2 Results and Discussions

The performance analysis of proposed framework in terms of provenance capturing overhead and provenance query execution time for different query sets including, select, aggregate, data update and provenance queries are presented below.

#### 5.5.2.1 Provenance Capture Analysis

To perform an experimental analysis of provenance capture, several query sets of various queries including select, aggregate, and data update queries are executed on ZILKVD architecture. A sample set of select queries are shown in Table 5.5. All the queries are executed 12 times without provenance support and then, the same sets of queries are again executed 12 times with provenance support. To calculate the average execution time of each query, we dropped the minimum and the maximum execution times, and

| QId | Query |
|---|---|
| Q1 | Find location of user with Screen_Name=‘ Gagan4041’. |
| Q2 | Display all tweets by user with screen_name= ‘SunilThalia’. |
| Q3 | Display all hashtags used by a user in one tweet. |
| Q4 | Display all hashtag used by a user in all tweets posted by a user. |
| Q5 | Display all tweets posted by a user on one particular day. |

**Table 5.5:** Sample Select Queries

then taken the average of remaining 10 values. Average execution times of all the queries are mentioned in milliseconds (ms). The execution performance of all the select queries is shown in Figure 5.12, where it is found that the performance overheads in terms of execution time for the queries with provenance support is very limited with respect to the queries without provenance support, except for query Q8. As the proposed framework captures and stores the provenance information of all the result tuples which exist in the result set of a query, the execution time increases with the increase in the number of result tuples. That's why the query Q8 with provenance support is taking longer to execute, as it is producing a large number of result tuples.

The proposed framework also provides the provenance support for several aggregate queries those are using various aggregate functions such as count, max, min etc. A sample set of aggregate queries are shown in Table 5.6. The performance analysis of aggregate

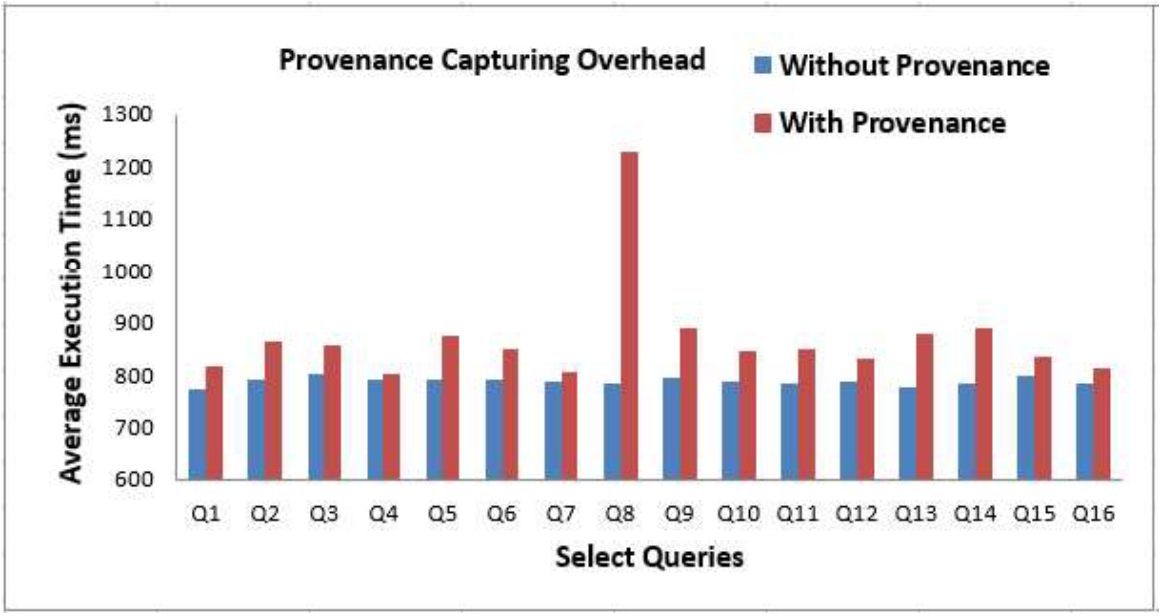


**Figure 5.12:** Performance of Select Queries without and with Provenance

| QId | Query |
|---|---|
| Q1 | Display total no of tweets posted by a user on one particular day of a month. |
| Q2 | Display total no of tweets posted by a user in one month. |
| Q3 | Display total tweets posted in one specific month. |
| Q4 | Display all the users with tweets count in a specific month in descending order. |
| Q5 | Display total no of tweets posted every day of a specific month. |

**Table 5.6:** Sample Aggregate Queries

queries in terms of average execution time with and without provenance support is also shown in Figure 5.13. Although, the framework efficiently captures and stores provenance information for aggregate queries such as query Q1, Q2, and Q4, yet it takes more time to execute for the queries in which aggregation is performed on a large number of input tuples, such as queries Q3, and Q5. For example, let's consider the case of query Q3: "count the total number of tweets posted in one month". Here, as the aggregation is performed on all the tweets of that month, which requires capturing and storing the provenance information of all such rows which have contributed to the result set. As a consequence, execution overhead increases.

The provenance capturing for data update queries is also supported by the proposed framework using ZILKVD. A sample set of data update queries are shown in Table 5.7. We executed a set of data update queries to capture and store their provenance information in "*update_provenance*" column family. The following parameters are used to capture the provenance information such as "*value_type*", "*old_value*", "*new_value*",

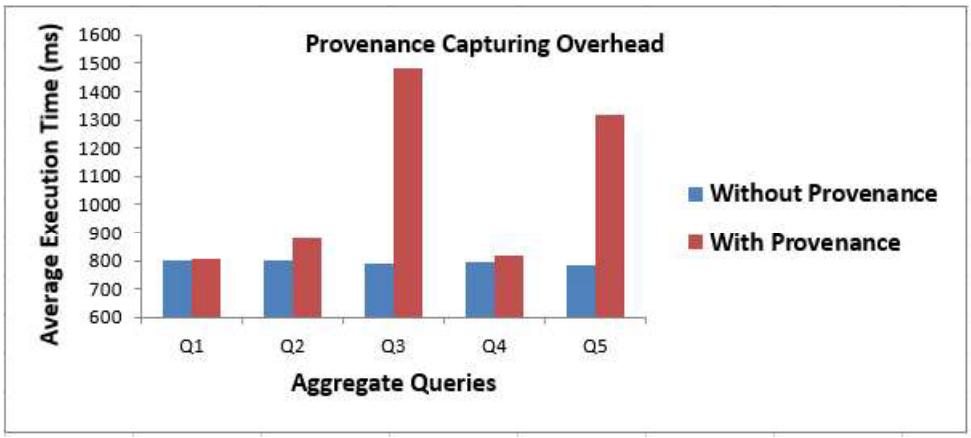


**Figure 5.13:** Performance of Aggregate Queries without and with Provenance

| QId | Query |
|---|---|
| Q1 | Update location of user with screen_name "DDNewsAndhra". |
| Q2 | Update location of friend named "Ashutosh" of user with screen_name "Bandho". |
| Q3 | Update url of user with screen_name "myshowmytalks". |
| Q4 | Delete user with screen_name "DDNewsAndhra". |
| Q5 | Insert a posted tweet in tweetset. |

**Table 5.7:** Sample Update Queries

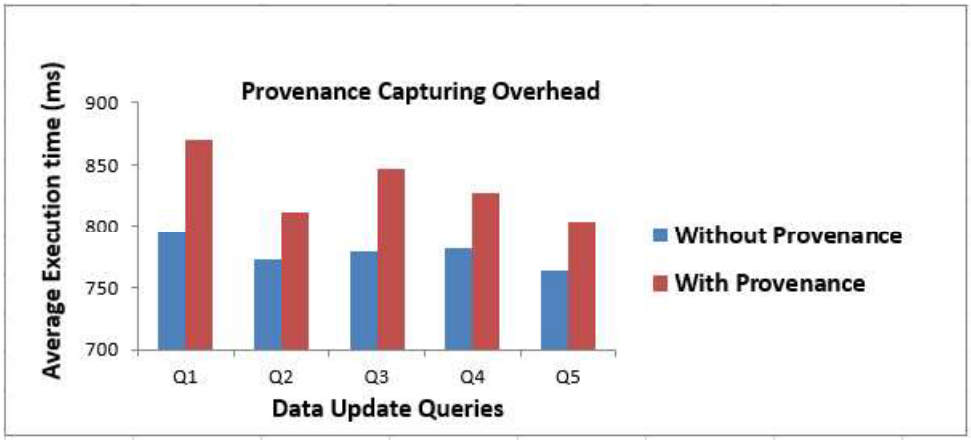


**Figure 5.14:** Performance of Update Queries without and with Provenance

"*old_value_writetime*", and "*provenance_path_expression*" etc. These parameters can be used for historical data queries, and queries executed in the past at any specific time i.e. historical queries as explained in previous section 5.4.2.3. Like select, and aggregate queries, the proposed framework also supports efficient provenance capturing and storing for data update queries with a very little execution time overhead (refer to Figure 5.14).

Ultimately, the overall execution performance of all types of queries with and without provenance capturing support is shown in Figure 5.15. The average query execution time for all "*update*", "*select*", and "*aggregate*" queries with and without provenance support are mentioned in milliseconds. The proposed framework is very efficient for "*update*", and "*select*" queries, while a small overhead is associated with "*aggregate*" queries (refer to Figure 5.16).

#### 5.5.2.2 Provenance Querying Analysis

The performance analysis of querying provenance information stored in Apache Cassandra is presented in this section. A set of different provenance queries are executed for

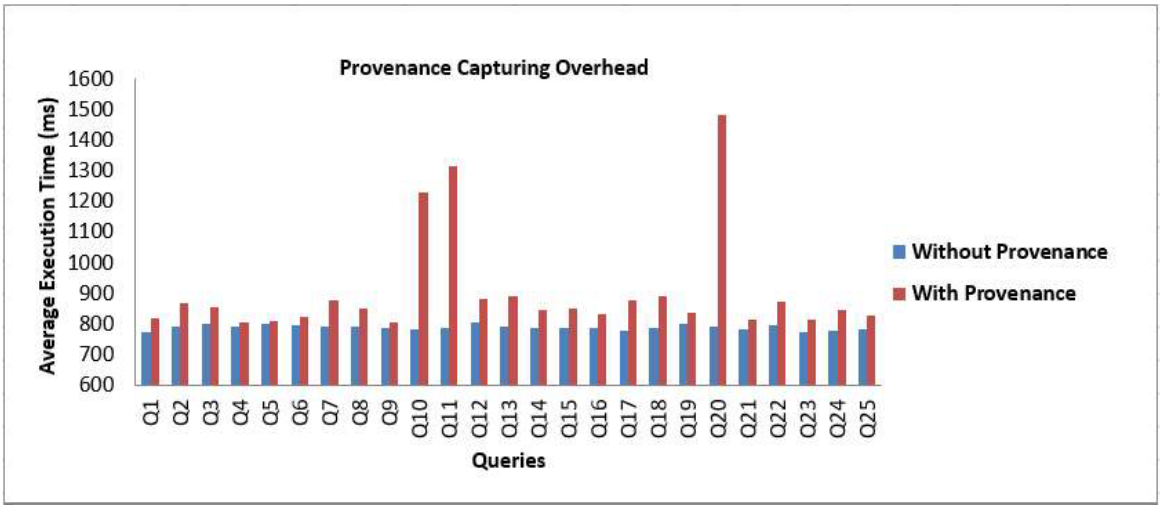


**Figure 5.15:** Overall Query Performance without and with Provenance

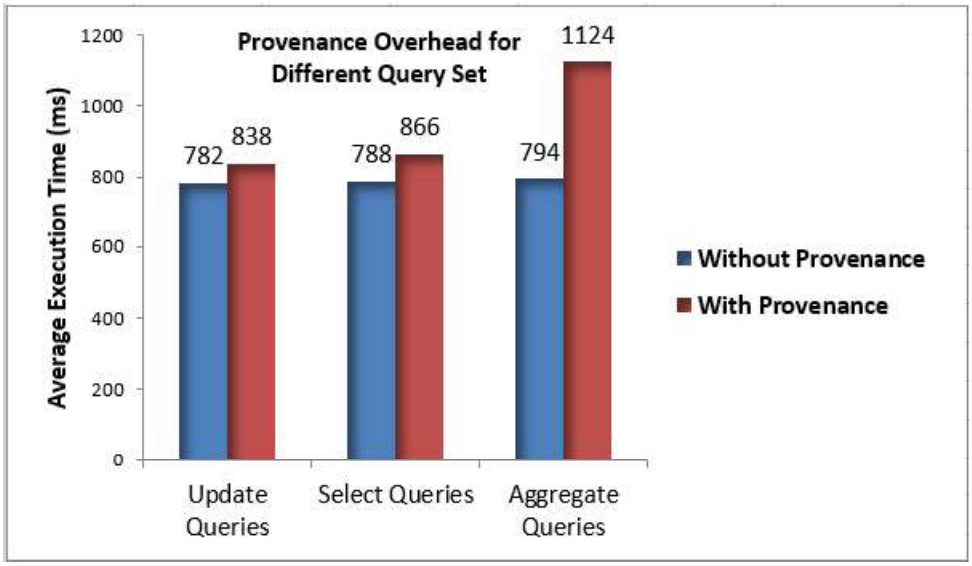


**Figure 5.16:** Provenance Overhead for Different Query Sets

analysis of provenance querying. A sample set of provenance queries are shown in Table 5.8. Initially, all the provenance queries are executed 12 times. To calculate the average

| QueryId | Query |
|---|---|
| Q1 | Display all the rows contributed to produce result tuples of query Q2. |
| Q2 | Display the row keys of all the rows those are contributed to produce result tuple t1 of query Q11. |
| Q3 | Display all location updates of a specific user till now. |
| Q4 | Display all location updates of a specific user till time 22/10/2019 8:00AM. |
| Q5 | Display the location of a specific user at time 22/10/2019 8:00AM. |

**Table 5.8:** Sample Provenance Queries

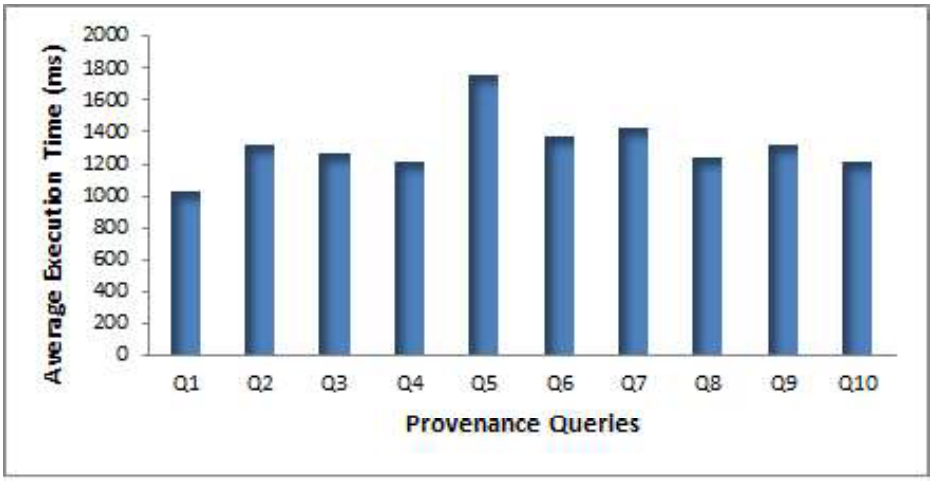


**Figure 5.17:** Provenance Querying

execution time of each query, we dropped the minimum and the maximum execution times, and then taken average of remaining 10 values. Average execution times of all the provenance queries are mentioned in milliseconds (ms) (refer to Figure 5.17. It shows that the proposed framework efficiently supports provenance querying for both source/origin tracing and historical data queries.

## 5.6 Application Scenarios

Several illegitimate activities are engendered by misusing these social contents to accomplish various, sometimes ignoble, objectives. One of the main causes behind these illegitimate activities on social media is the separation of digital contents from its provenance. The credibility of an analysis generally depends upon the quality and truthiness of input data which can be assured by the Big Social Data Provenance. Our proposed big social data provenance framework is capable to generate and visualize the provenance information in different scenarios such as social data analytics, preserving progressive user profiles etc. Some of the application scenarios are listed below:

1. **Information Discovery:** Provenance information is very much 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 source. The proposed provenance framework can be efficiently applied in information discovery applications by visualizing provenance informa-

tion from source to destination or vice versa. The proposed framework can be easily extended further for multi-layer provenance generation and multi-depth provenance querying to visualize the provenance with varying depths.

2. **Preserving Progressive User Profiles:** A social media user can update his/her profile or may add new information or remove any existing information at any time. For data analytics purpose, application may require when any data is inserted, updated, or removed along with maintaining all the data updates performed. Our proposed framework is applicable for such kind of applications where progressive user profile maintenance is needed.

## 5.7 Conclusions and Future Work

In this chapter, we designed and implemented a Zero-Information Loss Key-Value Pair Database (ZILKVD) on top of which a Big Social Data Provenance (BSDP) Framework has been developed to capture and querying provenance for live streamed Twitter data set. The proposed framework is capable to capture fine-grained provenance for various query sets including select, aggregate, and data update queries with insert, delete, and update operations. It also supports to capture provenance for historical queries using data version support in ZILKVD. The proposed ZILKVD architecture and KVP data model leads to an adequate design methodology that provides a flexible provenance management system for social data.

The proposed framework is efficient in terms of average execution time for capturing and storing provenance for select, and data update queries. However, a small execution overhead is measured for some aggregate queries, where the aggregation is performed on a larger number of input tuples. Proposed framework supports efficient provenance querying for both justifying answers of a query result, and historical data queries. Our provenance capturing and querying algorithms prove to be very promising; retrieving more precise information with an acceptable latency.

However, our framework has following limitations. It provides single layer provenance support (i.e. Tracing out direct sources that contributed to a query result) at this stage. Second, currently BSDP framework is implemented for a single node Apache Cassandra

rather than for several distributed nodes in a cluster.

In the future, we plan to extend BSDP framework for multi-layer provenance support (i.e. Tracing out both direct and indirect sources that contributed to a query result) by using multi-depth provenance querying. We also plan to further extend our framework for a distributed environment, where data is redundantly stored across multiple nodes in a cluster.