STREAMING DATA PROCESSING AND MANAGEMENT

A Seminar Report

Submitted in partial fulfillment of requirements for the degree of

Master of Technology

by

Sushmita Bhattacharya
Roll No : 133050022

under the guidance of

Prof. N. L. Sarda

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

March, 2014

Acknowledgements

I would like to express my deep gratitude to Prof. N. L. Sarda for his constant motivation, useful advices and engagement throughout the learning process of this seminar. His constant supervision and guidance helped me in accomplishment of this seminar. Without his deep insight into this domain and his valuable ideas for this seminar, it would not have been possible for me to move ahead properly. I would like to thank all who have helped me directly or indirectly in completion of the seminar.

Sushmita Bhattacharya
(Roll - 133050022)

Abstract

There are various range of applications where huge data are generated at a very high speed. For example data from web-logs, sensor network traffic, social media etc. These data need to be captured, processed, analysed and stored. Performing all these tasks simultaneously is a very hard problem. These data are generally termed as data streams. Unlike traditional DBMS, data streams are considered to be append only data where updates are less frequent. There are other requirements in processing data streams, such as they require unbounded amount of memory. So some part of the data element are discarded. Because of these reasons the queries are answered approximately.

In my seminar I have explored about streaming data management and analysis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data Stream is a sequence of digitally encoded signals used to represent information in transmission. Data stream generally comes continuously to the system for processing.

Data Stream can also be defined as the the ordered pair of (s, t) where s represents the sequence of tuples and t represents the sequence of positive intervals.

Data Stream comes continuously to the system. They should be captured, processed, analyzed, stored in real time. In addition to that, underlying data structure can also change. This is known as temporal locality.

The traditional database system processes data which is stored in disk. But huge amount of evolving data cannot be stored and retrieved from the store for real time processing on those data.

The data size is too high that one data should be processed only once. So traditional database algorithms (like data mining) should be modified for data stream such that they do not use multiple passes.

Continuous queries - accepting data as they arrive and producing output in a continuous fashion, is a very common type of query in streaming. Data can come before or after the query is registered in the system. Traditional databases are not designed for processing continuous queries. Traditional DBMS does not support continuous queries; the query needs to be registered to the system after data is stored in a persistent store.

The streaming data requires an unbounded amount of memory hence they allows some approximations on the answer of queries. But traditional DBMS try to solve the problem of exact answering of the queries, based on some fixed query plans.

This is why streaming data needs some special treatment for processing, storage and indexing, mining etc. So the traditional DBMS is not suitable for processing streaming data.

There are some applications of the streaming data. Examples of such systems include financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks generate data streams. In a large distributed network, huge pack-

ets are generated for routing, these need to be processed cleverly so that routing is performed within limited resource constraint. Another example is the network monitoring tool which joins streams. These require unbounded amount of storage. So some approximation techniques can be applied for answering queries in such systems. [1]

The report is structured as follows. Chapter 2 contains query language and other processing requirements like summarizing. It also contains stream data mining. Chapter 3 contains storage and indexing structure for conventional DBMS. Then it discusses why these structures are not suitable for data stream and some research ideas proposed for storage and indexing. Then it talks about a real implementation of streaming data management system by Stanford University - STREAM. Chapter 4 talks about special architectural platform proposed for streaming data namly map reduce for streams. Chapter 5 concludes the report.

Chapter 2

Query Language, Approximation and Mining

2.1 Query Language

Streaming Queries can be applied on both streaming and relational data. The processing of such query necessitates new query language. Such a query language CQL (Continuous Query Language, developed by Stanford University) is discussed here. [8]
Two types of data

1. **Streams** can be thought of unbounded, ordered (according to time of arrival),append only multiset of pairs <t,s>. The notation denotes tuple s arrives at time t.

2. **Relations** are unordered, and they support updates and deletions as well as insertions

Semantics of continuous query language

1. From clause may contain Relations, Streams or both.

2. From clause may be followed by an optional sliding window specification, and an optional sampling clause. Sampling clause specifies that a part of the whole input stream would be selected probabilistically for processing. For example Sample(2) can be used for selecting a tuple from input stream with probability 0.02.

3. CQL can contain two new operators - Istream and Dstream.

4. CQL window specification consists of an optional partitioning clause, a mandatory window size, and an optional filtering predicate. Partitioning clause partitions the input streams into different groups and computes the functions and merges them into one group for delivering the result. Window size is specified using either **Rows**(number

of rows specified) or **Range** (some attribute range should be satisfied for selecting a tuple from the window, exact number of rows is not specified). Filtering predicate is specified using where clause.

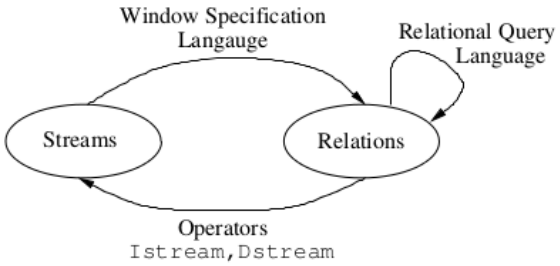

Figure 2.1: Mapping between Streams and Relation[8]

2.1.1 Mapping Operations

A stream can be transformed to a relation by introducing a window specification. Those specification includes CQL's Rows, Range, Partition By, and Where constructs. Application of these constructs to a stream produces a finite set of tuples thus forming a relation. Relations are transformed into streams using

1. **Istream** ("insert stream") applied to relation R containg a stream element <t,s> when s is in R at time t but not in time t-1.

2. **Dstream** ("delete stream") applied to relation R containg a stream element whenever tuple s is in R at time t-1 but not in R at t time

For evaluating result of a query at a time is obtained by taking all relations up to that time, all streams up to that time transformed to relations using window specifications. Result is relation unless the outer operation is Istream or Dstream.

The streams are ordered implicitly or explicitly according to the time of arrival. This is required for evaluating row based and time based sliding window query (Sliding window queries are those which are evaluated based on set of tuples in the recent past rather than all the tuples).

2.1.2 Examples

1. The following query finds the count of all requests from the corresponding stream which are generated from stanford.edu domain; the window clause represents that the tuples to be fetched should be

within 1 Day.
Select Count(*) From Requests S [Range 1 Day Preceding] Where S.domain = 'stanford.edu'

2. The following query finds the count of requests served by cs.stanford.edu, where each client's 10 most recent request from stanford.edu are to be considered.
Select Count(*) From Requests S [Partition By S.client_id Rows 10 Preceding Where S.domain = 'stanford.edu'] Where S.URL Like 'http://cs.stanford.edu/%'

3. The following query retrieves the client_id s from a list of client_id s in the request stream with a probability of 10%.
Select client_id from Request Sample(10)

2.2 Approximation and Analysis

Analysis of streaming data makes use of batch processing, random sampling, sketching techniques and histograms. The main motivation for using these kind of summary structures is, stream processing allows approximate answers within tolerable time and space limit.[1] [8] [5]
Approximation is needed due to

1. Queries and associate data requires unbounded amount of memory.
2. Multiple complex continuous queries has some timeliness requirements
3. Finite resources are available for query processing.
4. Some of the queries require recent data for evaluating answers. In such scenarios sliding window approximation is applied without loss of precision.
5. Input flow of data stream may overwhelm the query processor. So, to reduce the input rate, batch processing, sampling, synopsis structures are used.

2.2.1 Techniques for producing approximate query result, analysis

1. **Sliding Window** If the recent history is more important in answering query, then sliding window approximation technique is used. It is transparent to the users that which part of the input stream is punctuated in order to evaluate approximate query. There is no chance of producing random, bad approximation. The optimization of sliding window query is an open problem. As all the content of incoming stream cannot fit in memory or disk. Queries should be answered

only looking at the recent window. Its a very hard problem to find whether the recent window (window size rather) is sufficient for producing minimum error rate or not.

2. **Batch Processing** is used when update in data structure for new stream data arrival is fast but answer generation is slow. This approximation technique provides correct answer in an untimely basis. It is good for the cases where application cannot produce answer in peak load but can keep up with average load.

3. **Sampling** is used when computation of the answer is fast, but the update in data structure for incoming stream data is slow. Some of the tuples in the input stream may be skipped. Sampling-based approaches cannot give reliable approximation guarantees as probability that a tuple remains in the stream is completely random.

4. **Sketch or summary structure** Instead of keeping all the data in the stream a small summary of past data is maintained for query processing and future reference.

   (a) **Histograms** are succinct and space-efficient approximations of distributions of numerical values. Histograms are represented as a sequence of vertical bars whose widths are equal but whose heights vary from bar to bar. Varying-width histograms are also common. Histograms compactly represent piecewise-constant approximations of data distributions. Histograms are used in database for optimizing query executions and for approximate query processing.[5]

2.2.2 Types of Approximation

Available resources should be used in such a way that the query answer precision gets maximized. Efficient use of resources also helps the system to adapt with various change in requirements like varying data rates, query load etc.

1. **Static Approximation** deals with changing the query before execution for generating approximate query answers in a resource constraint system.

   (a) **Window Reduction:** Infinite data stream processing can be sped up by reducing window size or by introducing window in case there is no window in the input stream and processing is done only on the modified (or introduced) window.

(b) **Sampling rate reduction:** The output rate of queries can be decreased using sampling the input (if no sampling is done) or changing the sample rate of the input. This will require less resource and less response time for processing.

2. **Dynamic Approximation** The queries can be answered precisely when the load is low and approximately when load is high dynamically.

   (a) **Synopsis compression** Synopsis can be compressed for minimizing usage of memory and quick response time. When one synopsis is shared between multiple query plans, synopsis compression affects large number of query results and their response time.

      i. **Histogram** representation can be compressed.
      ii. **Sampling** can be used in dynamic approximation techniques as well. Depending on current load in the system, the sampling rate can be changed.
      iii. **Load Shedding** technique deletes chunk of data from the input stream when there is no space in the processing queue (unlike sampling where input is sampled probabilistically).

2.3 Streaming Data Mining

Data Stream Mining is the process of extracting knowledge structures, pattern from continuously, rapidly incoming data records.

Hypothesis can be generated and tested by exploring the available data. There are some specific problems regarding data streams - they consists of huge and continuously evolving data, where underlying data structure also changes rapidly. The data should be processed only once. So traditional machine learning techniques, statistical analysis techniques are modified to address these problem of mining of data streams. In a distributed system where data is coming at a very high speed, data mining can be done by finding out the global model of the data set, combining the local models at each station.

Recent development in the data management obviates the need for capturing data from small devices (such as PDA- with less computation intensive processor) and analysing and showing them at the user end. Huge data is captured and sent across the web for processing at the remote site. The satellite system also generates and analyzes huge data. Some computation is offloaded from the satellite core to the base station. This consumes huge bandwidth. There is a trade-off between the amount of computation done on those small devices and the use of network bandwidth. These monitoring and analysis should be continuous and real time. [4]

Some data mining techniques, their issues for data stream are discussed below. An data stream mining example is also discussed here. [9]

2.3.1 Techniques for Data Stream Mining

1. **Analysis Techniques**

   (a) **Data-based solutions** is an analysis technique where the subset of data is taken for analysis. The data is sampled vertically or horizontally for approximation.

      i. **Sampling** refers to the process of probabilistically choosing a data item to be processed or not. This technique is not preferred for unknown data size and fluctuating data rates.

      ii. **Load Shedding** It is a process of dropping a chunk of data from the stream. Load shedding suffers from the same problem as the sampling techniques and time dependent analysis cannot be performed with load shedding.

      iii. **Sketching** is a process of randomly projecting a subset of the features from the stream. It vertically samples the incoming stream. This can be used for aggregate queries but this technique is not accurate.

      iv. **Synopsis Data Structures** It is a technique for summarizing the incoming stream for further analysis. This includes Wavelet analysis, histograms etc.

      v. **Aggregation** It is a process of computing statistical measures such as means and variance that summarize the incoming stream. This method is not applicable for highly fluctuating data rate.

   (b) **Task-based solutions** are the techniques for time and space efficient solutions.

      i. **Approximation algorithms** The traditional approximate algorithms are modified for handling large data stream mining.

      ii. **Sliding Window** The end user is more interested in most recent data rather than past data. Sliding window is a technique to process stream using the recent data along with some approximation of the past data.

2. **Mining Techniques** Extracting knowledge from streaming information.

   (a) **Clustering** Data stream can be summarised by using clustering algorithms e.g K-means and K-median clustering algorithm.

   (b) **Classification** Data Streams can be classified using classification techniques e.g decision tree, linear regression model, k-nearest neighbour techniques etc for finding the pattern or grouping the similar data items.

2.3.2 Challenges and Open Issues of Data Stream Mining

There are several challenges in the mining of data streams. Few of them are discussed.

1. Continuous flow handling.
2. Minimizing energy consumption of the mobile devices.
3. Unbounded memory requirement for the flow of data stream.
4. Transferring data data mining results over a wireless network with limited bandwidth.
5. Sometimes the dynamics of the data is of user interest.
6. Visualization of data mining results on small screens of mobile devices.

Open Problem

1. Mining results verification is a costly operation. Some techniques are required for avoiding model overfitting.
2. Representation of data in a compressed way in small hardware having low computation power.
3. Capturing and acting upon the user feedback on the mining results, in a limited resource system.

2.3.3 Stream data Mining - An Example

One of the paper talks about **An Algorithm for In-Core Frequent Itemset Mining on Streaming data**. [7][9] This is an one pass algorithm for frequent itemset mining, which does not require any out-of-core summary structure.

Previously data mining was meant for databases which captured and stored data in a persistent store and analysed them later. But due to increase in need of data stream mining, all the data cannot be stored in a persistent store. Single pass over data to capture frequent mining set are required in a limited resource constraint system. There is an improvement of 5% over Apriori algorithm by using the proposed algorithm for streaming data.

Given a sequence of length N and a threshold θ, $0 < \theta < 1$, the goal of the algorithm is to determine the elements that occur with frequency greater than $N\theta$.

Initial algorithm was an iterative solution which stored each distinct element whose frequency was more than $N\theta$ with its count in a set. The

algorithm gave the superset of elements which occurred more than $N\theta$ times. Maximum number of distinct elements found was $1/\theta$. Second pass was required to generate a set whose element occurred more than $N\theta$ times. This algorithm required $O(1/\theta)$ space.

The refinement of the basic algorithm first finds frequent 1-itemset. Extension was done to this for frequent i-itemset with transaction fixed length.

Hybrid approach is proposed for finding frequent i-itemset. Frequent 2-itemset is found, then by applying Apriori algorithm frequent k-itemset (where k>2) is found. Apriori algorithm finds frequent k itemset if and only if all its subsets with (k-1) elements are frequent. Then second pass is used for eliminating false negatives.

Chapter 3

Storage and Indexing for streaming data and Case Study

3.1 Why Traditional DBMS Storage and Indexing not sufficient for Streaming

There is a very strict performance criteria for streaming data processing. Efficient data store, index structure play a vital role to achieve this goal. The storage for streaming data can be designed such that it can answer approximate query in small amount of time and space. The conventional DBMS stores all the past data in some store and uses indices on them, but in case of streams, the past data may not always be stored because of lack of space availability. So the storage system should maintain an internal state containing the summary of the past data. These designs helps answering continuous queries (e.g continuous joins) which are very common to streams but not in traditional DBMS. If queries are known before the design of data store, queues can be maintained and shared between operators, use appropriate indices to optimize the time and space required for execution.

3.2 Storage

Previous stream data storage designs were dedicated for a certain application. So they were inflexible. Traditional DBMS uses the concept of flexible data store which can be customized for different applications dynamically. Data stream can also use the similar concept of general purpose storage design. For designing such a store, tunable parameters are to be identified. These parameters help optimizing the data store design. For example, if read pattern is known beforehand, it will be easy to design data structure and indices for the store. Then some interfaces should be there by using which, those tunable parameter values can be changed.

There is a system implementation proposed in the literature [3]. It is

called Storage Manager for Streams (SMS). This is a customized data storage decoupled from the application and provides interfaces for instantiating tunable parameters of the storage system.

3.2.1 Tunable Parameters Identification

1. **Architectural parameters**
   (a) **Role** defines how does data come from stream to store. An **Active** store pulls data from stream to store, whereas in **Passive** store, SPE (Stream Processing Engine) pushes data to store from stream.
   (b) **Access Model** defines how does the data come from store to SPE(Stream processing engine) for processing. This can be either push model or pull model.
2. **Functional Parameters**
   (a) **Synchronization and schema** Synchronized store represents that data can be consumed only after some condition is met on some attribute. Schema knowledge should be known for implementing this.
   (b) **Persistence** indicates whether to store data in memory or in disk.
3. **Performance related parameters** are related to memory usage and response time.
   (a) **Access Pattern** Knowledge of read, write, update pattern helps to design optimized store. **Update pattern** refers to operator states and inter-operator stores get modified as a result of new data arrival, query processing, as well as outdated data eviction. Update happens to the store either when result data expires (indicated by producer's pattern) or data gets consumed (indicated by consumer's pattern).
   Producer operator's update patterns are
      i. **Never Expire** SELECT Attr1, Attr2 FROM Stream
      ii. **Unordered Expire** ON Stream S DELETE FROM Store W WHERE W.Attr1 = S.Attr1
      iii. **Ordered Expire** SELECT Attr1, Attr2 FROM Stream WINDOW 24 HOURS
      iv. **Replaced Expire** ON Stream S UPDATE Store W SET W.Attr2 = W.Attr2 + S.Attr2 WHERE W.Attr1 = S.Attr1

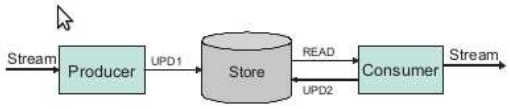

Figure 2: Operations on a store

Figure 3.1: Operations on Store[3]

Consumer operator's update pattern Unlike producers, where data is eagerly deleted from the system on expiration of data; consumer data can be deleted lazily from the store after being consumed.

i. **Never Consume** SELECT * FROM Store
ii. **Ordered Consume** SELECT Attr1, Attr2 FROM Store WINDOW 24 HOURS
iii. **Eager Consume** SELECT FIRST(S.Attr1), LAST(S.Attr1) FROM Store S WINDOW 24 HOURS

Update of Store The update pattern of the Store is determined by the update pattern of the Producer operator; because when producer's data expires, eager deletion occurs. Only when the producer has the Never Expire update pattern, the update pattern of the Store is determined by the update pattern of the Consumer operator. Types of store update patterns are

i. **NO-UPDATE** no fixed behaviour imposed by the Producer-Consumer operators
ii. **IN-PLACE** value replacements by key
iii. **RANDOM** unordered execution of delete operation by producer indicates such store.
iv. **FIFO** never expire, or they expire in an ordered fashion indicates such store.

Read Pattern denotes how consumer operator requests from store.

i. **Sequential Read** SELECT Attr1, Attr2 FROM Store WINDOW 24 HOURS
ii. **Random Read** SELECT Attr1, AVG(Attr2) FROM Store WINDOW 24 HOURS GROUP BY Attr1
iii. **Clustered Read** SELECT S.Attr1 FROM Store S WHERE PREV(S.Attr1) != S.Attr1 AND S.Attr1 ¡ NEXT(S.Attr1)

Sharing and access pattern Multiple consumers can share data store for optimization purpose. The update pattern of the store will depend on the update pattern of the different consumers.
If at least one of the consumer operators has a Never Consume update pattern, then the combined update pattern should be Never Consume. If all consumers follow an Eager Consume update pattern, then the combined update pattern may either be Eager Consume or FIFO. All other cases generate Ordered Consume update pattern.

(b) **Sharing** Some continuous queries running in parallel sometimes exhibit similarities. Thus sharing computation, intermediate results and operator state can improve multi-query optimization.

3.2.2 SMS Architecture and implementation

DSMS has two components - Stream Processing Engine and Storage Management System. SPE estimates various parameters and access patterns by looking at the queries. Those values are communicated to SMS which creates store instance that meet the SPE's requirement. Various components of SMS are described below.

Store Instance is an instance of a customized store.

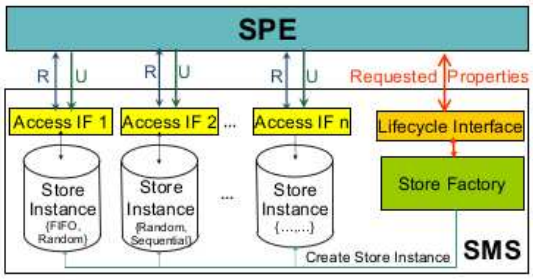

Figure 3.2: DSMS architecture with SMS[3]

Lifecycle Interface used by the SPE to specify creation and gaining access to store instances and specify the parameters.

Store Factory actually creates, manages and deletes the store; commands and parameters are taken from the lifecycle interface.

Access Interface enables the SPE to access a specific store instance. It provides basic per-item (and per-attribute) operations such as read and update.

Random store is designed using linked list. FIFO store can be implemented using circular list.

3.2.3 Performance

The update and read pattern are very important parameters for designing an optimal store. The resource sharing is also a major factor for improving the performance in multi-query optimization.

3.3 Indexing on sliding window queries

Continuous queries are very common in streaming data. Sliding window query is one of the most frequently used continuous queries, which deals with window of data coming continuously in the system, and the old data inputs lying outside of the current window are being discarded . Queries on these type of data need to access new type of index - sliding window index [6]. The data store also supports the sliding window storage. The current window is divided into several basic windows. Each time new data comes in, stored into the basic window. As and when the new basic window gets filled up, it is added to the current window and last basic window is deleted from the current window. The aggregate queries are evaluated easily using in this structure (the aggregate value of the current window - aggregate value of last basic window + aggregate value of the newest basic window).

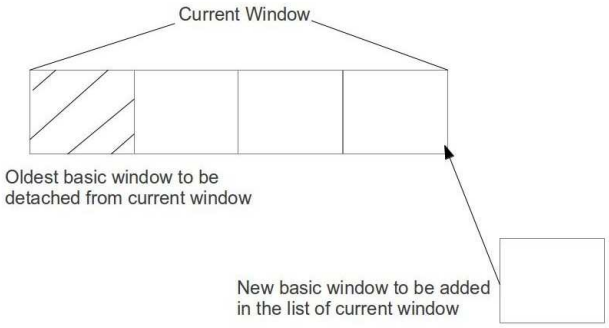

Figure 3.3: Sliding Windows

Assumptions

1. There are two types of windowed queries. One type of query is used to find the results satisfying some attribute value and another is set valued queries (required for intersections and joins). Different index structures are proposed for different queries.

2. There exist a sliding window consisting of N recent tuples. There can be two types of sliding windows, one is count based (storing N latest

items) and other is time based (storing recent items which occurred within time T). The windows are stored in main memory.

3. Each tuple has timestamp associated with it. Timestamp can be implicit (incoming order) or explicit (provided with the data)

3.3.1 Physical storage method and Maintenance

Each window is stored in a structure and there is a circular list of pointers. Newest window fills up and a pointer is attached to the newest window discarding the pointer to the oldest window in the list. The timestamp of the window is the timestamp of the oldest record in the window. Discarding the individual timestamp introduces small error in time calculation. But it saves space for storing records and gives rise to easy computation.

There are several window storage structures. LIST (list containing values for the window), AGGR (sorted list of distinct values with count), HASH (hash buckets for storing windows and actual entries), GROUP (range of values with their counts)

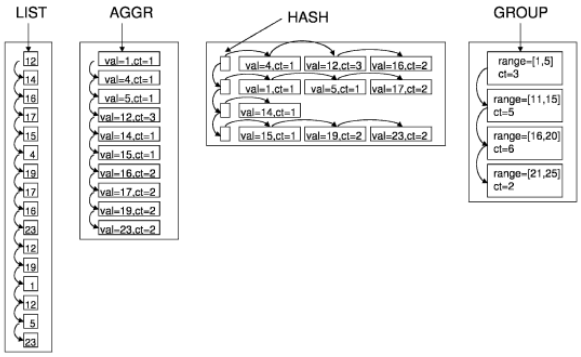

Figure 3.4: Basic Window Storage[6]

Queries can be re-evaluated after k windows are added. The query evaluation can also be scheduled more frequently for more important queries and less frequently for less important queries.

3.3.2 Input output modes

There are different types of input and output modes

1. Non-windowed input and non-windowed output: Tuples are consumed as soon as they arrive, results are generated as soon as possible.

2. Non-windowed input and windowed output: Tuples are consumed as soon as they arrive, output is materialized in sliding window.

3. Windowed input and non-windowed output: Windows are defined over input, results are generated as soon as possible.

4. Windowed input and windowed output: Windows are defined over input, output is materialized in sliding window.

3.3.3 Indices for Set valued Queries

There are several types of indices proposed for Set valued Queries.

1. **L-Index** Index containing list of distinct attribute values with their frequencies. The index is ordered by values. With LIST window each tuple has a pointer to a index node, with HASH and AGGR each distinct value in each basic window has a pointer to the L-INDEX. Each insertion needs to search the whole index.

2. **T-index** Scanning the entire index in the L-index is expensive. Self balancing tree index can be used to avoid full index scan.

3. **H-Index** Buckets containing linked lists of values and frequency counts. The entries are sorted by value.

4. **G-Index** is an L-INDEX that stores frequency counts for groups of values

Purging Index is done by deleting nodes of list/hash structure whose count reach to zero. Delayed deletion can be done, to reduce index restructuring. A data structure for zero count element can be kept for avoiding index scan while purging.

3.3.4 Runtime for Set-Valued Index

Complexity for **Inserting an element** in the index: Some terms are defined as follows.
d - Number of distinct values
D - Number of distinct windows
b - Number of tuples in a basic window
g - Number of groups in GROUP
h - Number of buckets in HASH.
b and d do not vary across basic windows.

The worst-case, per-tuple cost of inserting items in the window is O(1) for LIST, O(d) for AGGR, O(h) for HASH, and O(g) for GROUP. The space requirements are O(b) for LIST, O(d) for AGGR, O(d+h) for HASH, and O(g) for GROUP. Hence, AGGR and HASH save space over LIST, but are more expensive to maintain, while GROUP is efficient in both space and time at the expense of lost accuracy.

Index maintenance cost have two components one is the number of times the index needs to be visited for the insertion of an element and how expensive each scan is. The **number of times the index needs to be visited** is 1 for LIST and d/b for AGGR and HASH. **Cost each scan** is D for the L-INDEX, log D for the T-INDEX, and D/h for the H-INDEX.

G-INDEX is expected to be the fastest, while the T-INDEX and the H-INDEX should outperform the L-INDEX if basic windows contain multiple tuples with the same attribute values.

	L-INDEX	T-INDEX	H-INDEX
LIST	$O(D)$	$O(\log D)$	$O(\frac{D}{h})$
AGGR	$O(d + \frac{d}{b}D)$	$O(d + \frac{d}{b}\log D)$	$O(d + \frac{d}{b}\frac{D}{h})$
HASH	$O(\frac{d}{b} + \frac{d}{b}D)$	$O(\frac{d}{b} + \frac{d}{b}\log D)$	$O(\frac{d}{b} + \frac{d}{b}\frac{D}{h})$

Table 3.1: Per Tuple Cost Index Maintenance[6]

3.3.5 Indices for Attribute valued Queries

This queries are essential for accessing individual tuple. **Implementation of indices for Attribute valued Queries**

1. **Window Ring Index** The index structure is like a ring where index node points to the data node, each data node is connected to its next node in order and last node of the corresponding value is pointed back to the index node. Deletion of oldest window is done after newest window fills up.

2. **Faster Insertion with auxiliary window** A Temporary local ring index is maintained for newest basic window. Insertion of that auxiliary window is done after this window fills up.

3.3.6 Performance

Using **Set-valued index** L-index is the slowest and least scalable among all other indices. Using HASH window is cheaper than using LIST AGGR. HASH window works poorly when number of distinct values is large. Performance of the T, H index are the same. Performance of the G-INDEX is the best among all other indices available.

Indices for **attribute valued** window query performance depends on size of basic window, number of distinct values in each basic window.

3.4 Case study: STREAM: (STanford stREAm Management System)

The STanford stREAm data Manager is a real implementation of the streaming data management system. STREAM executes continuous queries over multiple continuous streams. The system accepts both relational query and long running (generally one time) continuous query on stream.

A declarative query language is also provided for easy retrieval of information. This query language supports both relational query semantics as well as stream query semantics. The query language used by the STREAM is called **CQL (Continuous Query language)** which is already discussed in 2.1.

This system provides a way to support plan and resource sharing between operators to improve the time and space efficiency.

This system uses sampling techniques and synopsis (short summary of data seen so far). Load shedding is also implemented (Load shedding is a process of elimination of chunk of data from the queue at once when size of queue increases.[1][8][2]

Issues such as Query optimization is also addressed.

3.4.1 Query Plans

Queries are registered into the system before it starts generating output. Query plan is associated with different components.

1. **Query operators** Query operator reads data streams from one or more input queue(s) and after processing writes back to an output queue.

2. **Inter-operator queues** Queues can be shared between different operators.

3. **Synopses** State of operators can be saved in synopses data structures. Synopsis stores summary of data to a operator received so far. These are needed for future query evaluation. Examples of such synopses are fixed-size hash tables, sliding windows, samples and histograms etc. Type of synopsis depends on query. Query looking for a particular value may need hash-table synopsis, query looking for a full window scan requires sliding window synopsis. RANGE window specifications has unbounded synopsis size, whereas ROWS window specifications can be estimated using tuple size, number of rows etc.

3.4.1.1 Resource Sharing in Query Plans

Synopsis may be shared between multiple query plans when involved queries have similar window size, sampling rate or similar filters. The queue can

also be shared with multiple queries with similar sub-expression. The rate of consumption and production of the operators should also be similar for sharing the resources in query plan.

3.4.2 Resource Management

Additional information about streams can be used to reduce the synopsis size. For example while joining R and S. If all tuples of R comes before S, there is no need to store synopsis for S. If Tuples of S are clustered by join key, tuples of R of a given value join key can be discarded after tuples with new key arrives in S.
The operators can be scheduled such that peak total queue size can be minimized during query processing. Operators can be scheduled greedily so that operator which consumes largest number of tuples (or producing lowest number of tuples) in unit time gets highest priority.

3.4.3 Approximation Techniques

These techniques are discussed in 2.2.

3.4.4 Implementation Details

1. **Power Users** have the ability to visualize and influence system behavior.

2. **Entity and Control Table** Operators, queues and synopses are subclasses of generic class Entity. Each instance of such entities is associated with a control table (CT). Each entity updates Control Table using some interface. Using Control Table the behaviour and attribute of an entity can be changed (e.g How much memory is required by Synopsis can be specified using the interface for updating the Control Table). Control Table can also be used for keeping track of the statistics of a query (e.g count attribute of Control Table for a queue denoting number of tuples passed through the queue).

3. **Query Plans** Query plans can be created, viewed, understood, and manually edited in order to perform query optimization. Graphical interfaces are designed and implemented using CT's generic interface.

4. **Programmatic and Human Interfaces** Users can write application from remote station in any language for registering query, reading and updating CT. This can be done using HTTP requests. The output is sent to the end user in XML form.

Chapter 4

Streaming data Architecture: Special Platform

4.1 Why Map reduce?

Model or Paradigm MapReduce paradigm is famous for processing large data sets in parallel and distributed fashion on a cluster of nodes. The similar paradigm can also be used in case of streaming data. One of the paper [10] describes and implements such an idea. The idea is to modify the MapReduce model for streaming data to handle pipelining of data between operators of map and reduce jobs. Thus faster completion time and improved resource utilization can be achieved for batch jobs. The control is returned early as compared to traditional MapReduce model because of the data pipelining. This is particularly important in the cases where the end user wants to see rough estimation of the job far before the job completion.

4.2 Traditional Map-Reduce Task Architecture

Traditional MapReduce model materializes and ships data of intermediate stages from producer to consumer periodically. Data to be shipped can be pre-aggregated to reduce network utilization.

Traditional map-reduce paradigm for batch processing is fault tolerant. Output of each map reduce task is materialized in local file before it can be consumed by the next stage. This works as checkpoint at the time of failure.

HDFS used for storing initial input and final output of a job. Intermediate result files are stored in the local file system. Master node (job tracker) accepts job from client, divides it into tasks, assign tasks to nodes. Task manager manages execution of tasks at each node and notifies the job tracker about the completion of the task.

4.2.1 MAP

Split is a block in HDFS given to a map task by task manager. There are two steps in Map task

1. Task's split file is read and parsed into records of key-value pair, map function is called for each of these record.

2. After committing of each map task all spill files are merged into single index, data file pair and registers these files with task tracker. The data files contain records sorted by key. Task manager followed by job tracker gets notification after completion of the Map task. Tasktracker reads these files for servicing requests from reduce jobs.

4.2.2 REDUCE

There are three phases in Reduce task

1. **Shuffle** Every map's output for the group of keys to be reduced by the reducer are gathered at reducer task

2. **Sort** Sorting and Grouping of records for each key are done.

3. **Reduce** User defined reduce function is invoked for each key.

Map completes its execution totally, then reduce starts working. Each output of reduce task is materialized in temporary location in HDFS, task tracker renames and stores them in a permanent location of HDFS after all the reducers complete their work.

Output of both map and reduce is stored in local disk to guarantee the fault tolerance.

4.3 Pipelining MAP-REDUCE

Pipelined version of Hadoop (HOP Hadoop Online Prototype) is a modification to MapReduce architecture in which intermediate data is pipelined between operators and not materialized.

Characteristics of HOP

1. Reduce is started as soon as Map done. By doing this rough initial estimation of result can be found very fast.

2. Modified Map task pushes data to reducer as soon as it is produced.

3. It supports continuous query, which is a very common type of query in data stream. By doing this result evaluation and analysis can be done on real time basis in HOP.

4. System utilization is improved and response time is reduced by pipelining data between operators.

4.3.1 Naive pipelining

Map tasks determine which reducer should reduce its output and push its output to the respective reducers. When a reduce task is initiated, it contacts with its respective map task to get map output by using TCP socket. Thus data gets pipelined between mapper and reducer. After all the map tasks for a reducer is completed, all spil files are merged by the reducer.

4.3.2 Problem with Naive Pipeline and its solutions

1. Large number of TCP connection between mapper and reducer were established. To minimize this, fixed number of mapper are assigned per reducer at a time. The mapper for which reducers are not scheduled, stored its output in local buffer, upon initiation of the reducer, data in the local buffer is pushed to the reducer.

2. Same thread was used for producing map output and pushing data to reducer. Map output production may be blocked due to network congestion for pushing data to the receiver. Separate threads can be used to solve this problem.

3. Eager pipelining prevents pre-aggregation at mapper, which causes generation of more TCP packets and hence congestion in the network. This can be avoided by combining the map output buffer to a spill file, send the spill file (not the buffers immediately) after they are filled. Rate of sending of the spill files depends on network load and reducers' processing speed.

4.3.3 Job Pipelining

System consisting of multiple jobs can also be pipelined. Output of the reducer of the previous job is sent to the mapper of the next job as it is produced. Execution of mapper of the next job and reducer of the previous job cannot be overlapped. For solving this online aggregation with snapshot output can be used for pipelining data between jobs. Independent jobs can be scheduled concurrently. Upstream jobs are given more importance than downstream jobs.

Fault tolerance can be taken care in HOP. Map task can fail. So reducer keeps its output as tentative spill file. Once job tracker confirms the commit of the map phase, reducer combines the spill files. Otherwise the tentative spill files are discarded and new spill files are used when map is restarted.

Reducer can fail. Map do not discard the its output data as soon as they are sent to the reducer. Rather it keeps those until job tracker notifies it that reducer is committed. If reducer fails just the reducer should be restarted. Checkpoint can also be done for avoiding unnecessary redoing of map. If maps spill file is successfully written, it would not be redone.

4.3.4 Online aggregation

Interactive data analysis is another very important class of problem. Traditional Hadoop can be extended to perform quick and dirty approximation of the result before proceeding to the next step in a chained multi-job scenario.

Result of the final reduce task is evaluated only after all its corresponding map tasks are completed. Summarized data collected so far can used to generate snapshot. Snapshots are stored in a predictable location of HDFS. Rate at which snapshots to be generated can periodic or user specific.

How often snapshots can be taken can be found out by using a metric called Progress metric. It reflects how much fraction of the input is consumed by the mapper and reducer. Progress score of the snapshot is the average of the progress of each spill file. Fault tolerance can be handled as follows. If there are two jobs j1, j2. Order of execution is j1 followed by j2. If j1 fails, j2's mapper uses the last snapshot in HDFS and replaces it if another snapshot having higher score is available. If j2 fails, then j2 can be restarted and latest (with higher score) snapshot produced by j1 is used.

4.3.5 Continuous Query

Continuous queries can be supported in HOP. Map Reduce continuously gathers data and analyse it immediately or after certain specified interval.

For fault tolerance, map stores its output in a ring buffer for sometime, when gets notification from job tracker that reducer has successfully processed those data, garbage collection is done at mapper. This means storing buffer data to spill file. If reducer needs large amount of mapper output, the ring buffer may be overwritten. To avoid this some of the map output is swap out in HDFS.

4.4 Performance

In traditional MapReduce architecture Sorting at mapper reduces the work at shuffle phase of reducer. But pipelining reduces the scope of map side sorting.
Reduce phase of pipelined version is more time consuming as map do not sort before sending spill file. Reduce task can be started earlier as pipelining is involved.

Chapter 5

Conclusion

Streaming data systems are very common in real life. It has huge range of applications starting from network packet flow to sensor application where data is continuously feeded from the environment. Such data needs to be captured and analyzed. Sometimes an action should also be taken for handling exception scenarios, or predefined scenarios. Efficient processing of such data is a very important topic of research.

Various data in different formats and sizes coming from various systems (sensors, PDA etc) should to be processed in real time. The data interoperability should also be taken care of.

Traditional DBMS is not suitable for handling the various issues of the streaming data. So DSMS (Data Stream Management System) is proposed for addressing those issues. The query language, mining techniques are modified to handle large and continuous incoming data streams. The storage and indexing are also modified. Finally some special architectural framework for distributed system like Map-Reduce is modified to use them for data stream.

There are some blocking processes (like aggregation and sorting), which cause the data stream application to slow down. There can be other issues like, some queries need to access some passed by data. The data to be stored such that such queries are answered efficiently and correctly. These issues are partly addressed and needed to be fully addressed in future.

References

[1] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[2] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, September 2001.

[3] Irina Botan, Gustavo Alonso, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. Flexible and scalable storage management for data-intensive stream processing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 934–945, New York, NY, USA, 2009. ACM.

[4] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: A review. *SIGMOD Rec.*, 34(2):18–26, June 2005.

[5] Anna C. Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 389–398, New York, NY, USA, 2002. ACM.

[6] M. Tamer Özsu Lukasz Golab, Shaveen Garg. On indexing sliding windows over online data streams. Advances in Database Technology - EDBT 2004, Lecture Notes in Computer Science Volume 2992, 2004, pp 712-729.

[7] Yasuhiko Morimoto. Mining frequent neighboring class sets in spatial databases. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 353–358, New York, NY, USA, 2001. ACM.

[8] Arvind Arasu Brian Babcock Shivnath Babu Mayur Datar Gurmeet Manku Chris Olston Justin Rosenstein Rohit Varma Rajeev Motwani,

Jennifer Widom. Query processing, resource management, and approximation in a data stream management system. Standford.

[9] G. Ruoming Jin, Agrawal. An algorithm for in-core frequent itemset mining on streaming data. Data Mining, Fifth IEEE International Conference on.

[10] Peter Alvaro Joseph M. Hellerstein UC Berkeley Tyson Condie, Neil Conway. Mapreduce online. NSDI, 2010 from `static.usenix.org`.