Design and implementation of a Data Stream Management System

Nagarjuna Malempati

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Master of Technology

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Department of Computer Science and Engineering

June 2012

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

[illegible]

(Signature)

MALEMPATI. NAGARJUNA

(Nagarjuna Malempati)

CS10M03

(Roll No.)

Approval Sheet

This thesis entitled Design and implementation of a Data Stream Management System by Nagarjuna Malempati is approved for the degree of Master of Technology/ Doctor of Philosophy from IIT Hyderabad.

Ch. Sobhan babu

ch. Sobhanbabu

-Name and affiliation-

Examiner

-Name and affiliation-

Examiner

[illegible]

Ravindra Guravannavar

-Name and affiliation-

Adviser

-Name and affiliation-

Co-Adviser

[illegible]

-Name and affiliation-

Chairman

Acknowledgements

I express my sincere gratitude towards my advisor Dr. Ravindra N Guravannavar and Co-Advisor Dr. Naveen Sivadasan for their constant help, encouragement and inspiration throughout the thesis work. Without their invaluable guidance, this work would never have been a successful one. I am thankful to Mr. Ajit Aluri (B.Tech student) for helping me in design and implementation of this system.
No amount of gratitude to my parents would be sufficient. I am very fortunate to be where I am today because of their constant support and encouragement.

Dedication

This thesis is dedicated to my father (*Govinda Rao M*), who taught me that the best kind of knowledge to have is that which is learned for its own sake.

It is also dedicated to my mother (*Bhagya Lakshmi M*), who taught me that even the largest task can be accomplished if it is done one step at a time.

It is also dedicated to my sister (*Parvathi M*), who let me know that my efforts really are worthwhile.

It is also dedicated to all my teachers, lectures and professors who taught me not only knowledge, but also taught me to be a good person in the society.

Abstract

The amount of data stored by companies has grown exponentially over the last decade. Of late, data is being continuously collected for various purposes - click stream analysis, credit card transactions for fraud detection, weather monitoring, stock tickers in financial services, link statistics in networking, user logins and web surfing statistics, highway traffic congestion analysis and so on. The data that is being collected is in the form of a stream - arrives continuously, at a variable rate, and can occupy potentially infinite storage. As organizations have realized that fast and efficient processing of this data can help in profitable predictions, there exists a need for developing systems to handle this collected data effectively.

We present in this thesis, the architecture of a generic Database Stream Management System (DBSMS) to handle streaming data. While literature has provided insights into Data Stream Management Systems (DSMS), the DBSMS is a different approach that tries to integrate a DSMS with the traditional Database Management Systems (DBMS). We discuss the need for such a generic DBSMS and present the system that we have implemented using the discussed architecture. We also present the performance of our system, in terms of space taken, time taken to answer a query and the accuracy of the result compared to a DBMS. Finally, we conclude with brief discussion on certain goals and open challenges that are of interest and which still need to be addressed by the system.

Contents

Declaration ii
Approval Sheet iii
Acknowledgements iv
Abstract vi

Nomenclature **ix**

1 Introduction **1**
1.1 Terminology 3
1.2 Challenges 4
1.3 Organization 4

2 Literature Survey **6**
2.1 Streaming Algorithms 6
2.1.1 Count-Min Sketch 7
2.1.2 Point Query 8
2.1.3 Range Query 10
2.1.4 Heavy Hitters 10
2.2 STREAM 11

3 Database Stream Management System Architecture **12**
3.1 DBSMS architecture 12
3.2 Types of queries 13
3.2.1 One-Time query 14
3.2.2 Continuous query 14
3.3 Query language 15
3.3.1 Registering a stream 15
3.3.2 Registering a query 16
3.3.3 Querying the result 16
3.4 Query execution and conversion from one type of query to another type . . 17
3.5 Query processing structure 18
3.6 Project goals 19

4 Query mapping and Integrating algorithms into Query Plan 20

4.1 Query mapping into approximation operators 20

4.1.1 Streaming statistics queries 20

4.1.2 Streaming aggregate queries 21

4.1.3 Streaming data queries 21

4.2 Integrating streaming algorithms into the streaming database 22

4.3 User controlled approximation 22

4.4 Execution structure of user defined algorithm 23

4.4.1 Streams collection 23

4.4.2 Queries collection 23

4.4.3 StreamSource and OperatorQueueThread communication structure . 24

5 System design 25

5.1 Parser component 25

5.2 Input component 25

5.2.1 Register request 26

5.2.2 Action request 26

5.2.3 QueryResult request 26

5.2.4 Show request 26

5.3 Configuration component 26

5.4 Stream component 26

5.4.1 Stream Element 26

5.4.2 Stream source 27

5.4.3 Operator queue thread 27

5.4.4 Streams collection 27

5.4.5 Queries collection 27

5.5 Algorithm component 27

5.6 Process to plug in user defined stream source 27

5.7 Process to plug in user defined algorithm operator 28

6 Performance and Results comparison 29

7 Conclusion and Future Work 35

7.1 Error Propagation 35

7.2 Query mapping and Feasibility 35

7.3 Multi-Query Optimization 36

References 37

Appendices **39**
.1 Input component 39
.1.1 Register request 40
.1.2 Action request 41
.1.3 QueryResult request 42
.1.4 Show request 42
.2 Configuration component 43
.3 Stream component 43
.3.1 Stream Element 43
.3.2 Stream source 44
.3.3 Operator queue thread 44
.3.4 Streams collection 44
.3.5 Queries collection 45
.4 Algorithm component 46
.5 Process to plugging in user defined stream source 46
.6 Process to plugging in user defined operator 47

Chapter 1

Introduction

Owing to the technological advancements of the 21st century, the previous decade has witnessed a huge increase in the amount of data that is being exchanged, stored and processed across the globe. As more and more organizations began to understand that processing of collected data has a direct business impact, (in terms of understanding consumer preferences), there has been a sudden interest in finding solutions that can handle voluminous amount of data and process them at a fast rate. Given the high speeds of data transfer that are now available at low prices, there is a need to develop new techniques to handle and answer queries on these high-speed data-streams effectively.

As a motivating example,we present the Internet advertising business. There are three main stake holders in this business - the advertisers, the publishers and the commissioners. The commissioners render advertisements to publishers who get paid by advertisers for displaying the advertisements on their websites. The commissioners get paid by the publishers to provide them with advertisements to be displayed on their web pages. When a web user clicks on an advertisement at the publisher's web page he is redirected to the commissioner's server which logs the click for accounting purpose and then further redirects him to the advertisers' web page. Since the publishers get paid in proportion to the number of times they display the advertisers' content, they usually expect the commissioner to provide them with advertisements that are more likely to be clicked. This requires the commissioner to monitor a huge amount of internet traffic to discover the prevalent trends and identify those ads that are more likely to be viewed/clicked.

A commissioner on an average has 120 million unique monthly users, about 50,000 publisher sites and 30,000 advertiser campaigns, each having numerous advertisements [1]. He receives about 70 million records an hour, and has to perform a query roughly every 50 microseconds to identify information such as the top-k advertisements or the advertisements that are more frequently clicked ! This sheer amount of data streaming in per second has necessitated the need for developing systems that can handle and manage these data streams.

There exist several other examples why design of Data-Stream Management Systems

(DSMS) has been gaining popularity. For example, consider a router that is a part of the Internet and is one level ahead of popular web server i.e. all the HTTP requests are routed to the server through this router. Monitoring the network traffic at this router can possibly help in an early detection of Distributed Denial of Service (DDOS) attacks. Let us assume we have a DSMS that is continuously monitoring the following information from all the packets arriving at this router - source IP address, destination IP address, type of request. Since packets in a network will be arriving continuously, the data which we are monitoring is essentially a data-stream, S (1.1 provides a formal definition of a data stream). If the DSMS is equipped with a querying facility on the data stream, ideally, the following query written in the traditional SQL format will help in early detection of a DDOS attack:

```
Q1:   SELECT *
      FROM (SELECT src_ipaddr, count (*) as num_req
            FROM S
            GROUP BY src_ipaddr) S
      WHERE S.num_req > 100000
```

The above query helps in identifying all the source IP addresses that have been generating more than 100,000 HTTP requests. Such IP addresses with an abnormally high amount of requests in an unit interval can be potential hosts that are being used to perform a DDOS attack on the server. Though the example presented is very simplistic in nature, [2] provide a novel method based on this concept to identify a DDOS attack using sub-linear time and memory. This example demonstrates the need for systems that can handle huge amounts of data per second and also process them very quickly to provide the required information.

Though traditional database systems might be a possible solution for handling data streams, certain inherent qualities of streams such as their unknown size and frequent updates make DBMS unsuitable for managing them. We now present an example to help understand the relation between traditional queries on data bases and queries on data-streams. Consider a set B that contains IP addresses that are blacklisted. Only a certain number of requests are to be allowed from these IP addresses. If more than a specified number of requests are received, an alert or a flag has to be raised. If B and the data stream S were standard tables in a RDBMS, an SQL query of the following type could be used to obtain the number of requests from each of the blacklisted IP addresses -

```
Q2:   SELECT *
      FROM (SELECT S.src_ipaddr, count (*) num_reqs
            FROM B, S
            WHERE B.ipaddr=S.src_ipaddr
            GROUP BY S.src_ipaddr) R
      WHERE R.num_reqs > 50000
```

In a typical DBMS, the query gets processed assuming that the size of the data is known prior to computation (e.g. the GROUP BY clause). Certain query optimizations are also performed based on meta data that is available on the input tables. A DBMS has random access to the entire data and employs a fixed query plan to answer the queries. Moreover, the result set depends on the current state of the DBMS and doesn't change once it is returned to the user.

In contrast, a DSMS has to process the data in one sequential pass using limited memory. Often the meta data available in DBMS is usually absent here. Moreover, the result to query such as the above is usually a stream in itself. As soon as a new blacklisted IP address requests more than the allowed value, it has to be sent to the user immediately. As with the property of the stream, the nature of the query in DSMS is now continuous; the query continuously runs on the stream and all tuples that satisfy the query must be passed along to the user as and when they arrive. The queries registered on the streams run for long period of time owing to their continuous nature and hence multi query optimization takes an important part in the design of DSMS. Also since the voluminous amount of data cannot be stored in its entirety, the results returned by DSMS are usually approximate and the system must be capable of returning the results within user mentioned approximation levels.

Having seen the need for a stream management system, we now present a formal definition and properties of data streams that we will be using through the rest of the thesis.

1.1 Terminology

A **data stream** is often visualized as an unbounded sequence of tuples σ $(a_1, a_2, ..., a_n, ...)$, that are indexed on the basis of time of arrival at the receiver or in some cases their time of generation at the source. A data stream describes an underlying signal $\mathbf{A}$, i.e. a one dimensional function $\mathbf{A} : [1...N] \longrightarrow R$ representing a mapping from a discrete domain to the reals. Any data stream is characterized by the following properties:

- elements of a data stream arrive for processing continuously over a period or time rather than being available apriori.
- there is no limit on the number of tuples in a data stream; the length of the data stream is unbounded or unknown.
- the arrival rate or the order in which the tuples arrive is not constant.
- the tuples in a data stream can be multi-dimensional.

Based on the method of updates, a stream can represent a signal in any of the four following ways [3, 4, 5]:

- In an *aggregate model*, each element of the stream contains a range value for a particular value in the domain of the signal S. For example, the total CPU usage in a five minute interval of each CPU in a distributed computing environment.
- In a *cash-register model*, each element of the stream contains a non-negative value for a particular domain value. The domain value represented by the tuple is incremented by the value contained in the tuple. For example, visualize a_i as (j,V_i),j $\in [1...N]$, where upon seeing any a_i, an update $A[j] = A[j] + V_i$, is performed. Here, $V_i \geq 0$.
- The *turnstile model*, is a generalization of the cash-register model and relaxes the assumption that $V_i \geq 0$.
- In a *reset model*, each element in the stream replaces the earlier value of a particular domain value with the new value. On arrival of a tuple a_i (j,V_i),j $\in [1...N]$, the update A[j] $\longleftarrow V_i$, is performed.

1.2 Challenges

Given the definition of data streams above, any Data Stream Management System should be capable of handling the following challenges [6]:

- High Speed Nature of the Data Streams - An intrinsic characteristic of data streams are their high speeds. Updates on arrival of new elements must be fast enough to minimize the number of tuples lost during the update time. The high arrival rates also force the DSMS to process the data in a sequential one pass manner.
- Unbounded Memory Requirements - Since the size of the data in a stream in not known, the DSMS must be capable of handling data of potentially infinite size. The DSMS must employ techniques such as load shedding, sampling, aggregation, synopsis creation and other methods to address this challenge.
- Accuracy and Efficiency - The DSMS must be able to provide accurate results with low time and space complexities. Approximation algorithms developed for this purpose must be employed by the DSMS to address this challenge.

1.3 Organization

In the remaining part of the thesis we present the work done by us in designing a DSMS capable of handling the above mentioned challenges. Chapter 2 explains in detail the specific streaming algorithms that were employed by us to handle the challenge of accuracy and efficiency. Chapter 2 also presents the existing work in literature on the design of DSMS.

Chapter 3 presents the architecture we propose for a generic DSMS. This chapter also outlines the design goals that any data stream management should implement. Chapter 4 discusses the approach employed by us in integrating the mapping a query to a corresponding streaming algorithm and also the process of integrating any new algorithm into the system. Chapter 5 explains in detail the implementation of our system. The chapter presents the various components of the system such as the parser, input, configuration, algorithm, stream, and other components. The chapter also explains the procedure to plug in new features such a new stream source or operator into the system.

Chapter 6 discusses the performance of our system in comparison to a traditional DBMS. We present performance results comparing the time taken to answer a query, time taken to process a new element in the stream, the space used to answer the query and the accuracy of the returned result. Finally we conclude the report in chapter 7, by presenting certain open challenges and features that are yet to be addressed and implemented by the system.

Chapter 2

Literature Survey

In this chapter we present various streaming algorithms that exist in the literature for handling data streams. We also present a brief summary of the systems that currently exist for managing these high speed data streams.

2.1 Streaming Algorithms

Given the properties of data streams such as its high speed, fluctuating data rates and infinite size, it seems evident that we need algorithms which process queries on these data streams fast taking as little memory as possible. We expect the algorithms for computing functions on the data stream to have the following characteristics (*desiderata*) [5]-

- space used by the algorithm should be poly-logarithmic in the number of unique elements in the stream
- processing an update should be fast and simple
- answering queries must be fast and within user acceptable accuracy

Since the entire stream of data cannot be stored, a summary of the entire or a selected subset of the stream is stored and used for the purpose of analysis. Because of this compression in the input size, no algorithm that computes functions on the input data is precise. The algorithms are only approximate, and accuracy of the result is provided in terms of user specified parameters ε and δ; the result is accurate with an error ε with a probability $1 - \delta$.

The following are few of the techniques used to summarize a data stream [6, 7]-

- *Sampling* - Each data item is selected for processing based on a probabilistic choice. Error bounds are provided in terms of sampling rate. However, this technique has the disadvantage of missing out anomalies in the stream and also doesn't address the problem of fluctuating data speeds.

- *Load Shedding* - It refers to the process of dropping a sequence of data items or a fraction of the stream during periods of overload. Traditionally, load shedding has been used in querying data streams for optimization purpose. Load shedding is introduced in the query plan to minimize the drop in accuracy. The disadvantages of this technique are similar to those of sampling.

- *Sketching* - It is the process of randomly projecting a subset of features of the data stream. Elements of the stream are processed in chunks and partitioned and stored as clusters. The major drawback of this technique is its accuracy.

- *Synopsis Data Structures* - Synopsis data structures store the entire summary of a data stream. A lot of work has been done to develop data structures that summarize the stream in $O(log^k N)$. Histograms, Wavelet analysis, Sketches are a few examples of these type of data structures.

Of these, given their ability to summarize the entire stream and their better accuracy levels, the synopsis data structures are one of the widely used techniques to store a summary of the data streams. The *Count-Min (CM)* sketches [8] are one such synopsis data structures to store the stream. In their paper [8], have shown that the performance of CM sketches are much better than the other synopsis data structures. As mentioned above, since only a summary of the data stream is stored, algorithms that use CM sketches also return the query result in a (ε, δ) approximation, i.e. The answer is within an error range of ε with a probability of $1 - \delta$. We now describe what a CM sketch is and then proceed to explain in detail some of the queries on streams that are answered using CM sketches.

2.1.1 Count-Min Sketch

Count-Min sketch or the CM sketch [8] is named after two basic operations used to answer point queries (discussed in the section below) - counting first and computing the minimum next.

A *CM sketch* with parameters (ε, δ) is represented by a two dimensional array, with w rows and d columns; *count*[1,1],...,*count*[w,d]. Given the parameters ε and δ, set $w = \lceil \frac{e}{\varepsilon} \rceil$ and $d = \lceil ln(\frac{1}{\delta}) \rceil$. Each entry in the array *count* is initialized to 0. Additionally we chose d hash functions,

$$h_1 ... h_d : \{1...n\} \longrightarrow \{1...w\}$$

uniformly at random from a family of universal hash functions. (n represents the number of unique elements in the data stream)

When an update (i_t, c_t) arrives at time t, meaning that the current value of item i_t is updated by a value c_t, a particular cell in every column is incremented by c_t and the cell to be updated in a column is dictated by the hash function h_j 2.1. Formally, we set $\forall 1 \leq j \leq d$,

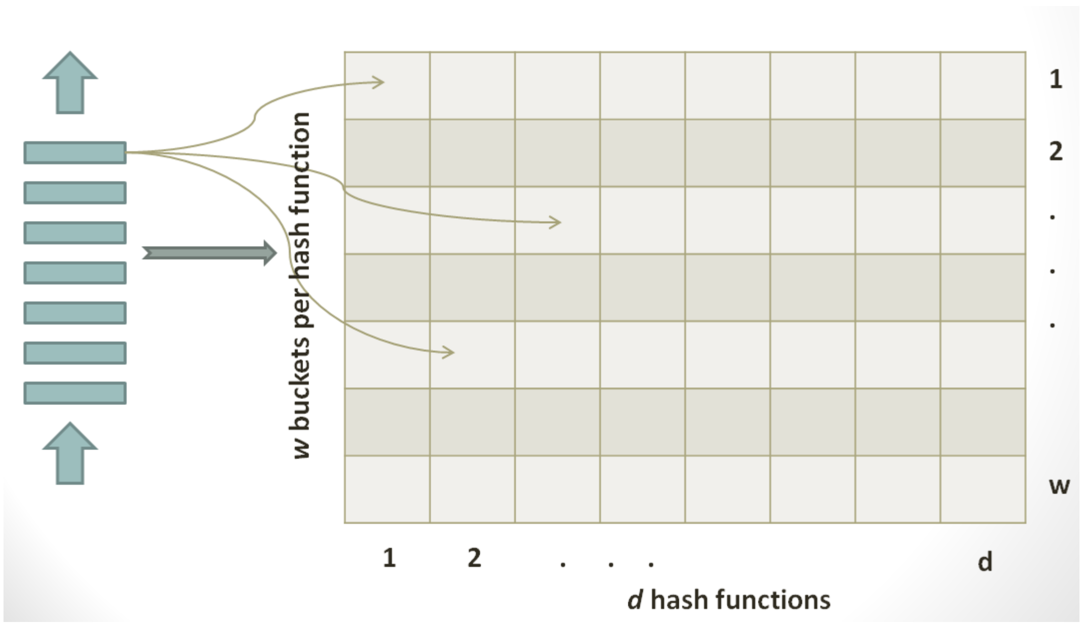


Figure 2.1: Count-Min Sketch

$$count[h_j(i_t), j] \longleftarrow [h_j(i_t), j] + c_t$$

The space used by *CM sketch* is $w \times d$ words and d hash functions each of which can be stored using 2 words. We now present certain queries on streams along with their error bounds that can be answered using the Count-Min sketch.

2.1.2 Point Query

Let the input stream be represented by

$$\sigma = a_1, a_2, ..., a_N, \text{ where each } a_i \epsilon \{1, 2, ...n\} \text{ X R .}$$

B represents a vector of length n, with all elements initialized to 0. When a new element of the stream $a_{i_t}(i_t, c_t)$ is encountered at time t, the value B(i_t) is incremented by c_t. Point query seeks to find the value of B(i), at a given instant of time for an input value i.

This can be achieved in constant time using memory $\Theta(n)$. However, in case of data streams, n can be as large as 10^9. Since this cannot be stored stored in main memory we go for storing the stream using a CM sketch. the trade off is the accuracy of the result. While 100% accuracy is achieved in the first case, since CM sketches use reduced memory, the result returned by it is an (ε, δ) approximation.

Let us assume that a CM sketch *count* is used to store the stream, as described in 2.1. Now, the result to the point query is an estimation $\widehat{B(i)}$ given by,

$$\widehat{B(i)} = min_{1 \leq j \leq d}(count[h_j(i), j])$$

Analyzing the accuracy, in case of a cash register model it can be shown that, for given values of ε and δ,

$$Pr(\widehat{B(i)} - B(i) > \varepsilon\|B\|_1) < \delta$$

i.e., with $1-\delta$ probability, the estimated value of the result lies between the true value and ε times the L_1 norm[8]. For any hash function h_j j ϵ 1,2...d,

$$count[h_j(i), j] = \text{B(i)} + \Delta_j$$

where Δ_j represents the error in the actual value due to collisions in the j_th and is non negative in case of cash-register model. This error, Δ_j is a random variable and, its expected value is,

$$\begin{aligned} E(\Delta_j) &= \text{Prob of collision} \times \text{sum of values colliding with } i \\ &= \sum_{k=1...n, k\neq i} Pr(h_j(i) = h_j(k)) \times B(k) \\ &\leq \frac{1}{w} \times \sum_{k=1}^{n} B(k) \\ &= \frac{1}{w}\|B\|_1 \end{aligned}$$

By Markov's inequality, we have,

$$Pr(\Delta_j > \varepsilon\|B\|_1) \leq \frac{1}{w\varepsilon} \leq \frac{1}{e}$$

That is, probability that single hash function returns a value greater than $\varepsilon\|B\|_1$ is at most $\frac{1}{e}$. Since the algorithm returns the minimum error value as the result, for the overall estimate returned to be a bad result, each of the individual values returned by a particular h_j must be bad. Therefore,

$$Pr(\widehat{B(i)} - B(i) > \varepsilon\|B\|_1) = \Pi_{j=1}^{d} Pr(\Delta_j > \varepsilon\|B\|_1) \leq \frac{1}{e^d}$$

From the definition of d,

$$Pr(\widehat{B(i)} - B(i) > \varepsilon\|B\|_1) \leq \delta$$

Therefore, for any given value of ε and δ, if $w = \lceil \frac{e}{\varepsilon} \rceil$ and $d = \lceil ln(\frac{1}{\delta}) \rceil$, we can guarantee that the estimate returned as a result for the Point Query satisfies

$$B(i) \leq \widehat{B(i)} \leq B(i) + \varepsilon\|B\|_1$$

with a probability $1-\delta$.

The time take to return the query result is O(d), i.e. O($ln(\frac{1}{\delta})$), because we have to minimum amongst d elements. The overall space complexity is $O(\frac{1}{\varepsilon} ln\frac{1}{\delta})$. Time taken to update the sketch upon the arrival of new element in the stream is again O($ln(\frac{1}{\delta})$).

2.1.3 Range Query

Let the input stream be σ and B be the zero initialized vector as defined in 2.1.2. Range Query is defined as follows -
Given l,r, compute $B[l,r] = \sum_{i=l}^{r} B(i)$, where $l < r$, and l,r ϵ {1,2,..n}.

While a simple way of approach would be to issue r-l point queries and return the summation of the values as the result, such an algorithm has a maximum error of O($n\varepsilon\|B\|_1$). Ideally we would want the error bounds to be the same, if not decrease.

In [], a solution using *dyadic* ranges has been proposed which answers the range query with a maximum error of $\varepsilon\|B\|_1$. Using sketches for maintaining dyadic ranges in the powers of 2, it has been shown that the estimated result, $\widehat{B[l,r]}$ using this procedure satisfies,

$$B[l,r] \leq \widehat{B[l,r]} \leq B[l,r] + \varepsilon\|B\|_1$$

with a probability of 1-δ, where ε and δ are user specified values.

The total time taken to answer the query is $O(log(n)ln(\frac{1}{\delta}))$, while the space complexity is $O(log^2(n)ln(\frac{1}{\delta}))$.

2.1.4 Heavy Hitters

Consider again the input stream σ and the zero initialized vector B as described in 2.1.2. A heavy hitters query is defined as follows -

For any given parameter ϕ ϵ (0, 1], return indices I $\subseteq$ {1,2,..n}, such that, for each i ϵ I,

$$\text{B(i)} \geq \phi\ \|B\|_1$$

These elements that satisfy the above condition are called *heavy-hitters.*
While a trivial solution would be to issue point queries till we get $\lceil\frac{1}{\phi}\rceil$ heavy hitters, such an approach would take O(n) time to find all the true heavy hitters. A better approach that is mentioned in [], involves summarizing the stream in dyadic ranges as mentioned in 2.1.3. A sketch is maintained for each dyadic range in powers of 2, resulting in $\log_2(n)$ CM sketches being used. Based on a simultaneous binary search beginning from top level going downwards, the algorithm outputs set of elements I such that for each i ϵ I,

$$B(i) \geq (\phi + \varepsilon)\|B\|_1$$

with a probability $1 - \delta$, for given ε, δ. The algorithm ensures that amongst all the values that it returns as the heavy hitters, none of the true heavy hitters are missed. The space complexity is O($log(n)\ \frac{1}{\varepsilon}\ log(\frac{log(n)}{\delta\phi})$) and the time to taken to obtain the result for a given query is ($log(n)\ \frac{1}{\phi}\ log(\frac{log(n)}{\delta\phi})$).

2.2 STREAM

We now present STREAM, one of the existing data stream management systems and a key motivator for our system design and implementation. STREAM [9] is *St*anford st*re*am dat*a* *m*anager, a product of the Stanford university for

> "re-investigating data management and query processing in the presence of multiple, continuous, rapid, time-varying data streams ".

STREAM is a general purpose data stream management system, that supports a large class of declarative continuous queries over continuous streams and traditional stored data sets. The STREAM prototype targets environments where streams may be rapid, stream characteristics and query loads may vary over time, and system resources may be limited. STREAM uses CQL [9], as the query language and uses a windowing operator to identify the data on which a query has too be executed. STREAM has operators such as *relation-relation, stream-relation, relation-stream* operators to answer queries. These operators can be used either with standard table of a RDBMS or a data stream or both. However, STREAM does not have the feature of user controlled approximation. The error is not under the control of the user. STREAM also uses only the primary memory and does not use the secondary or the persistent storage to store and answer the queries. In our system design, we primarily try to address these two issues and present certain other challenges that need to be addressed both by STREAM and our system.

Chapter 3

Database Stream Management System Architecture

Database stream management system (DBSMS) is motivated from the Stanford Stream Data Manager (STREAM), a system for executing continuous queries over multiple continuous data streams as described in section 2.2. DBSMS tries to integrate a data stream management system (DSMS) and a traditional database management system (DBMS). The STREAM system supports declarative query language, and it copes with high data rates and query workloads by providing approximate answers when resources are limited. DBSMS system will extend STREAM to provide features like user controlled approximation, persistence support, integrating user defined approximate algorithms into STREAM, mapping given query into approximation operators and multi query optimization.

3.1 DBSMS architecture

DBSMS system architecture is similar to the STREAM architecture. Primary extension to provide a relation database integration to the streaming database. DBSMS will separate processing system into two units - *Stream Manager* and *Relation Manager*. STREAM can be used as data stream management system (DSMS) and any relational database can be used as data base management system (DBMS). DBSMS will use these two systems to process both streaming and static data. [Refer: Fig 3.1].

The DBSMS system supports declarative query language [refer 3.3] using extend version of Continuous Query Language. In this architecture user can register a query similar to the STREAM system and can also register a stream. User can plug in any approximation algorithm as an operator to answer certain kind of queries with user desired accuracy.

Queries will be submitted to the query processor. Query processor analyses the query and identifies it's type. Query processor is divided into *query parser* and *query analyzer*. Query parser parses the input query and validates its syntax. After validating the syntax,

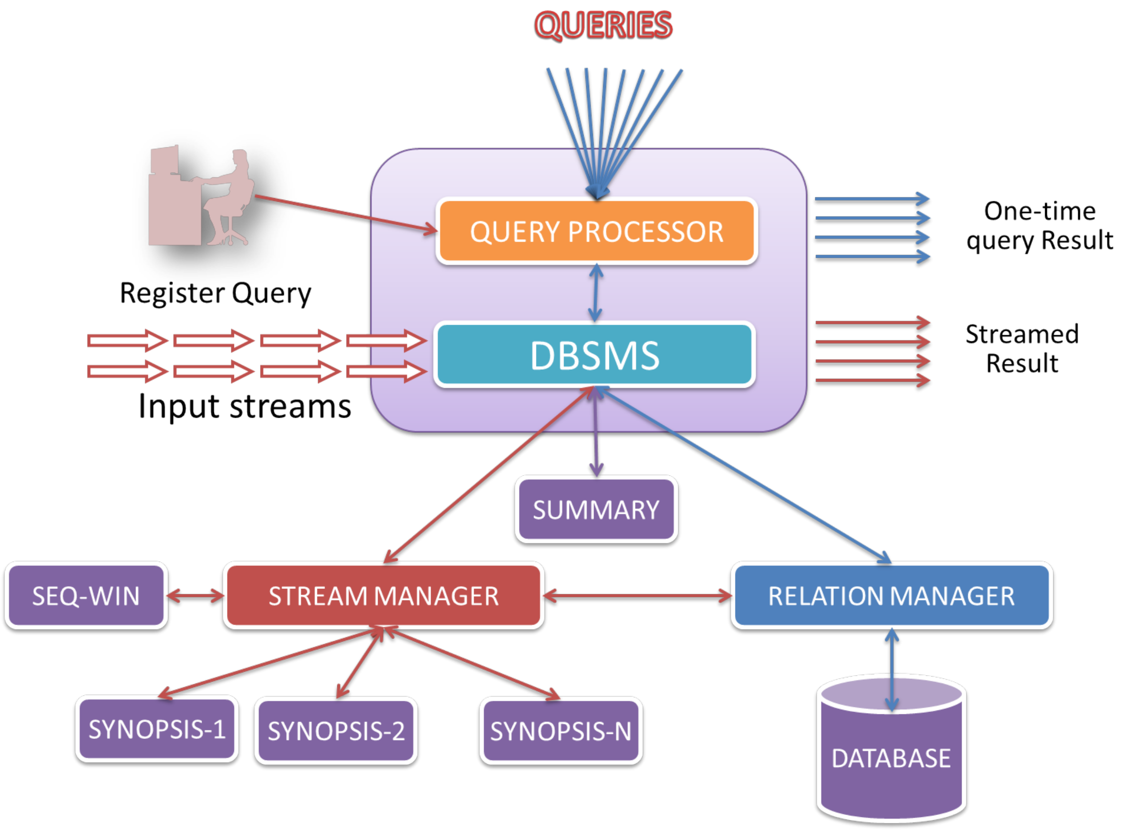


Figure 3.1: Database Stream Management System Architecture

the query will be passed to the query analyzer for semantic checking. Finally, after validating it syntactically and semantically, it will be passed to the DBSMS system which decides the type of query - whether it is CQL, SQL or User Defined Algorithm query. DBSMS system generates the execution plan of query according to the type of query. For each query, result can be a one time result or a continuous result i.e. result is also a stream. Query type and their result types are discussed in the following section.

In this system user has the option to specify the approximation algorithms and it's equivalent query template. By using this template we can generate a query plan and can replace similar query plans using the corresponding approximation operator. DBSMS also maintains summary about streams and queries to answer queries on stream statistics such as minimum key, L_1 norm etc. efficiently. Execution plan of DBSMS can use the help of both DBMS and DSMS. Execution plan and query mapping from one type another are discussed in the further sections.

3.2 Types of queries

Queries in our system are classified into five types.

3.2.1 One-Time query

One time query is the query which is submitted any time after the stream on which it is registered has started. It is executed on the current state of DBSMS and the result set is static and not a stream. This is further divided into two types depending on whether the query is executing on relation or stream.

3.2.1.1 One time query on relation (Type - 1)

This query is equivalent to querying the relational database. DBMS will be used to give an exact answer to this query.

3.2.1.2 One time query on stream (Type - 2)

This query is equivalent to querying the streaming database. DSMS will be used to give approximate answer to this query.

3.2.2 Continuous query

Continuous query is the query which will be running continuously. This is divided into three queries depending on time of submission of the query. This type of query will generate streaming result. Result is updated as and when a tuple satisfying the query appears in the stream.

3.2.2.1 Preregistered query (Type - 3)

This type of query will be submitted before the stream is started. The DBSMS will prepare the query plan at the time of starting the stream itself and will maintain all data structures needed to answer the query efficiently.

3.2.2.2 Ad-hoc query using previous knowledge (Type - 4)

This type of query is submitted after the stream on which it is registered has started. The DBSMS will prepare the query plan at that time of registration. System uses existing data structures and previous knowledge about streaming data to answer this query efficiently.

3.2.2.3 Ad-hoc query using current knowledge (Type - 5)

This type of query is submitted after stream has started. DBSMS will prepare the query plan at that point of time. System will create new structures for this query and use the current knowledge and data seen from the point of registration.

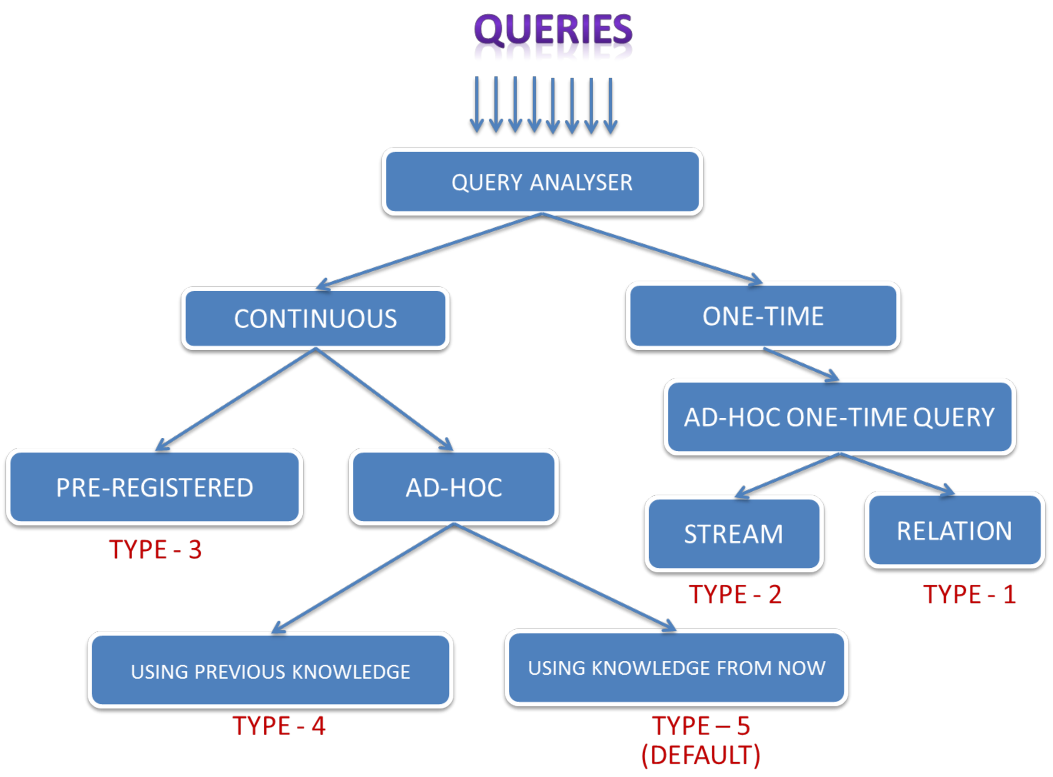


Figure 3.2: Types of queries

3.3 Query language

Continuous Query Language (CQL) is extended version of Structured Query Language(SQL). In CQL queries are continuous as opposed to the one time queries supported by standard database management systems (DBMS). For syntax and semantics of CQL refer [1]. The DBSMS system supports declarative query language using extend version of Continuous Query Language (CQL) [1], So we call our language EX-CQL (pronounced as ex-sequel), for Extended Continuous Query Language. In EX-CQL queries are classified into different types according to their properties. Syntactically EX-CQL will extend CQL by allowing user to specify algorithm and it's arguments to be processed over stream. By using approximation algorithms, system can execute approximation queries efficiently.

3.3.1 Registering a stream

Register a stream using *register* keyword followed by *stream*, stream name and stream arguments. Stream arguments are used to create stream source. Stream source generates stream of elements.

`register stream` streamname (arguments)

3.3.2 Registering a query

Register a query using *register* keyword followed by *query*, query name, query type and query arguments. Two types of query types are supported - CQL and UDA. CQL query type will be used to register a CQL query.

`register query` queryname `querytype CQL` (CQL query)

SQL query type will be used to register a SQL query.

`register query` queryname `querytype SQL` (SQL query)

UDA query type will be be used to register user defined query. Query arguments are used to create operator queue thread object. Operator queue thread will be used to get the stream elements from the stream source. User can define their own algorithms to be executed. User defined algorithm can be registered as follows.

`register query` queryname `querytype` UDA (USER_DEFINED_ALGORITHM [arguments])

Query can be registered in any one of three types. Syntax to all types are as follows -

1. Preregistered query (Type - 3) Syntax :

   `pre_register query` queryname `querytype (SQL|CQL|UDA)` (query)

2. Ad-hoc query using previously stored data (Type - 4) Syntax :

   `register_with_knowledge query` queryname `querytype (SQL|CQL|UDA)`(query)

3. Ad-hoc query using current data (Type - 5) Syntax :

   `register query` queryname `querytype (SQL|CQL|UDA)` (query)

3.3.3 Querying the result

User can query the system using `queryresult` followed by queryname or stream name. Querying on stream will return the properties of stream.

`queryresult queryname` queryName arguments

`queryresult streamname` streamName statistics

3.4 Query execution and conversion from one type of query to another type

Depending on resource availability and query type one query can be converted to another. Type-1 query can be directly submitted to the relational manager and get back the one time result.

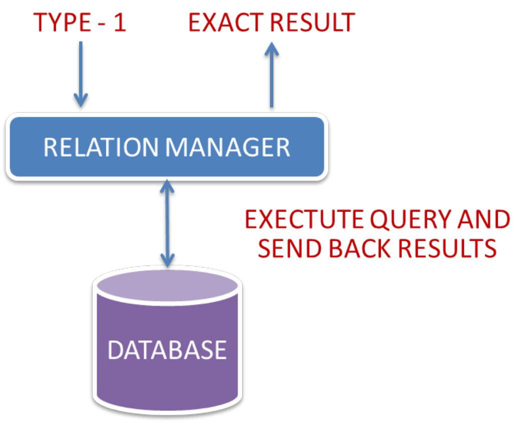


Figure 3.3: Query execution on DBMS

Except Type-1 query all remaining queries will be submitted to the stream manger. Stream manager returns onetime or streamed result based on the type of the query.
Type-2 query will be executed using existing sketch and synopsis information. This is one time query so query result is one time result .
Type-3 query will be executed using existing sketch and synopsis information and send the results on event basis. This query will be registered with the system and will be executed on element arrival.
Type-4 query will be executed using existing sketch and synopsis information and then Type-4 query will be mapped to the type-3 query.
Type-5 query will be directly mapped to the Type-3 and it will not send any result. After mapping to the Type-3 query it will be answered according to the events occurred.

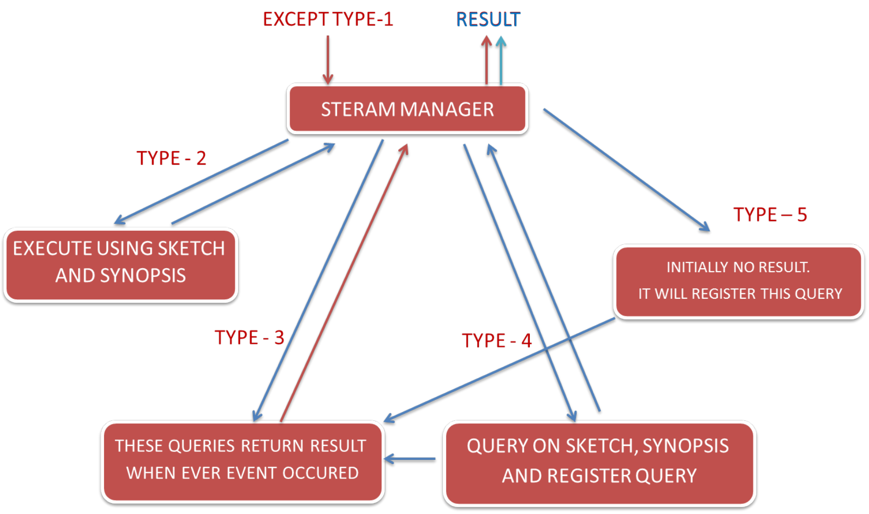

EXCEPT TYPE-1
RESULT
STERAM MANAGER
TYPE - 2
EXECUTE USING SKETCH AND SYNOPSIS
TYPE – 5
INITIALLY NO RESULT. IT WILL REGISTER THIS QUERY
TYPE - 3
TYPE - 4
THESE QUERIES RETURN RESULT WHEN EVER EVENT OCCURED
QUERY ON SKETCH, SYNOPSIS AND REGISTER QUERY

Figure 3.4: Query type conversion

3.5 Query processing structure

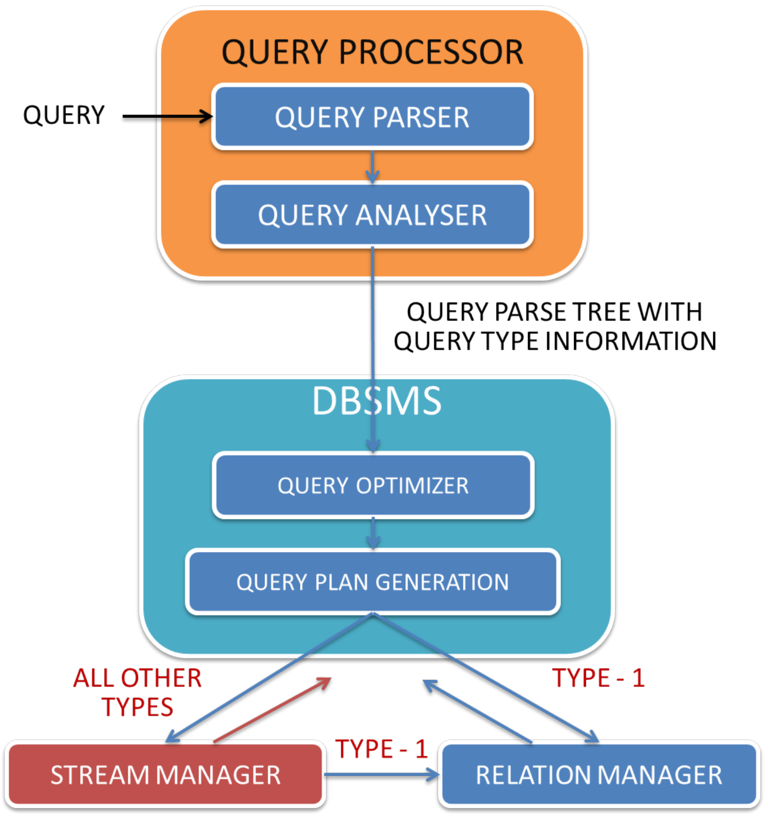


Figure 3.5: Query processing structure in DBSMS

3.6 Project goals

We now elucidate the goals that we had in mind during the design phase of the system.

- Query mapping into approximation operators - System should be capable of automatically mapping a given query to the corresponding approximation operators that have to employed to answer the query.

- Integrating streaming algorithms into the streaming database - Need an easy mechanism to integrate new approximation algorithm operators into the system without modifying the system much.

- User controlled approximation - Since we store only a summary of the sketch, the result returned by the system is an approximate result and the user must have control on the accuracy of the result returned to him. Depending on this the system must employ storage structures to ensure that this desired accuracy is achieved.

- Persistent support - Currently, all the existing DSMS are memory based. We explore the possibility of using persistent storage also to answer a query. For example, initially when a stream starts, the storage is in the main memory and after a fixed pre-determined interval of time or based on the percentage of saturation of the sketch, we move this sketch to a disk. All the data that is moved into the disk must also be used to answer the query registered on this stream.

- Error propagation - Consider two approximate queries on a stream. If there is another query that makes use of the results from these two queries how does the overall error specified by the user for the top-level query, propagate to these two below it ? If there are multiple such levels and multiple approximate queries, the system must be able to calculate the error propagation to the lowest level and must be able to return the overall result within the user specified error.

- Query feasibility - Considering both the error propagation and the fact that we store only a part of the stream (stream is stored as a key,value pair - all other attributes are discarded) can a query be actually answered by the system ? If there is multi-dimension tupled stream, and we store only two attributes of the multiple dimensions, the system can definitely not answer any query on the discarded attributes and such a query must be returned as non-feasible to the user.

- Multi query optimization (Query plan sharing - Sub query matching) - As compared to traditional query optimization which looks at only one query while query while generating the best plan for its execution, multi query optimization comes into play for streams. The system has to look at all the queries registered on a stream and check whether any of them can be used as sub query to obtain the result for the actual query.

Chapter 4

Query mapping and Integrating algorithms into Query Plan

4.1 Query mapping into approximation operators

Streaming queries are divided into three types according to the properties of the query and results. They are streaming-statistics queries, streaming-aggregate queries and streaming-data queries. Assume stream A is collection of key and value pairs. Let A[1], A[2] ... A[m] be the elements in the stream. Let the stream be stored as a key and value pair.

4.1.1 Streaming statistics queries

Streaming statistics queries are queries related to statistics of stream like multiplicity,cardinality, sequence length, self-join size, join size and $K_t h$- Norm etc.

4.1.1.1 Multiplicity

Multiplicity of X is, number of times that the value X occurs in the sequence A. Following queries (all five types of queries) are supported by DBSMS.

```
Type-1:
    select key, count(*) from relation group by key;
Type-2:
    istream(select key, count(*) from stream[window] group by key);
Type-3:
    pre_register relationToStreamOperator(
        select key, count(*) from stream[window] group by key);
Type-4:
    register_with_knowledge relationToStreamOperator (
        select key, count(*) from stream[window] group by key);
```

```
Type-5:
    register relationToStreamOperator(
        select key, count(*) from stream[window] group by key);
```

Equivalent approximation algorithm to multiplicity is approximate Point query on a stream with value 1 for every key.

4.1.1.2 Cardinality

Cardinality is number of distinct elements occurred in A.

`Equivalent query : select count(distinct key) from relation;`

Equivalent approximate algorithm to maintain cardinality is to maintain approximate L_0 norm [8].

4.1.1.3 Sequence length

Sequence length is number of elements in stream A.

`Equivalent query : select count(key) from relation;`

Equivalent exact algorithm to maintain sequence length is to maintain one variable counting number of elements arrived. Query plan sharing Multi Query optimization section

4.1.2 Streaming aggregate queries

Streaming aggregate queries are queries related to aggregate operations like minimum, maximum, sum, and average etc. on value column. For all these aggregate queries we can maintain a variable to maintain the result. These queries can be answered efficiently by this method.

4.1.3 Streaming data queries

Streaming data queries are queries related to streaming data like point query, range query, heavy hitters, top-k elements, intersection of two streams etc.

4.1.3.1 Point query

Get sum of values of a key in stream. [Refer 2.1.2]

`Equivalent query : select sum(value) from stream[window] group by key having key=?;`

Equivalent approximation algorithm to multiplicity is approximate point query using count min sketch.

4.1.3.2 Range query

Get value of stream at any point of time.

```
Equivalent query : select sum(value) from stream[window] where key<=l and key>=h;
```

4.1.3.3 Heavy hitters

The heavy hitters of a multi set of a1 (integer) values each in the range 1 . . . n, consist of those items whose multiplicity exceeds the fraction of the total cardinality, i.e., $\phi \times L_1$. There can be 0 to $\frac{1}{\phi}$ heavy hitters.

```
Equivalent query : select key, sum(value) from relation group by key
having sum(value) >= value;
```

4.2 Integrating streaming algorithms into the streaming database

A query plan is set of ordered steps to access or modify information. A query plan is also called as upside down tree of operators. Query plan consists of set of nodes(operators). Each operator position determines its order of execution. Execution starts down the left-most branch of the tree and proceeds to the right.

In this system each approximation algorithm could be an operator. An approximation algorithm will have its equivalent query so that it has equivalent query plan. In the original query plan identify the sub trees (query plans) and replace with an approximation operator. Sub query will be replaced by approximation operator only if query is feasible after replacement. [Refer Query feasibility - section 3.6]. User can plug in a query plan by specifying equivalent query and it's approximation algorithm and it's properties. After query is replaced with approximation operators for each approximation operator physical algorithm will be mapped. Then query executor will execute the query. Common query plans between queries can be shared [Refer Multi query optimization - section 3.6].

4.3 User controlled approximation

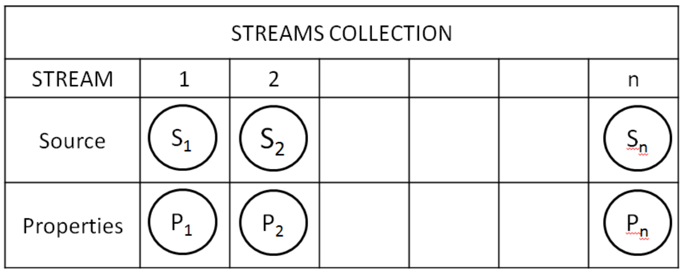
User controlled approximation is controlling the error by specifying the approximation parameters such as ε and δ. ε is error value and δ is error probability. It is algorithm's responsibility to provide tuning parameters for approximation. All integrated approximation algorithms are based on ε and δ approximation. User can specify approximation parameters at time of registration of query. Entire storage structures are created based on these parameters. These parameters affect the time and space taken to answer a query . Here there is a trade off between error value and time to answer query. User can specify approximation levels as follows `register query` queryname `querytype` UDA (USER_DEFINED_ALGORITHM [arguments])

For example to arguments for point query are ε and δ. For heavy hitters ε, δ and ϕ value. Currently, user can control the error of user defined algorithms. To generalize approximation to the entire query plan , error propagation has to be analyzed [Refer Error propagation in chapter 7].

4.4 Execution structure of user defined algorithm

4.4.1 Streams collection

Streams Collection class consists of information about all registered streams. When a stream is registered *Stream Source* object will be created. *Streams Collection* maintains mapping between properties of stream and stream source. This class helps us in integrating all streams and related information into a single unit.

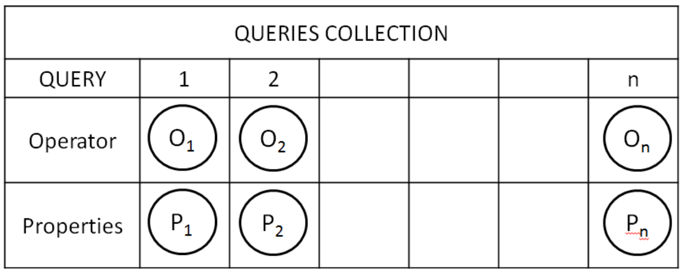


STREAMS COLLECTION						
STREAM	1	2				n
Source	S_1	S_2				S_n
Properties	P_1	P_2				P_n

Figure 4.1: Streams collection class

4.4.2 Queries collection

Queries Collection consists of information about all registered queries. When a query is registered with a stream, an *Operator Queue Thread* and an object to the corresponding algorithm will be created. *Queries Collection* maintains mapping between properties of operator queue thread and query properties. This class helps us integrating all queries and related information into single unit.

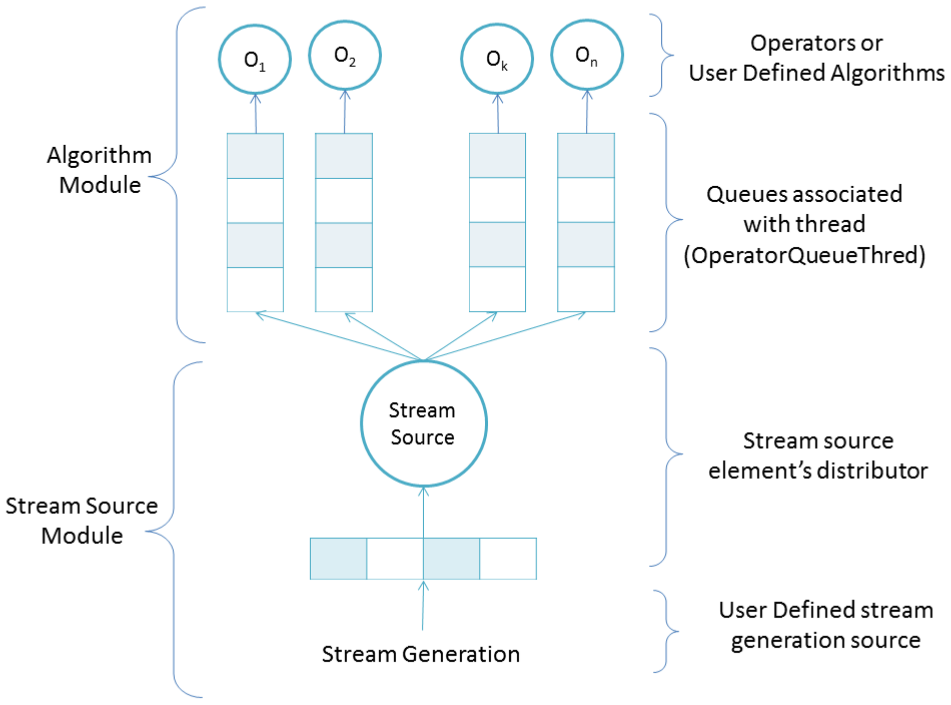


QUERIES COLLECTION						
QUERY	1	2				n
Operator	O_1	O_2				O_n
Properties	P_1	P_2				P_n

Figure 4.2: Queries collection class

4.4.3 StreamSource and OperatorQueueThread communication structure

Stream Source module is combination of stream elements generation and stream elements distribution. Stream distribution module is common for all generalized stream sources. Stream distribution module maintains collection of operator queues of operators registered with this stream. Once an element is generated that element will be inserted into operator queue of stream source. Stream distribution engine will be notified. Distribution engine will keep copy of element into all registered queues. Operator queue thread will notify element arrival information to the corresponding operator. Corresponding operator will perform operation.



O_1
O_2
O_k
O_n
Operators or User Defined Algorithms
Algorithm Module
Queues associated with thread (OperatorQueueThred)
Stream Source
Stream source element's distributor
Stream Source Module
User Defined stream generation source
Stream Generation

Figure 4.3: Stream source architecture

Chapter 5

System design

5.1 Parser component

Parser component is to parse the input query language EX-SQL. Parser is written in JAVACC. Input query can be received from input terminal (InputConsoleQueryReader) provided by the system or as from the port on which the system is listening. Once system gets the query, it will be first passed to the parser component. Parser component will parse the query and check for syntactic correctness of the query. While parsing the query, parser will set properties of the query into the *Request* object. Using this object, Input component detects kind of query and the operation to be performed.

5.2 Input component

Input Component defines the input structure of input to be sent to the DBSMS engine. Input structure means request to the system.

```
public enum RequestType {
    REGISTER, QUERY_REQUEST, SHOW, ACTION
};

REGISTER_REQUEST : Request to register an entity into the system
QUERY_RESULT : To Query the system
SHOW : Show properties and entities of system
ACTION : Request to control the system
```

Requests are classified into four types according to the functionality of the request.

5.2.1 Register request

Register request is used to register an entity into the system. Register request divided into two types of register requests - stream register request and query register request.

5.2.2 Action request

Action request is used to control the system. Action request Types are defined in ActionRequestTypesEnum.

5.2.3 QueryResult request

QueryResult request is used to query the system. QueryRequestInputTypeEnum is used to specify query request type - i.e. whether to execute the query using query name or query id or on a stream using stream name.

5.2.4 Show request

Show request is to obtain system details. ShowRequestTypeEnum consists of details which can be obtained by show request.

5.3 Configuration component

Configuration component consists of the entire system configuration. This configuration consists of properties required to connect with the stream system to answer CQL queries from the DBSMS system. This module also consists of system check point properties and other properties of the system.

5.4 Stream component

Stream component is a core component of the system. This component includes *Stream Source*, *Operator Queue Thread* etc. System was tested over Key and Value pairs. Same concept can be extended to the rows also. Basic element in the system is *Stream Element.* Stream element is combination of Key and Value.

5.4.1 Stream Element

Stream element is a collection of a key and a value pair. But it can be multidimensional also. In this system key is an integer. User is responsible to map a key to the corresponding integer.

5.4.2 Stream source

Stream source is responsible to generate and distribute stream elements to the all registered operators.

5.4.3 Operator queue thread

Operator queue thread is combination of a queue and thread. Thread will be informed when an operation was performed on the queue. By using this property derived class will be notified when an operation was performed in the parent class.

As persistency is one of most crucial property to the system, to synchronize system with the secondary storage following two classes will be used.

5.4.4 Streams collection

This class will help to store all data related to the streams.

5.4.5 Queries collection

This class will help to store all data related to the operator queue threads.

Implementation classes are provided so that user need not implement basic functionality. It is enough to provide core functions which are needed. Some of implementation classes are StreamElementImpl, StreamSourceImpl, OperatorQueueThreadImpl. These implementation classes will make user task much easier.

5.5 Algorithm component

Algorithms module is to plug in user defined algorithms into the system. Any algorithm can be easily plugged into the system by simply extending algorithm package.

5.6 Process to plug in user defined stream source

- New Stream Source class must extend StreamSourceImpl class and make an entry into enum StreamSourceTypesEnum.
- Map class to the corresponding class name in StreamSourceObject.properties. This file consists details of corresponding class name and stream source type.

```
public enum StreamSourceTypesEnum {
    FILE, DBTABLE, RANDOM
}
```

```
StreamSourceObject.properties
```

```
FILE FILE
DBTABLE DbTable
RANDOM RandomStreamGeneration

public class TestStreamSource extends StreamSourceImpl {
    @Override    public void generateStream() {    }
    @Override    public void initAlgorithmDeSerialization() {    }
}
```

5.7 Process to plug in user defined algorithm operator

- New algorithm class must extend *algorithm* class and make an entry into enum OperatorTypes.
- Map class to the corresponding class name in OpthreadObject.properties. This file consists details of corresponding class name and operator type.

```
public enum OperatorTypes {
     POINT_QUERY, RANGE_QUERY, HEAVY_HITTERS, TOP_K,
STATISTICS,DATABASE_STATISTICS, TEST_ALGORITHM
};
OpthreadObjectProperties.properties
POINT_QUERY  CountMinSketch
RANGE_QUERY  CMRangeQuery
HEAVY_HITTERS  HeavyHitters
TEST_ALGORITHM TestAlgorithm

public class TestAlgorithm extends algorithm{
    @Override    public void initAlgorithmDeSerialization() {    }
    @Override    public void operationOnArrival(StreamElement element) { }
    @Override    public QueryResult getResultSet(String[] arguments) { }
    @Override    public OperatorTypes getType() {    }
}
```

For more detailed structure of implementation refer Appendix.

Chapter 6

Performance and Results comparison

We now present some of the tests that were preformed to analyze the efficiency of our system. Tests are conducted between DBSMS and DBMS. Data stream is generated using network packet data. Network packets are captured and then simulated as stream source. Simulated stream size is 2,000,000 records of key and value pairs. In all these experiments stream generation time is also included. Each record consists of IP address and packet size. *Stream Source* object maps IP address to {1,2,..n}. Different experiments are conducted to identify the system behavior and performance is compared with a well-known database. Point query, range query and heavy hitters queries are a few algorithms selected for testing.

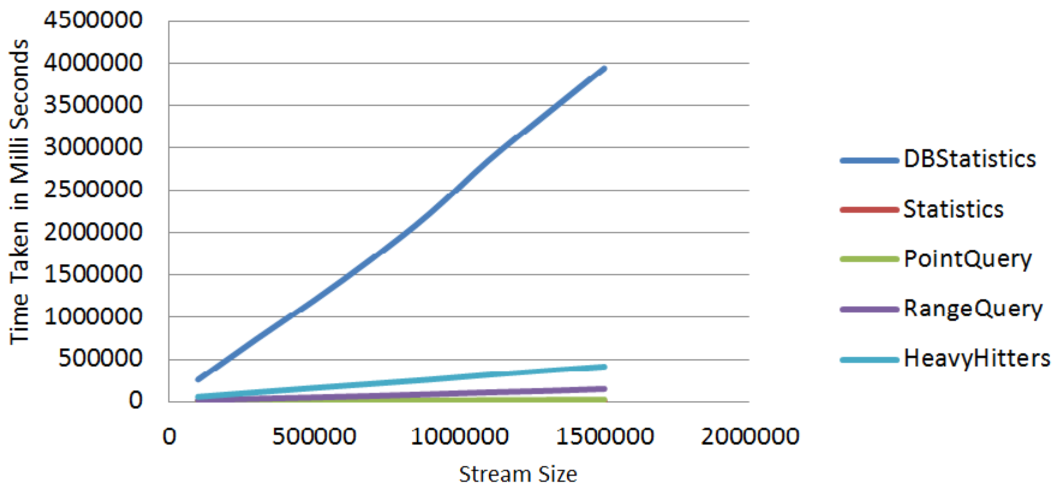


Figure 6.1: Insertion times of DBMS vs DBSMS

Insertion times into DBSMS and DBMS are shown in [Fig. 6.1]. In DBSMS insertion time is different among queries because storage structures are different for different queries.

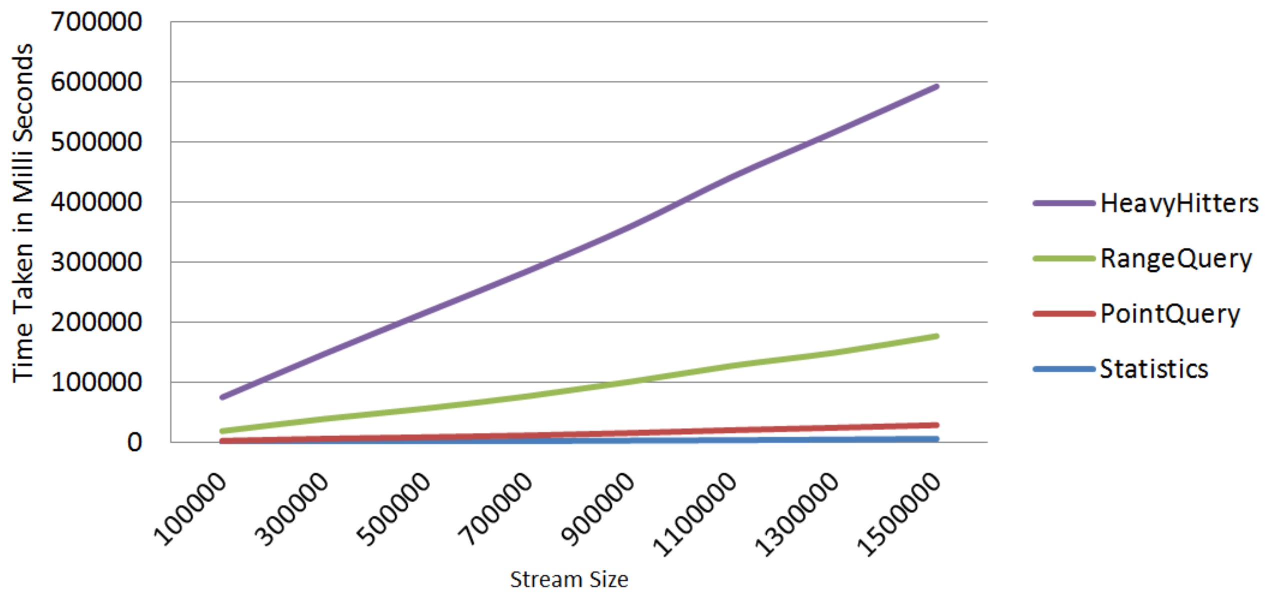


Figure 6.2: Insertion times for different streaming algorithms on DBSMS

Insertion times for different algorithms into DBSMS is shown in [Fig. 6.2]. Heavy hitters is taking more time when compared to other algorithms because number of operations to be performed during insertion into data structures for heavy hitters is high when compared to the remaining algorithms.

Execution times for different algorithms on the DBMS is shown in [Fig. 6.3].

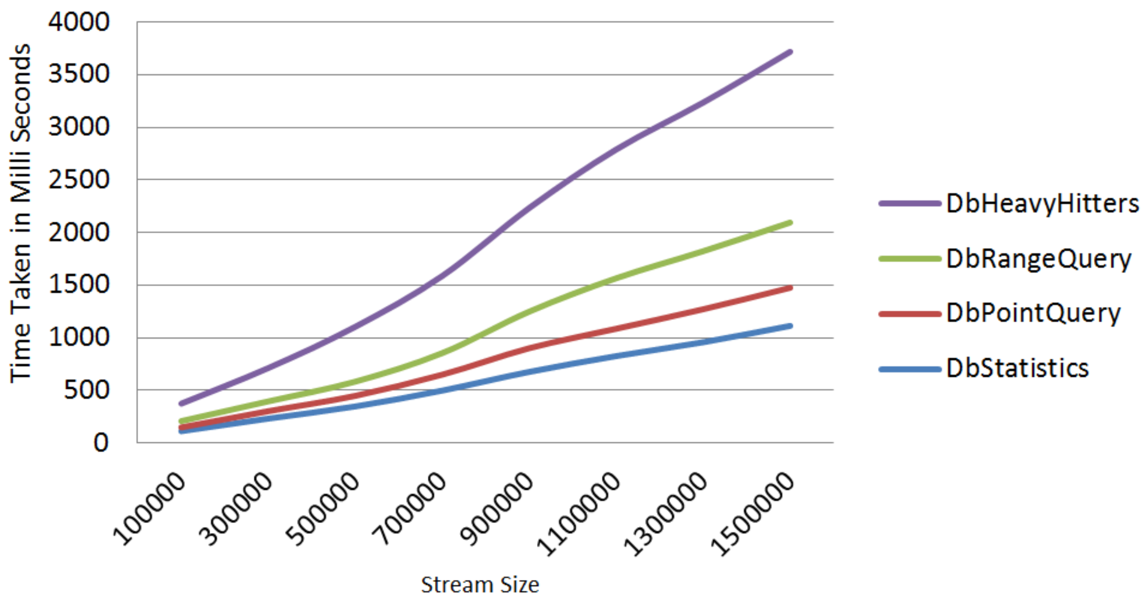


Figure 6.3: Execution times for different streaming algorithms on DBMS

Execution times for different algorithms on DBSMS is shown in [Fig.6.4]. In this almost every algorithm is taking constant time because on an average number of operations to be performed for various sizes of streams is same.

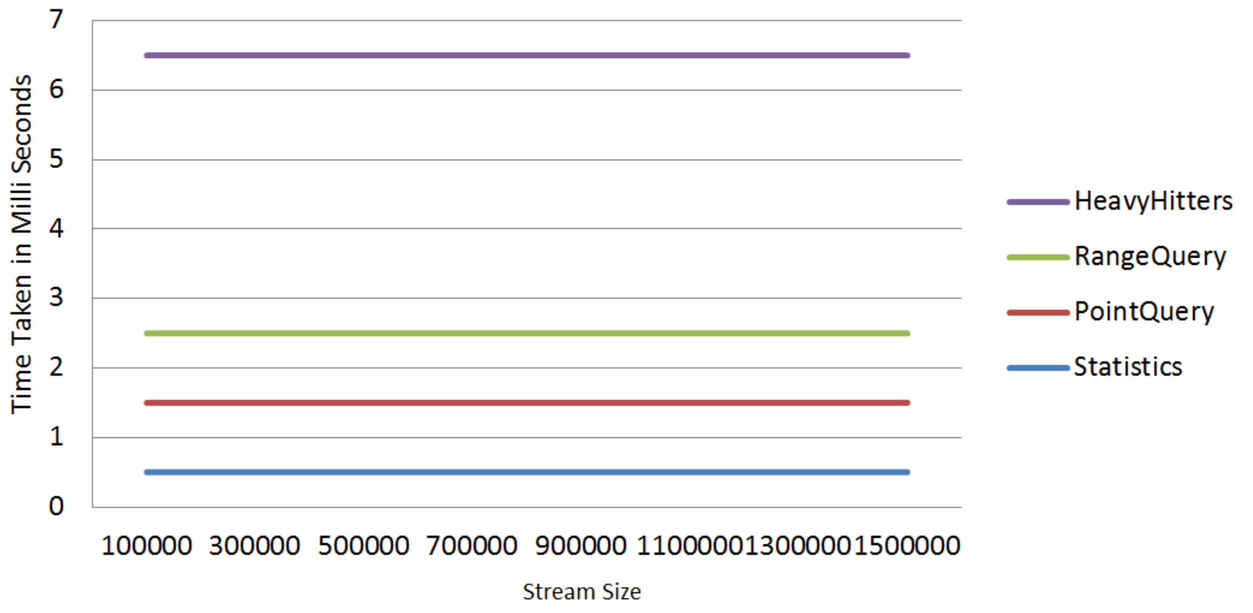


Figure 6.4: Execution times for different streaming algorithms on DBSMS

[Fig. 6.5] compares execution time of statistics query over DBMS and DBSMS. In DBSMS statistics take very less time because statistics are pre-computed unlike computing results every time in databases.

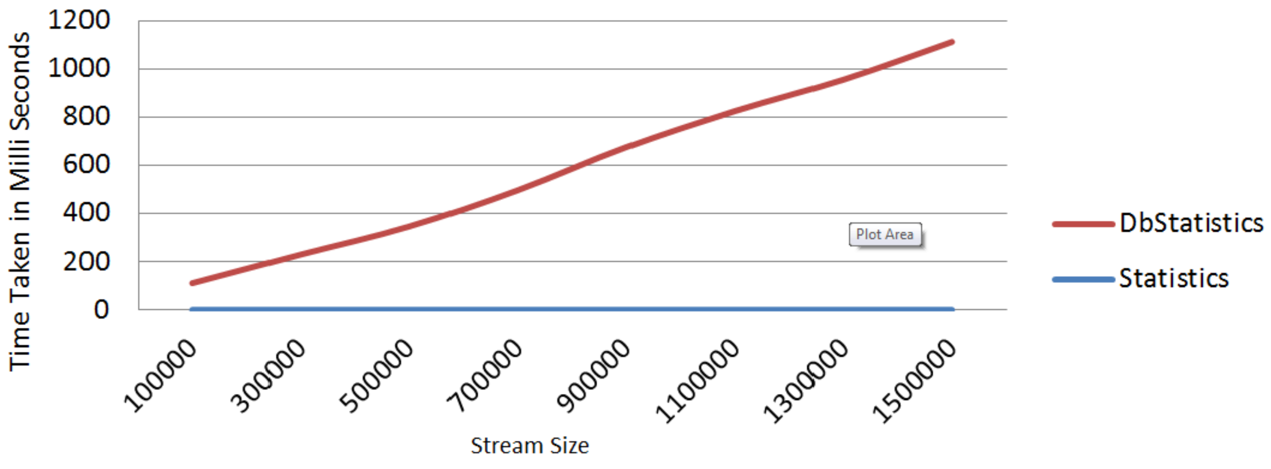


Figure 6.5: Execution time of statistics query DBSMS vs DBMS

[Fig. 6.6] compares execution time of point query over DBMS and DBSMS. In DBSMS the query takes very little time to execute because it is answered using approximation algorithms (count-min sketch) unlike in databases which treat execute it like a normal SQL query using all the different optimizations.

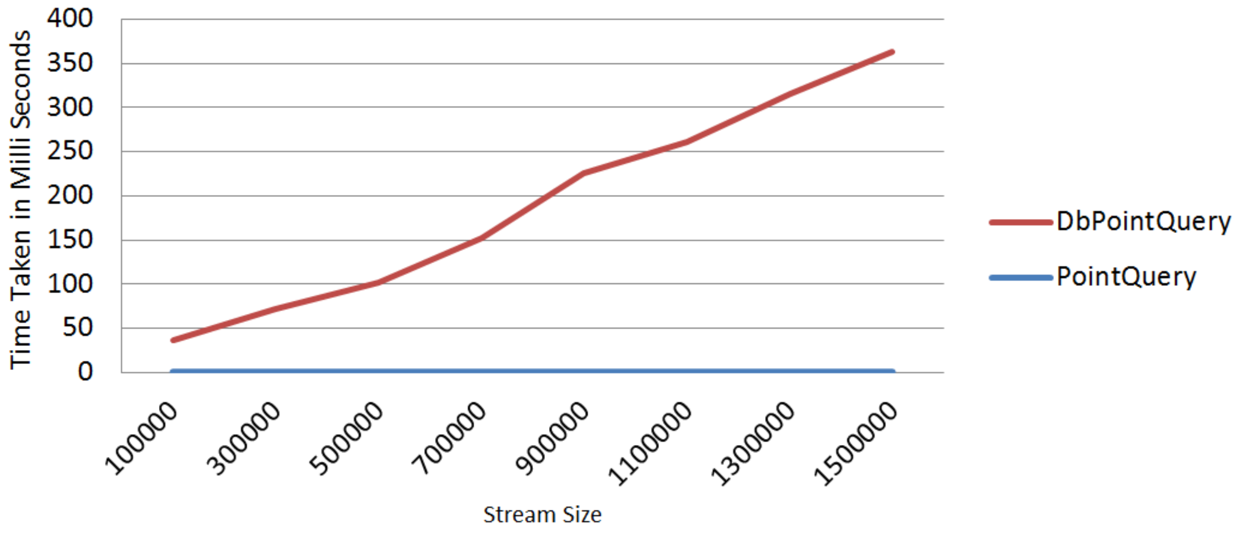


Figure 6.6: Execution time of point query DBSMS vs DBMS

[Fig. 6.7] compares execution time of range query over DBMS and DBSMS. Compared to DBMS the query takes lesser time because it is answered approximately using dyadic ranges unlike providing an exact answer in databases.

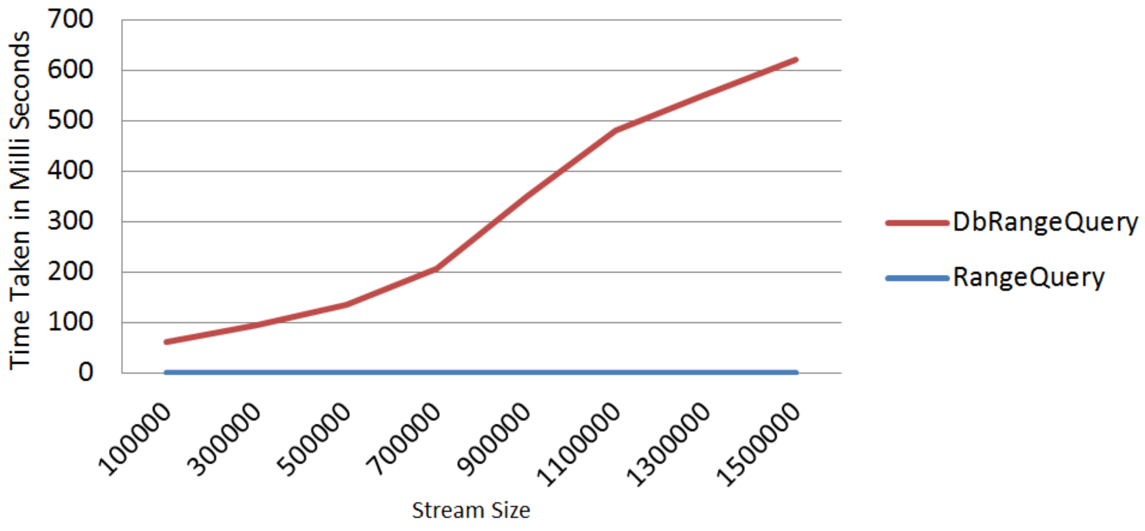


Figure 6.7: Execution time of range query DBSMS vs DBMS

[Fig. 6.8] compares execution time of heavy hitters over DBMS and DBSMS.

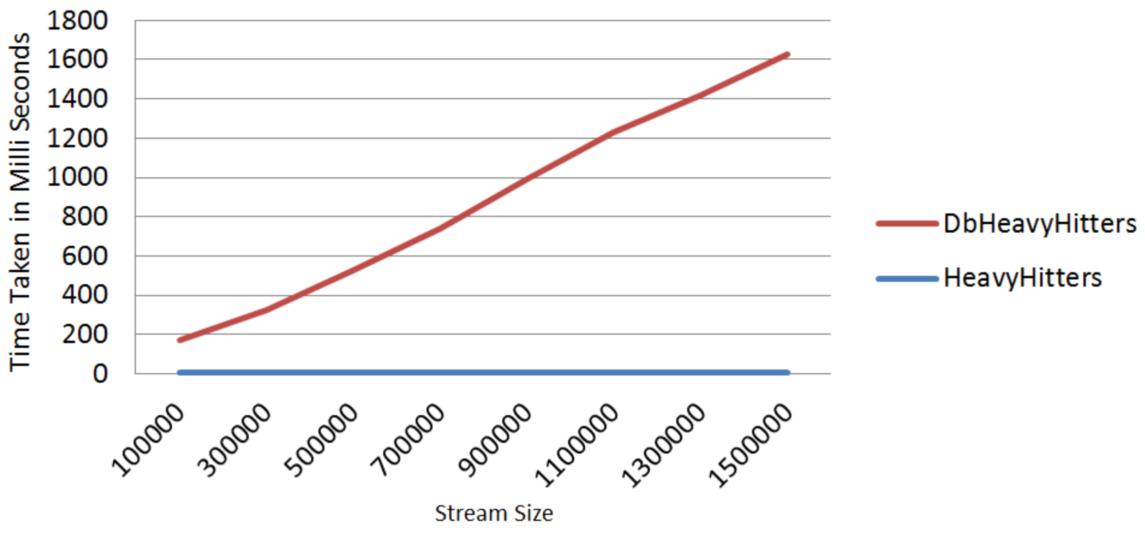


Figure 6.8: Execution time of heavy hitters query DBSMS vs DBMS

[Fig. 6.9] represents error observed when approximation algorithm is used for heavy hitters. DBMS answers the query exactly so error is zero.

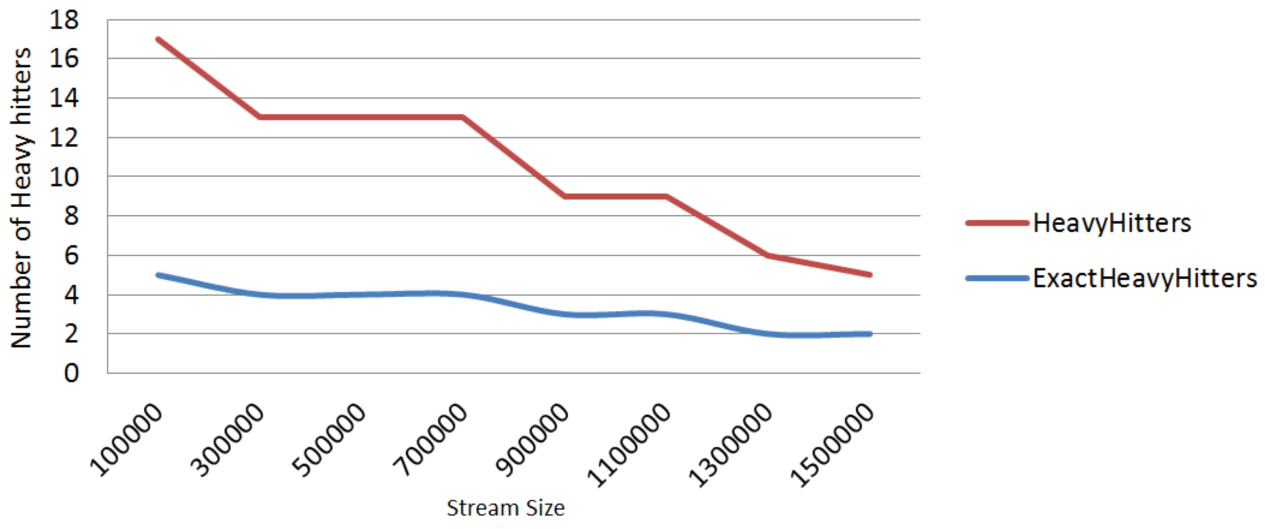


Figure 6.9: Number of Heavy hitters DBSMS vs DBMS

[Fig. 6.10] represents false positive ratio with respect to the exact answer provided by the DBMS.

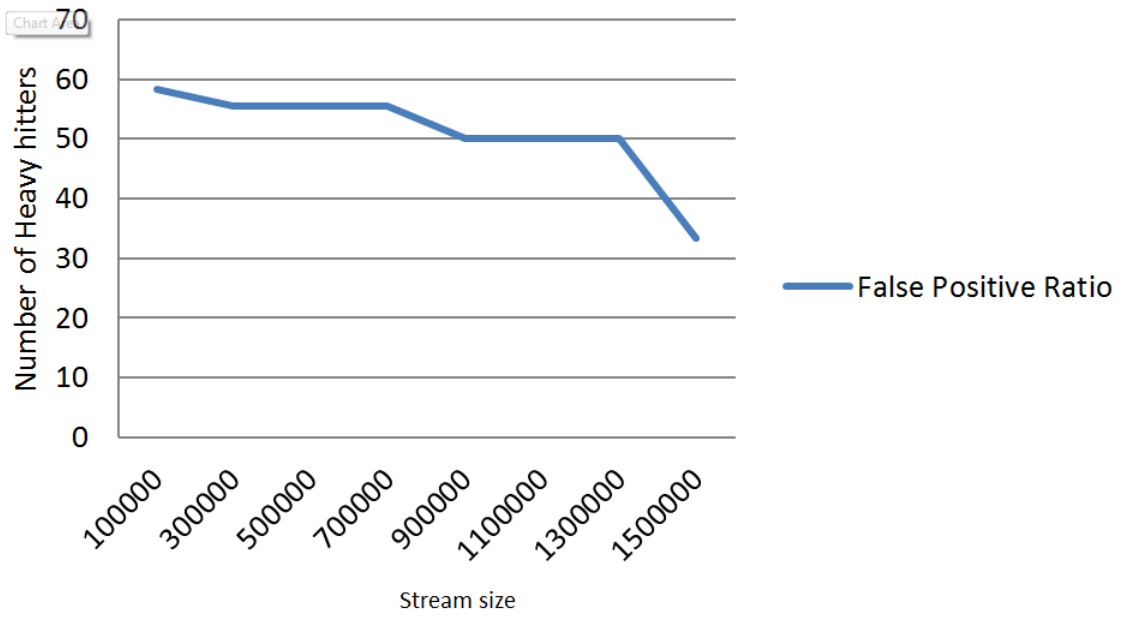


Figure 6.10: False positive ratio of heavy hitters

[Fig. 6.11] describes the error behavior for different storage space used by the algorithms. The storage space used is based on the ε, δ value provided by the user. As expected, the error decreases as the storage size increases.

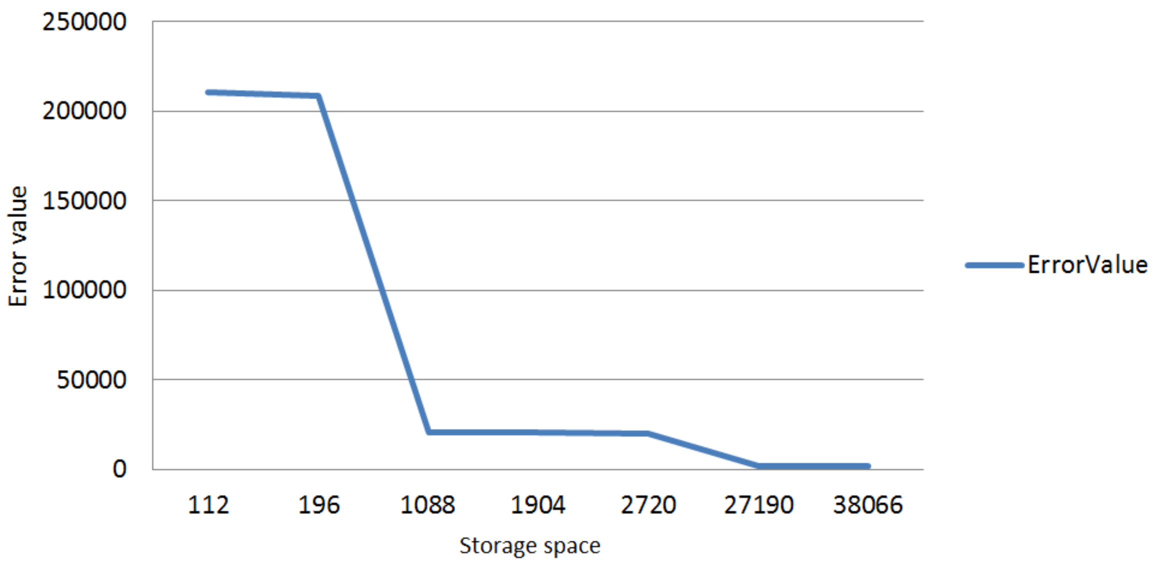


Figure 6.11: Error vs. Space

Chapter 7

Conclusion and Future Work

So far, we have presented in this report a design for generic database stream management system. We have also discussed the challenges that User controlled approximation. such a system should be capable of handling - query mapping into approximation operators, integrating approximate streaming algorithms into the DBSMS. persistent support, error propagation, query feasibility and multi query optimization. We have also presented our implementation of the system where we have introduced user error control and have partially addressed the challenges of query mapping, integrating the approximation algorithms into the system and persistent support. We now present a few open issues and other challenges to be addressed in the system design.

7.1 Error Propagation

While error control has been provided to the user, we still need to address the problem of error propagation. Error control has so far been provided only at single query level which maps to exactly one approximation operator. The algorithm implemented by the operator automatically creates storage structures to answer queries based on the ε and δ which are received as input from the user. However if the query is mapped to more than one approximation operators or if a complex query involves results from simpler queries, the system needs to calculate the the error propagation across the different operators.

For example, consider a query that reduces to a point and range query. The user provides the overall error for the single query and not at the level of operators to which it is mapped. The system should thus be capable of calculating the error values ε_1 and ε_2 that are to be used with the point and range query operators to ensure that the overall error ε is met.

7.2 Query mapping and Feasibility

Currently, the user has to mention the type of query (point, range, heavy-hitters and so on) at the time of registering the query. However, we would want such a mapping to occur automatically. The user has to input an SQL type query with the desired accuracy and the system using a set of well defined rules must be able to transform it to simpler queries that can be mapped with the existing approximation operators. For example, as seen above, queries with the following template should be automatically identified as point query and must be mapped to the point query approximation operator.

```
SELECT sum (value)
FROM stream
GROUP BY key
HAVING key = ?
```

Such sort of mapping is most desirable and yet to be incorporated in the system. Also, since we reduce a multi dimensional tuple to a two dimensional tuple, and store on the reduced tuple, we might not be able to answer queries which use other attributes of the tuple that were discarded. This leads us to introduce a check on whether a query on a stream is actually feasible or not. Such check on query feasibility currently does not exit and needs to be introduced into the system.

7.3 Multi-Query Optimization

Consider a point query registered on a stream *S*. If a range query has already been registered on that stream, then the same data structure can be used to answer the point query provided it satisfies the error specified. We can see that as opposed to the traditional query optimization which looks at only one query while generating the best plan for its execution, DBSMS has to check more than one query to generate the best plan for a given query. Given the continuous nature of the queries, the plans generated tend to be dynamic and hence multi query optimization plays an important role in the DBSMS. Multi query optimization is currently an area of active research and needs to be integrated into our system.

References

[1] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In IN ICDT. 2005 398–412.

[2] S. Ganguly, M. Garofalakis, R. Rastogi, and K. Sabnani. Streaming Algorithms for Robust, Real-Time Detection of DDoS Attacks. In Proceedings of the 27th International Conference on Distributed Computing Systems, ICDCS '07. IEEE Computer Society, Washington, DC, USA, 2007 4–.

[3] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001 79–88.

[4] M. Hoffmann, S. Muthukrishnan, and R. Raman. Streaming Algorithms for Data in Motion. In B. Chen, M. Paterson, and G. Zhang, eds., ESCAPE, volume 4614 of *Lecture Notes in Computer Science*. Springer, 2007 294–304.

[5] S. Muthukrishnan. Data streams: algorithms and applications. *Found. Trends Theor. Comput. Sci.* 1, (2005) 117–236.

[6] C. Aggarwal, ed. Data Streams – Models and Algorithms. Springer, 2007.

[7] K. Lakshmi and C. Reddy. A survey on different trends in data streams. In Networking and Information Technology (ICNIT), 2010 International Conference on. 2010 451 –455.

[8] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms* 55, (2005) 58–75.

[9] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. Technical Report 2004-20, Stanford InfoLab 2004.

Appendices

.1 Input component

Input Component defines the input structure of input to be sent to the DBSMS engine. Input structure means request to the system.

```
public enum RequestType {
    REGISTER, QUERY_REQUEST, SHOW, ACTION
};

REGISTER_REQUEST : Request to register an entity into the system
QUERY_RESULT : To Query the system
SHOW : Show properties and entities of system
ACTION : Request to control the system
```

Requests are classified into four types according to the functionality of the request.

.1.1 Register request

Register request is used to register an entity into the system. Register request divided into two types of register requests as stream register request and query register request.

.1.1.1 Stream register request

```
 public class StreamRegisterRequestImpl
     implements StreamRegisterRequest {

public StreamRegisterRequestImpl() {    }
@Override  public RequestType getType() {    }
public String getStreamName() {    }
public void setStreamName(String streamName) {    }
public ArrayList<String> getArguments() {    }
public void setArguments(ArrayList<String> arguments) {    }
public Integer getMaxUniqEle() {    }
public void setMaxUniqEle(Integer maxUniqEle) {    }
public StreamSourceTypesEnum getStreamSourceType() {    }
public void setStreamSourceType(
StreamSourceTypesEnum streamSourceTypesEnum) {    }
public String getStreamSourceArgument() {    }
public void setStreamSourceArgument
          (String streamSourceArgument) {    }

}
```

.1.1.2 Query register request

Query request to register a query into the system.

```
      public class QueryRegisterRequestImpl
       implements QueryRegisterRequest {

@Override public RequestType getType() {    }
public QueryRegisterRequestImpl() {    }
public String getQueryName() {    }
public void setQueryName(String queryName) {    }
public String getQueryArguments() {    }
public void setQueryArguments(String queryArguments) {    }
public String getStreamName() {    }
public void setStreamName(String streamName) {    }
public int getQueryTypeId() {    }
public QueryTypeInfo getQueryTypeInfo() {    }
public void setQueryTypeInfo(QueryTypeInfo queryTypeInfo) {    }
public String getQueryType() {    }
public void setQueryType(String queryType) {    }
public void isValidQuery() throws QueryException {  }

}
```

.1.2 Action request

Action request is used to control the system. Action request Types are defined in ActionRequestTypesEnum.

```
public class ActionRequestImpl implements ActionRequest {
    @Override public RequestType getType() {      }
    public ActionRequestImpl () {    }
    public String getActionRequestType() {    }
    public void setActionRequestType(String actionRequestType) {    }
    public void isValidQuery() throws QueryException {    }
    public ActionRequestTypeEnum getActionRequestTypeEnum() {    }
    public void setActionRequestTypeEnum
                  (ActionRequestTypeEnum actionRequestTypeEnum) {   }
}

public enum ActionRequestTypeEnum {
SAVE, START_ALL_STREAMS,  STOP_ALL_STREAMS,  START_STREAM, STOP_STREAM
};

SAVE : To save the state of the system to the disk.
START_ALL_STREAMS : To start all streams.
STOP_ALL_STREAMS : To stop all streams
START_STREAM : To start the stream using streams identifier.
STOP_STREAM : To stop the stream using streams identifier
```

.1.3 QueryResult request

QueryResult request is used to query the system. QueryRequestInputTypeEnum is used to specify query request type means on query using query name or query id or on stream name.

```
public class QueryRequestImpl implements QueryRequest {
    @Override public RequestType getType() {    }
    public String getQueryName() {    }
    public void setQueryName(String queryName) {    }
    public Integer getQueryId() {    }
    public void setQueryId(Integer queryId) {    }
    public QueryRequestInputTypeEnum getQueryRequestInputTypeEnum() {    }
    public void setQueryRequestInputTypeEnum(
     QueryRequestInputTypeEnum queryRequestInputTypeEnum) {    }
    public void isValidQuery() throws QueryException {    }
    public String getQueryArguments() {    }
    public String[] getQueryArgumentsArray() {    }
    public void setQueryArguments(String queryArguments) {    }
    public String getStreamName() {    }
    public void setStreamName(String streamName) {    }
}

public enum  QueryRequestInputTypeEnum {
    QUERYNAME, QUERYID, STREAMNAME
};

QUERYNAME : To query using queryname use query request input type as QUERYNAME.
QUERYID : To query using querid use query request input type as QUERYID.
STREAMNAME : To query on stream use stream name.
```

.1.4 Show request

Show request is to show system details. ShowRequestTypeEnum consists of details can be shown by show request.

```
public class ShowRequestImpl implements ShowRequest {
    @Override public RequestType getType() {    }
    public ShowRequestTypeEnum getShowRequestTypeEnum() {    }
    public void setShowRequestTypeEnum
            (ShowRequestTypeEnum showRequestTypeEnum) {    }
    public String getShowRequestType() {    }
    public void setShowRequestType(String showRequestType) {    }
    public void isValidQuery() throws QueryException {   }
}

public enum  ShowRequestTypeEnum {
    QUERIES,STREAMS,QUERYINFO,STREAMINFO
};

QUERIES : Shows registered queries with the system.
STREAMS : Shows registered streams with the system.
QUERYINFO : Shows registered query description.
STREAMINFO : Shows registered stream description.
```

.2 Configuration component

Configuration component consists entire system configuration. This configuration consists properties require to connect with the stream system to answer CQL queries from the DB-SMS system. This module also consists of system check point properties and all properties of the system.

.3 Stream component

Stream component is core component of the system. This component includes stream source, operator queue thread etc. System was designed tested over Key and Value pairs. Same concept can be extended to the rows also. Basic element in the system is Stream Element. Stream element is combination of Key and Value.

.3.1 Stream Element

Stream element is a colletion of a key and a value. But naturally stream element can be multidimensional also. In this system key is an integer. User is responsible to map key into corrsponding integer.

```
public interface StreamElement {
    Key getKey();
    Value getValue();
    Integer getStreamSourceId();
}
```

.3.2 Stream source

Stream source is responsible to generate and distribute generated stream and distribute stream elements to the all registered operators.

```
public interface StreamSource extends Serializable {
    void start();
    void stop();
    void registerListenerQueueThread(OperatorQueueThread opQueueThread);
    public OperatorQueueThread getStatistics();
    public void initDeSerializeObject();
    public OperatorQueueThread getDbStatistics();
}
```

.3.3 Operator queue thread

Operator queue thread is combination of a queue and thread. Thread will be informed when an operation was performed on the queue. By using this property derived class will be notified when an operation was performed in the parent class.

```
public interface OperatorQueueThread extends Runnable,Serializable {
    public void enqueue(StreamElement streamElement);
    public StreamElement dequeue();
    public boolean isEmpty();
    public int size();
    public void operationOnArrival(StreamElement element);
    public OperatorTypes getType();
    public QueryResult getResultSet(String[] arguments);
    public void serilaThreadInitiation();
    public void serilaThreadStop();
    public void initDeSerializeObject();
}
```

As persistency is one of most crucial property to the system. To synchronize system with the secondary storage below two classes will be used.

.3.4 Streams collection

This class will help to store all data related to the streams.

```
public class StreamsCollection implements Serializable {

    public static StreamsCollection
```

```
                getSingletonObject(String fileName) {    }
    public static StreamsCollection
                getSingletonObject() {    }
    private static StreamsCollection
                getStreamsCollectionObject(String fileName) {    }
    protected StreamsCollection() {    }
    public void startStream(String streamName) {    }
    public void startStream(Integer streamId) {    }
    public void startAllStreams() {    }
    public void stopStream(String streamName) {    }
    public void stopStream(Integer streamId) {    }
    public void stopAllStreams() {    }
    public boolean serializeObject() {    }
    public boolean serializeObject(String fileName) {    }
    public boolean isPresent(int streamId) {    }
    public boolean isStreamNameExists(String streamName) {    }
    public Integer getStreamId(String streamName) {    }
    public int totalNumberOfStreams() {    }
    public final synchronized boolean registerStream(String streamName
                , Integer streamId, StreamSource streamSource) {    }
    public synchronized boolean addListenerToTheStreamSource
                (Integer streamId, OperatorQueueThread opThread) {    }
    public synchronized boolean addListenerToTheStreamSource
                (String streamName, OperatorQueueThread opThread) {    }
    public int getNumberOfRegisteredStreams() {    }
    public void printStreamNames() {   }
    public StreamSource getStreamSource(Integer streamId) {    }
    public StreamSource getStreamSource(String streamName) {  }
    public void initDeSerializeObject() {   }

}
```

.3.5 Queries collection

This class will help to store all data related to the operator queue threads.

```
public class QueriesCollection implements Serializable {
    public static QueriesCollection getSingletonObject(String
      queiresCollectionFileName, String streamsCollectionFileName) {    }
    public static QueriesCollection getSingletonObject() {    }
    protected QueriesCollection() {    }
    private static QueriesCollection getQueriesCollectionObject(String
      queiresCollectionFileName, String streamsCollectionFileName) {    }
    private static QueriesCollection getQueriesCollectionObject(String
      queiresCollectionFileName, StreamsCollection streamsCollection) {    }
    public boolean isPresent(int queryId) {    }
    public boolean isPresent(String queryName) {    }
    public int totalNumberOfQueries() {    }
    public final synchronized boolean registerQuery(String queryName,
      Integer queryId, OperatorQueueThread OperatorQueueThread) {    }
    public int getNumberOfRegisteredQueries() {    }
    public OperatorQueueThread getOpThread(String queryName) {    }
    public OperatorQueueThread getOpThread(Integer queryId) {    }
    public int getSize() {    }
    public void setSize(int size) {    }
    public void stopProcessing() {    }
    public void initDeSerializeObject(StreamsCollection streamsCollection) { }
    public void reRegisterAllStreams(StreamsCollection streamsCollection) { }
    public boolean serializeObject(String queiresCollectionFileName,
      String streamsCollectionFileName) {    }
    public boolean serializeObject() {    }
}
```

Implementation classes are provided so that user no need to implement basic functionality. It is enough to provide core function which are needed. Some of implementation classes are StreamElementImpl, StreamSourceImpl, OperatorQueueThreadImpl. These implementation classes will make user task much easier.

.4 Algorithm component

Algorithms module is to plugin used defined algorithms into the system. Any algorithm can be easily plugged into the system by simply extending algorithm package.

```
public abstract class algorithm extends OperatorQueueThreadImpl { }
```

.5 Process to plugging in user defined stream source

- Stream Source must extend StreamSourceImpl class and make an entry into enum StreamSourceTypesEnum.

- Map class to the corresponding class name in StreamSourceObject.properties. This file consists details of corresponding class name and stream source type.

```
public enum StreamSourceTypesEnum {
     FILE, DBTABLE, RANDOM
}

StreamSourceObject.properties
FILE FILE
DBTABLE DbTable
RANDOM RandomStreamGeneration

public class TestStreamSource extends StreamSourceImpl {
    @Override    public void generateStream() {    }
    @Override    public void initAlgorithmDeSerialization() {    }
}
```

.6 Process to plugging in user defined operator

- Algorithm must extend algorithm class and make an entry into enum OperatorTypes.
- Map class to the corresponding class name in OpthreadObject.properties. This file consists details of corresponding class name and operator type.

```
public enum OperatorTypes {
     POINT_QUERY, RANGE_QUERY, HEAVY_HITTERS, TOP_K,
STATISTICS,DATABASE_STATISTICS, TEST_ALGORITHM
};
OpthreadObjectProperties.properties
POINT_QUERY  CountMinSketch
RANGE_QUERY  CMRangeQuery
HEAVY_HITTERS  HeavyHitters
TEST_ALGORITHM TestAlgorithm
public class TestAlgorithm extends algorithm{
    @Override    public void initAlgorithmDeSerialization() {    }
    @Override    public void operationOnArrival(StreamElement element) { }
    @Override    public QueryResult getResultSet(String[] arguments) { }
    @Override    public OperatorTypes getType() {    }
}
```