Runtime Reprioritization for Online Aggregation Queries

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in
the Graduate School of The Ohio State University

By

Karthik Tunga Gopinath, B.E

Graduate Program in Computer Science and Engineering

The Ohio State University

2013

Dissertation Committee:

Dr. Arnab Nandi, Advisor

Dr. Srinivasan Parthasarathy
Copyright by
Karthik Tunga Gopinath
2013
Abstract

The need for interactive ad-hoc analytics has been steadily rising. The traditional batch processing of large data takes order of seconds to several minutes for query execution, which cannot guarantee an interactive experience. With ever increasing size of data, gaining an interactive experience is non-trivial. Online aggregation solves part of the problem by returning estimated results at regular intervals. An online aggregation system allows the user to observe the progress of the query but falls short when it comes to controlling the query execution. We propose an online aggregation system that takes query priority as a dynamic user defined input, allowing the user to control the progress of query execution by adjusting the priority at runtime. The framework derives concepts from online aggregation and shared scan query execution model, coupled with our priority based scheduling to allow for dynamic user defined query reprioritization. We also show how the proposed system behaves for different underlying storage formats.
This document is dedicated to Claudia.
Acknowledgments

I would like to convey my gratitude to my advisor Dr. Arnab Nandi, for giving me the opportunity to be part of his exciting and young research group. He has been my source of inspiration and I would like to thank him for all guidance he has given throughout my research. I appreciate his ability of identifying crucial and challenging research problems as well as giving me the freedom to learn and explore different aspects of the field on my own. His high standard of expectations has always kept me pushing for more.

I would like to thank Prasanth Jayachandran and Niranjan Kamat for their hard work and dedication to research. I have lost track of the countless caffeine filled late night programming sessions that we had in the department lounge. For most part of our research, we followed “get home before the sun rises” principle, which is evident in our code check-in patterns. Without their passion for solving interesting problems I do not think I would have got this far.

I would like to thank the entire datateam for providing all the support that they have. They were always present during our submissions and appreciate them for that.

Next I would like to thank my awesome friends and roommates Puneeth and Rohan for all their support. This most certainly would not have been possible without them.
I would also like to mention my close friends – Abhishek for all the free food, Chatty for all this brilliant puns, Farhang for his simple “NO” replies, Manas for all the late night walks to get food, Vineet for his sheer dedication, Tarun for his “you are at least making progress, I am doing nothing” talks and finally Ramya for her amazing home cooked food.

I will always be short of words to thank my parents for their love and all their sacrifices without which I would not be who I am today.

Finally, Claudia, for always being behind me every ever I go. She is a huge part of what I am today. She alone has the power of saying a lot without saying anything at all.
Vita

May 1987 .......................................................... Mangalore, India.

2005 .......................................................... Innisfree House School, Bangalore, India

2009 .......................................................... B.E, R.V College of Engineering, Bangalore, India

June 2012 – August 2012 ............................... Intern, Microsoft, Redmond

Jan 2013 to present ................................. Graduate Research Associate, Department of Computer Science and Engineering, The Ohio State University

PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer Science and Engineering
# Table of Contents

Abstract ............................................................................................................................................. ii

Acknowledgments .......................................................................................................................... iv

Vita .................................................................................................................................................... vi

Publications ....................................................................................................................................... vi

Fields of Study .................................................................................................................................. vi

Chapter 1: Introduction ...................................................................................................................... 1

Chapter 2: Related Work .................................................................................................................... 3

Chapter 3: Online Aggregation ......................................................................................................... 5

  3.1 Implementing an online aggregation system .............................................................................. 5
    3.1.1 Architecture .......................................................................................................................... 6
    3.1.2 Event based VS polling ......................................................................................................... 9
    3.1.3 Calculating Estimates and Confidence Interval ..................................................................... 10

Chapter 4: Bounded Time Execution ............................................................................................... 13

  4.1 Why is bounded time execution important? .............................................................................. 13
  4.2 Actor Model .............................................................................................................................. 14
  4.3 DICE Description ...................................................................................................................... 16
  4.4 Akka .......................................................................................................................................... 17
List of Tables

Table 1: List of notations used.......................................................... 11
List of Figures

Figure 1: System Architecture Diagram ................................................................. 7
Figure 2: Mapreduce using actor model .................................................................. 15
Figure 3: Actor ...................................................................................................... 18
Figure 4: Shared Mailbox ..................................................................................... 20
Figure 5: AutoExpireMailbox ............................................................................. 24
Figure 6: Demonstrating the effect of change in priority ....................................... 28
Figure 7: Illustrating absence of Query Starvation .............................................. 35
Figure 8: Hysteresis curve for different values of CONSTANT .......................... 39
Figure 9: Control flow in the execution engine .................................................... 40
Figure 10: Varying Priority with independent scans ............................................ 43
Figure 11: Elevator scan ..................................................................................... 44
Figure 12: Parallel Execution of query ................................................................ 48
Figure 13: Shared Scan Optimization .................................................................. 50
Figure 14: Varying Priority with shared scans ..................................................... 51
Figure 15: Lock and Lock-free approach ............................................................ 53
Figure 16: Lock and Lock-free block execution time .......................................... 54
Figure 17: Lock and Lock-free total execution time ............................................. 56
Figure 18: Execution time breakdown ................................................................ 57
Figure 19: Comparing Row Storage and Column Storage ................................. 59
Chapter 1: Introduction

Traditionally most databases are like black boxes. Everything inside a database is isolated from the user. The user has no idea about what is happening while her query is executing. Once issued, the user has to wait until the database returns the result. The response time of the query varies, depending on an array of factors such as the number of queries currently running, the system resources that are available etc. This is where online aggregation becomes very useful [13]. An online aggregation engine continuously returns estimated results with some confidence interval. The user need not wait to get a ballpark estimate of the true value of the result since the system returns estimates of the value at frequent intervals. The one useful feature is that the user can stop the query when she is satisfied with the confidence interval. Even with online aggregation the user has no control over the query execution. She cannot decide how fast or slow her query executes. A solution to this problem is to associate a “priority” or “weight” with each query. This value defines the “importance” of the query. A high value of priority could mean higher importance when compared to a low value of priority. However, this is a static value, which has to be assigned when issuing the query. This provides a coarse level of control to the user. Once issued the user still cannot perform any other operation that will affect the execution rate of the query. To provide true control over the execution
rate of the query, the user should be given the ability to change the priority of the query at runtime. The user should be able to increase or decrease the rate of execution of the query during its execution.

We propose an online aggregation system that takes query priority as a user defined input. The priority of a query dictates how fast or slow the query executes. This priority, associated with each query, can be modified at runtime. If the user wants to increase the rate of execution of a particular query, increasing the priority will bring about such an effect. This ability to change the rate of execution using query priority gives the user a larger degree of control when compared to the previous approaches.

With the conventional approach the user is forced to wait until the query returns with the result. This archaic approach can sometimes become frustrating to the user. Online aggregation answers this problem by returning estimated results at regular intervals. This works well for small to medium sized data. When the size of data becomes very large, it will take a long time for the confidence interval to reach a desired value. This again might keep the user waiting. By providing the ability to change the priority of a query at runtime, the user can change the rate of execution of the query. This ability will also keep the user engaged and provide an interactive experience, which is not possible with conventional querying approaches.
Chapter 2: Related Work

Query scheduling is not a new problem. There has been extensive research in this space, resulting in many approaches to query scheduling. Query scheduling is a very broad topic of study. Scheduling of queries is performed for varying number of reasons. Martens, Rahm, et al. [1] propose dynamic query scheduling to better utilize the system resources. It looks at factors such as CPU and IO to formulate the scheduling algorithm. This is different from what we are proposing. We are looking at user input as the factor for scheduling queries. Pang, Carey, et al. [2] also discuss query scheduling for databases. This paper works along the lines of real-time database systems. It deals with time-constrained queries. It proposes an algorithm called “Priority Adoption Query Resource Scheduling” (PAQRS). System resource the paper mainly talks about is memory, more specifically dynamic memory allocation for queries. The “priority” in the algorithm is for urgent queries. Urgency of a query is dependent on its deadline. Earlier deadline queries are given a higher priority. This differs from our approach since the priority in the above paper is system defined and the priority of a query in our system is user defined. We also do not deal directly with resource scheduling. Garofalakis, Ioannidis, et al. [3] talk about how to handle multi-dimensionality of resources that each query requires. The paper talks about Time-Shared and Space-Shared resources. Even though
these components do occur in our approach we do not directly address them. This paper is orthogonal to our research and will only supplement our effort. Bouganim, Fabret, et al. [4] talk about query fragments scheduling, which we do not deal with in our research. Gupta, Sudarshan Vishwanathan [5] discuss ideas about common sub queries across many queries and how to approach it. Agrawal, Kifer, Olston [9] talk about how to reschedule queries based on their sharable aspect. Even though we look at shareable queries, it is not a factor in our scheduling algorithm. The paper also focuses on batch type jobs such as map reduce, while we focus more on ad hoc type queries. Hoogeveen [10] focus on scheduling with multi-criteria optimization for single as well as parallel environments. Zukowski, Héman, et al. [6] is a very interesting paper that speaks about query scheduling to make better use of shared scans. Unlike elevator scan cooperative scans are not sequential. Cooperative scan decides which chuck of data is the most “interesting” to load. This paper is the closest to some of the problems we solve. We adopt a modified form of elevator scan and so do not overlap with work done in this paper. Additionally, we have another factor, priority, which brings in added complexity. Agarwal, Mozafari, et al. [20] perform bounded execution time with corresponding error rates. Bounded time execution is discussed in detailed in chapter 4.

To the best of our knowledge, our system is the first to look at query scheduling with dynamic user define reprioritization for online aggregation queries.
Chapter 3: Online Aggregation

Online aggregation is an idea put forth by Joseph M. Hellerstein, Peter J Haas and Helen J. Wang in their paper “Online Aggregation” [13]. As already mentioned an online aggregation system returns estimated results with error bounds at regular intervals. Unlike traditional database systems, where the user is kept waiting while the query executes, this approach provides continuous feedback to the user. Online aggregation is not always suitable. It is not applicable for queries that just want to retrieve information stored in a database. It cannot be used when information such as name or address is to be retrieved or inserted into a database. In our research, we do not address such queries. It is useful when performing analysis, such as “average age of users” or “total number of items sold”. If the data is very large and exact answers are not a necessity, online aggregation is a perfect match. It always informs the user about estimated values so that the user can terminate the query if she is satisfied with the estimate and the corresponding confidence interval.

3.1 Implementing an online aggregation system

In this section we will discuss how we have built the online aggregation system. Our system is written entirely in java. It is built taking query reprioritization into account. All the components are asynchronous and highly parallel. By being asynchronous it has a
very good reaction time to a change in query priority. We have a threaded execution framework that can execute a query over different data blocks in parallel. This execution engine can also stop a query in the middle of its execution. It also has the unique ability to pause a query execution. When resumed the query execution will continue from the point it was paused without any need to recalculate results.

3.1.1 Architecture

Like any other online aggregation engine, our system also returns continuous results to the user. In addition to that the user of this system has the flexibility to increase or decrease the rate at which her query is executing. If the query execution is slow and she wants to see the results at a faster rate, all she has to do is increase the priority of the query. A user of this system need not necessarily be the end user viewing the results. Another system, that feeds inputs, could be used to leverage the abilities of our system.

Figure 1 describes the architecture of the online aggregation system. It has two main components

- Master
- Slave

Master

This component is the interface for the user. It accepts queries and returns results in an online manner. The master also accepts changes in priority for a particular query. The
master has two components 1) Query Catalog 2) Aggregator. The query catalog as the name suggests stores information about each query. It stores catalog information such as number of rows processed so far and the updated result of every query. It can also store other information such as how fast or slow the user wants to see the results for her query.

![System Architecture Diagram](image)

**Figure 1: System Architecture Diagram**

This is different from the rate of execution of the query. This information tells the system about how frequent the user wants to view a change in the result. For example, let us assume that initially the user is viewing change in result every 4 seconds. As the query
progresses she might want to increase this rate, setting it to 1 second, to catch the finer changes in the estimated result.

The aggregator module is responsible for aggregating the results that are returned by the slaves. This module is important when the system has more than one slave node. If the system consists of a single slave node, then the slave node can perform all the aggregations. In case of a distributed system the data is partitioned over multiple slave nodes and needs additional aggregation at the master end.

**Slave**

The slave forms the core of the entire system. It manages the execution of all queries in the system. This component is highly parallel and has an asynchronous implementation. Every module in this component runs independently of the other modules. This is very important not only for an online aggregation system but also for very quick response time in case of a change in priority of a query. The catalog modules are for storing and retrieving information about the queries.

Query catalog stores information about each query such as number of rows processed, rate of execution and current priority of a query. This information frequently changes and is updated after every cycle of execution of the query. It has a more detailed explanation in chapter 5.

Execution catalog stores the execution information of each query. It holds information such as the number of blocks executed so far as well as startBlock and maxBlock values. The importance of startBlock and maxBlock is discussed in chapter 5.
Executor and Aggregator are the modules responsible for the actually execution of a query. These components are intricately connected. Executor manages multi-threaded execution of the query. It is asynchronous and can instantly include a newly issued query into its current execution cycle. The aggregator performs the aggregation on the data. It provisions basic aggregations like SUM, COUNT, AVG, MIN, MAX. It also has added features like lock and lock-free aggregation.

### 3.1.2 Event based VS polling

Our initial implementation used Jetty to send and receive data between master and the slaves. Jetty is a HTTP webserver. It has many features like SPDY, Web Sockets and JNDI that are very useful for a distributed application. Jetty uses a threaded model to handle concurrent requests. Each query request is handled from start to finish by a single thread. Any I/O calls, like a remote service call, is synchronous. With synchronous I/O calls, the thread is blocked and is idle till the I/O call returns, effectively blocking incoming requests.

In order to deal with idle time of the thread when it is blocked on I/O operations, jetty introduced a feature called continuations. This feature enables a connection to be suspended, which frees the thread. The thread can be reused when the connection is resumed. This feature is useful for an online aggregation system where the execution engine has to continuously return results back to user.

Continuations does not scale well with increase in number of concurrent queries. As the number of concurrent queries increased, the frequency of suspending and resuming
connections also increased, introducing additional overhead in the system. This is due to the high volume of result messages sent from the slave back to the master. Jetty did not meet the throughput requirements of our system.

To deal with this problem, we moved to an event driven system. Here the thread is not blocked on I/O calls instead it is interrupt driven. When a thread issues I/O calls, instead of waiting for the calls to return, the thread continues to process other requests, and only comes back when the I/O calls return. The event driven model scales well as the number of queries increase. We made use of an event driven approach that met our throughput requirements.

3.1.3 Calculating Estimates and Confidence Interval

As already stated, online aggregation returns estimated results with some confidence interval. To calculate the confidence interval, COUNT and VARIANCE values for each group are needed. These values are calculated in each thread of the execution engine. The variance and count are combined, from all threads, resulting in a single value of count and variance for each group in the result. The procedure to calculate combined variance is given below. The table describes each variable needed in the formula
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s^2_h$</td>
<td>variance of group $h$</td>
</tr>
<tr>
<td>$n_h$</td>
<td>number of tuples in group $h$</td>
</tr>
<tr>
<td>$n_{hi}$</td>
<td>number of tuples belonging to group $h$ from the $i^{th}$ query</td>
</tr>
<tr>
<td>$m_{hi}$</td>
<td>mean of the group $h$ from the $i^{th}$ query</td>
</tr>
<tr>
<td>$m_h$</td>
<td>mean of the group $h$ from all queries</td>
</tr>
<tr>
<td>$v_{hi}$</td>
<td>variance of the group $h$ from the $i^{th}$ query</td>
</tr>
<tr>
<td>$\bar{\sigma}{\hat{\tau}}$</td>
<td>variance of the estimator for the measure SUM</td>
</tr>
<tr>
<td>$\bar{\sigma}{\hat{\gamma}}$</td>
<td>variance of the estimator for the measure AVG</td>
</tr>
<tr>
<td>$\bar{\sigma}{\hat{\rho}}$</td>
<td>variance of the estimator for the measure COUNT</td>
</tr>
<tr>
<td>$H$</td>
<td>Total number of groups in the union of all the queries</td>
</tr>
<tr>
<td>$N$</td>
<td>Total number of tuples in the table</td>
</tr>
</tbody>
</table>

Table 1: List of notations used.

Below is the formula for combining many variances into a single variance, per group.
\[ s_h^2 = \frac{1}{n_h - 1} \left( \sum_{i=1}^{numQ} n_{hi}(m_{hi} - m_h)^2 + \sum_i (n_{hi} - 1)v_{hi} \right) \]

The derivation of the formula is discussed in DICE [21]. After combining variances the confidence interval, for each group, is calculated using the below formula.

\[ P = \frac{n_h \times N}{\sum_{h=0}^{H} n_h} \]

\[ C.I = (m_h \times P) \pm (1.96 \times s_h \times P) \]

The combining of variances has to be performed both on the slaves as well as the master. At the slave side, combining is performed across the results from different execution threads. At the master side, the combining is done on the continuous results returned by the slave.
Chapter 4: Bounded Time Execution

4.1 Why is bounded time execution important?

Bounded time execution, in context of database, is to execute a query within a specific amount of time. There are many time critical applications that need response time to be within a specified time. Some examples of such applications are control systems at airports, stock exchanges, medical applications etc. All these applications are time constrained. The response time always has to be within the specific time limit. Varying response time could be the difference between winning and losing millions in case of stock exchange and much larger consequences in case of medical applications. It is imperative that query response is retained within the specific time.

Bounded time execution does not necessarily entail real time responses, even though this is often the case. Bounded time can also signify predictable response time. To design an execution engine with predictable query response time is a non-trivial task.

There are different ways to handle time bound execution, one of which is to use sampling. Instead of processing the entire data a sample of data is processed, returning estimated results with error bounds. Such an approach is shown in BlinkDB [20], where samples of various resolutions are built offline. Depending on the response time and the
error bounds, the query is executed on the appropriate samples. DICE [21] is another system that also uses sampling to perform time bound execution of speculative queries.

4.2 Actor Model

DICE uses the actor model for the architectural design. As mentioned in the previous chapter the actor model is more scalable and built for highly concurrent applications.

In an application using the actor model the “actor” forms the building block of the application. In the actor model, information flows in the form of messages. Typically an actor receives a message and performs appropriate computation in response to that message. When the computation is complete the actor can either reply with another message or simply do nothing.

The actor model adheres to the principle that everything is an actor. In this aspect the actor model resembles object oriented programming model where everything is an object, but is different in terms of execution flow. Object model typically has a sequential and synchronous implementation, while the actor model is typically built for concurrent and asynchronous applications.

An actor in response to a message it receives can perform any of the below tasks:

- Perform some computation based on the message.
- Forward the message to other actors.
- Can create a new actor to process each message.
All of the above mentioned tasks could be done sequentially and synchronously or asynchronously and in parallel. It is the parallel and asynchronous execution that makes the actor model such a powerful concept.

The actor model being asynchronous and concurrent has no global actor to coordinate between the various actors in the system.

Figure 2: Mapreduce using actor model
Figure 2 shows how a simple mapreduce program can be implemented using the actor model. The major difference is that here each map actor is a single thread compared to a java virtual machine in a typical mapreduce implementation. Also, with the actor model reducers needs to be split in to reduce and aggregator (Agg) actors. The reduce actors perform the task of shuffle and sort. The aggregation of the values are done using the aggregator actor. Here all the data is transferred in the form of messages. The <Key,Value> pair emitted from the map actor can be sent to the reduce actor in the form of a message. The correct reduce actor can be chosen using an appropriate hash function. A good hash function can take care of skew in data as well.

4.3 DICE Description

DICE is a distributed and interactive cube exploration system. DICE consists of three main components.

Faceted Cube Exploration – The systems introduces a novel session-oriented cube exploration model. It introduced a facet-based cube exploration model. It models the transitions as traversals from one facet to the next. The permitted traversals are parent traversal, child traversal, sibling traversal and pivot traversal.

Distributed Execution Engine – DICE has a distributed execution engine that employs a hierarchical master-slave approach, such that all queries are issues to the master and responded to by the master. The execution engine delivers sub-second latencies for query execution over a fixed user-defined sample of data. The execution engine supports parallel execution of queries over large amounts of partitioned data. The execution engine
is highly scalable and has been tested up to 50 nodes while retaining interactive latencies in query execution.

*Speculative and Bounded Time Execution* – It also introduces the concept of speculative execution. Speculative queries, which are based on the current executing query, are executed in anticipation of the user’s next cube traversal. Using the current facet the system enumerates all the possible traversals, which are then prioritized and executed within a fixed bounded time of 5000ms. The results of these speculative queries are cached to decrease the execution time of the next traversal in case of successful speculation.

### 4.4 Akka

Both DICE and our online aggregation system make use of a framework called Akka. Akka is an open source implementation of the actor model. It has support for both synchronous and asynchronous implementation. Having a non-blocking IO framework makes it an ideal candidate for implementing a distributed event driven system. Once all components of the architecture are implemented scaling out, with akka, is as simple as installing the components on a new node and connecting it to the system. Akka also has fault tolerance built into the framework, which alleviates the need for the application to implement fault tolerance.
4.4.1 Components

Akka has many components. We will discuss two components that are essential to our implementation of bounded time execution.

4.4.1.1 Mailbox

Every actor is associated with exactly one mailbox. All the messages for an actor are sent to its mailbox. Mailbox is the element that connects the receiving actor with the message sender. The actor processes messages from its mailbox in order of their arrival. Internally the mailbox is implemented as a queue. All the different senders enqueue their respective messages in the mailbox and the receiving actor dequeues the messages and processes them. An important point to note here is that message order is maintained per sender, but not across senders. What this means is that, the messages from different senders will not have an ordering between them.

![Actor Diagram](image)

Figure 3: Actor
The default mailbox follows first in first out (FIFO) approach. This is the preferred implementation. There are other mailbox implementations such as priority mailbox. Here, dequeue operation is based on the priority of the messages in the queue. This will be useful in systems that support priority, such as ours, where there are multiple senders to a single receiving actor. Even though there is no relative ordering between the multiple senders, the priority of the message will ensure an ordering between them after the message is enqueued.

Another implementation is the shared mailbox. Unlike the previous approach, a single mailbox is shared across all the actors. This mailbox supports concurrent operations. This implementation is useful when load balancing across all the actors is required. Figure 4 describes this type of implementation.
The mailbox implementation might look like a performance bottleneck, but it is not the case. The entire logic for obtaining the next message is in the dequeue operation. An entire scan of the mailbox is not needed.

4.4.1.2 Router

A router is an actor whose only task is to route the messages to other actors. This forms a very critical part of the system. This is the component where different routing algorithms can be implemented. We will discuss two routing algorithms that play a role in bounded time execution implementation.
**RoundRobinRouter**

This is the default router. As the name suggests it performs round-robin routing. It routes messages to different actors in a round robin manner. Whether this is a good load balancing approach is debatable. If each message takes the same amount of time to process then this router divides work equally across the actors. On the other hand if the processing time of each message is not equal, this approach is not optimal.

**SmallestMailboxRouter**

The problem with round robin approach is when each message takes varying time to process and the time is not known prior to processing the message. Messages that take long to process might be routed to the first actor and messages that take small amount of time to process might be routed to the second actor. The second actor will complete all its messages and remain idle while the first actor is still processing its messages. To overcome this problem, the smallest mailbox router is used. This router routes messages to actors that have finished processing their respective previous message and are waiting for the next message. If the previous example is considered, both the actors will be occupied and the resources are completely used. This is the router used by DICE.

**4.5 Bounded Time Implementation**

Akka has no implementation that supports bounded time execution. Bounded time execution needs support for time out. DICE has a hard limit of 5000ms for the speculated query execution, which implies that after 5000ms all the currently executing queries have
to be stopped and all the pending queries in the mailbox have to be invalidated. None of the components of akka support such a requirement, which meant a custom implementation is required.

Out of the two requirements, stopping a currently executing query is simpler. This is achieved by interrupting the thread that is executing the query. Interrupting the thread at the application level is not sufficient. There is a separate thread inside the database context that is still executing. It has to be made sure that the interrupt is propagated to the executing thread in the database as well.

The solution to the second requirement is more complicated. One major problem with time out in a distributed system is clock synchronization. The 5000ms time limit should start counting down when the master sends the speculated queries to the slave. After 5000ms the queries have to be invalidated at the slave node, which means the slave node should be aware of the exact time the count down started at the master node. This poses a time synchronization problem.

One solution to synchronize the clocks across the cluster is to use Network Time Protocol (NTP). Using NTP, time across all the nodes in the cluster can be synchronized. However, this is not a viable solution. Dedicated NTP servers are required to perform clock synchronization. Scaling out the cluster will now have added overhead of clock synchronization. The biggest problem with this approach is that it still might not guarantee synchronization at the millisecond level. This is a very crucial limitation and given that DICE is dealing at millisecond level, the problem becomes more complicated.
4.5.1 AutoExpireMailbox

We implemented a new mailbox called *AutoExpireMailbox*. This mailbox is implemented considering auto expiry of messages after 5000ms. The implementation does not assume any form of clock synchronization between the nodes in the cluster. The algorithm for timeout is implemented at the slave nodes.

As stated above, the problem is to determine when the count down starts. To solve this problem we implement a new mailbox in the Akka system, *AutoExpireMailbox*. The mailbox only maintains queries that are within 5000ms of the count down start time.

*AutoExpireMailbox* uses an unbounded concurrent queue to hold query messages. It also makes use of a variable, *startTime*, which stores the timestamp of the first query message that is enqueued. This indicates the start of the count down. During every dequeue operation, we check the time difference between the time dequeue operation was invoked and *startTime*. If the difference is greater than 5000ms, we flush the query message queue, reset *startTime* value and return null. If the difference is less than 5000ms, we return the head of the queue and remove the query message from the queue. This doesn’t not add overhead since the check is performed only on a dequeue operation and each check has O(1) runtime.
4.5.1.1 Algorithm

Below is the pseudo code for the enqueue and dequeue operation in *AutoExpireMailbox*.

```python
1 enqueue(queryMessage)
2     if startTime == -1
3         startTime = getCurrentTime()
4     queue.add(queryMessage)
```

Algorithm: Enqueue
```java
def dequeue()
    if queue is empty
        return null
    time = getCurrentTime()
    diff = time - startTime
    if diff >= 5000
        queue.flush()
        startTime = -1
        return null
    else
        return queue.pop()
```

Algorithm: Dequeue
Chapter 5: Query Reprioritization

5.1 Why is query prioritization important?

In any database, which runs queries in parallel, not all queries have the same importance. There are always few queries, which are more important than other queries. Without prioritization a database cannot differentiate such queries. With current systems we can assign a priority to a query at the time of issuing the query. This is static priority. The priority remains the same throughout the life span of the query execution. This definitely helps when we know the exact query and its importance. In case of exploratory analysis the user might not know the exact query that gives the results she desires. In such cases the user will not know the importance of a specific query beforehand. She might find out as she is viewing estimated results in an online fashion. In a system that supports static priority, she cannot change the priority of the query after issuing it or while viewing the estimated results in an online manner.

Online aggregation does not guarantee faster execution of the query. It only provides the ability for the user to view the estimated value of the result at regular intervals. A query over large data is going to take relatively long time to attain low error bounds. Queries that run in the order of several seconds are not necessarily at an interactive level. Research such as Nandi, Jagadish [23] and Kamat, Jayachandran, et al. [21] has shown
how important it is even for queries over large data to be interactive. If the rate of execution of the query can be changed at runtime, by changing the using user-defined priority, the execution can be done at a more interactive level when compared to a simple online aggregation system.

Another use case for dynamic priority is ad hoc queries. These queries are when a ballpark approximation of the true result is required. Depending on the estimate result that is returned, the user might want to speed up or slow down the execution rate of the query. In an online aggregation system, results are returned to the user only at regular intervals. The system does not expose anything else. The user has to be contented with the rate at which the confidence interval changes. Even if she desires a faster decrease in confidence interval, it cannot be achieved with current online aggregation system. Current online aggregation engines do not address the above-mentioned problems. A solution to these problems is to allow the user to change the priority of a query at runtime. Doing this will provide the user with a fine-grained control over the execution of the query. The user can then increase, decrease, pause or stop the execution as she pleases.
Figure 6: Demonstrating the effect of change in priority

Figure 6 describes the idea behind runtime reprioritization. The graph plots time in milliseconds on the X-axis and number of rows processed on the Y-axis. For simplicity we use only two queries, Query-1 and Query-2, issued by different users. Both these queries are run on our online aggregation engine. They start returning estimated results as soon as they are issued. Initially, both the queries have equal priority, i.e. at the time of issuing both the queries have the same priority value. We observe that during the initial stages of execution, both these queries process equal number of rows over time, right up to 4000 milliseconds. After viewing the estimated results for a couple of seconds, the user of Query-1 increases the priority of Query-1. Immediately the system takes the change in priority into account and starts processing Query-1 at a faster rate compared to Query-2. This is visible from the graph. The line depicting rows processed for Query-1 starts rising
and completes at the 16000-millisecond mark. At this point the system is running only Query-2 and hence there is a rise in the rows processed for Query-2 until it completes. The line at the top shows the total number of rows processed over time across all queries. The remaining line, passing in between Query-1 and Query-2, illustrates what happens when there is static prioritization. In this case, no change in priority is allowed at runtime, resulting in both queries running at the same rate and completing at the 26000-millisecond mark. The reason why all the query lines flattened out at the end is because the queries have completed their execution between the two time intervals.

5.2 Varying Rate of Execution

In our system the rate of execution of queries can be changed at runtime. This is made possible by providing the user the ability to associate a query with a dynamic priority. The priority can be changed during the execution of the query. This value decides the rate of execution of a query. The number of parallel queries that the system can handle is decided on the resources available. There is no fixed universal number. The system does not restrict on the number of parallel queries. Like any other database system, if the number of queries exceeds what the system can handle, performance of the system decreases.

The rate of execution of a query can be modeled in different ways. One way is time slicing. This is the traditional approach taking in most operating systems today. At the kernel level the each process gets a certain time to execute before it swapped with another process. This happens in a round-robin manner. Our system uses the number of
rows processed as a factor to decide the rate of execution. More number of rows processed equates to higher execution rate. Each query is executed for a certain number of rows before being swapped by another query. Intermediate query result is sent back to the user each time the query is swapped. The current priority of the query decides the number of rows it processes each time, before it is swapped. A query with higher priority processes more number of rows before it is swapped as compared to a query with lower priority. The value of the priority cannot be taken as an absolute term for deciding the number of rows to process. It has to be normalized before execution. This is important since a query with a very high priority will execute for a very large number of rows. If this query is IO bound it will be blocked for most its execution time. This inherently blocks the other queries running and an online aggregation engine cannot have long blocked periods. For example – If Query-1 needs 4 seconds to execute 1 million tuples, all other queries are blocked for 4 seconds. Therefore, it becomes essential that we break the execution into chunks and normalize the priority. We describe how we normalize the priorities.

\[ P_{TOTAL} = \sum_{i=1}^{n} P_i \]

\[ P_{wi} = \frac{p_i}{P_{TOTAL}} \]
First we sum priorities \( P_{TOTAL} \) of all the queries currently executing. The weighted priority \( P_{w1} \) is then calculated. This weighted priority decides how many rows a query processes before being swapped.

As mentioned before, we define the basic unit of storage as a block. Each block consists of a fixed number of rows. Queries are executed at blocks level granularity. Research [19] has shown that processing data in blocks instead of a single row at a time, results in better performance due to pipelining and processor cache.

We execute queries in a round robin manner. A calculated number of blocks are processed per query before being swapped. A cycle is complete when all the queries have executed additional blocks of data. All queries are always executed in one cycle. In order to reduce blocking delays we introduce a parameter, \( BLOCKS\_PER\_CYCLE \). The “BLOCKS” in the parameter name refers to blocks of data and not blocking delay. The value of this parameter indicates the number of blocks to execute per cycle. For example if \( BLOCKS\_PER\_CYCLE \) is set to 10000, 10000 blocks of data is processed across all the queries per cycle. If there are two queries running with priorities of 25 and 5 respectively, Query-1 processes 8400 blocks per cycle and Query-2 processes 1600 blocks per cycle.

\[
P_{TOTAL} = 25 + 5 = 30
\]
\[
P_{w1} = \frac{25}{30} \approx 84 \, (84\%) 
\]
\[
P_{w2} = \frac{5}{30} \approx 16 \, (16\%)
\]
Query-1 processes 84 percent of 10000 blocks, 8400 blocks and Query-2 processes 16 percent of 10000 blocks, 1600 blocks. This is how the rate of execution is decided.

The $BLOCKS\_PER\_CYCLE$ parameter is deeper implications. It, along with the number of rows per block, decides the frequency at which the user is provided with a new estimated result. This is different from the rate of execution. This signifies how frequent the user views a change in estimated result. If the $BLOCKS\_PER\_CYCLE$ has a low value, the result is sent back to the user at a high frequency. On the other hand if the value is very high the result is sent back at a very low frequency, longer intervals.

$BLOCKS\_PER\_CYCLE$ also affects the total number of context switches in the system. Context switching in our engine is switching between query executions. This entails procedures such as saving the current query’s execution info, updating query catalog, sending the result back etc. However, sending the result back does not affect the time, since it is done asynchronously in a separate thread, but all other procedures add to the overhead. When many queries are simultaneously executing, context switching cannot be avoided. The goal is to find a balance between context switching overhead and query execution progress.

\[
Context \ Switches_{Cycle} = \ Number \ Of \ Queries
\]
\[
Number \ of \ Cycles \ = \ \frac{Total \ Number \ Of \ Blocks}{BLOCKS\_PER\_CYCLE}
\]
\[
Context \ Switches_{TOTAL} \ = \ Context \ Switches_{Cycle} * Number \ of \ Cycles
\]
\[
Time_{Effective} \ = \ Time_{Process\ Blocks} - Time_{Context \ Switches_{TOTAL}}
\]
The above equations are self-explanatory. We have to increase $T_{effective}$ by reducing $T_{Context\ switches_{TOTAL}}$. This is achieved by decreasing the number of cycles, which is in turn accomplished by having higher value of $BLOCKS\_PER\_CYCLE$. The consequences of having a high value for $BLOCKS\_PER\_CYCLE$ are already discussed.

Finding a balance between context switching overhead and query execution progress is not a straightforward approach. It depends on lots of factors like resources available, query type (IO bound or computation bound), nature of the underlying storage etc and hence varies across different usage of the system. Looking at all the above aspects, choosing a good value for $BLOCKS\_PER\_CYCLE$ is non trivial and hence the parameter is made configurable.

5.3 Query Starvation

In any scheduling algorithm starvation is always a problem. Starvation is when a process does not get sufficient resources to execute. This happens when the scheduling algorithm always has high priority processes coming in. In our context starvation equates to query starvation, when a particular query does not execute for a long time. In an online aggregation system, which provides the ability of dynamic query reprioritization, query starvation is a real possibility. In our online aggregation engine, query starvation equates to a query being blocked for a long time because of other high priority queries in the system. By definition of online aggregation this is not allowed. A query should always
return estimated results at regular intervals without being blocked for a long period of time.

A common solution to process starvation is employing a technique called Aging. Each process has an aging factor. This factor increases every time a process is denied resources when requested. As the aging factor increases the priority of the process is also increased. Some systems also have an upper bound on the aging factor, after which the process is forcefully given resources.

Incorporating an Aging factor is one approach. Instead, our execution engine avoids starvation altogether. This is achieved by simply following a round robin approach in selecting the next query for execution. The priority of the query is not considered when choosing the next query. The priority rather, as explained before, decides how many blocks of data are processed in a cycle.

For all the experiments we use machines having 4 GB memory, intel xeon quad core processors and 720 GB of disk space. The cluster has a 1 Gbps switch. Data is row-oriented with 5 million rows and 6 columns, consisting of strings, integers and floating-point data types.
Figure 7: Illustrating absence of Query Starvation

Figure 7 demonstrates how we avoid query starvation. Query-1, Query-3, Query-4 are executed with a priority of 10 and Query-2 are executed with a priority of 1. Query-1 and Query-2 are initially issued. Query-3 is issued after 2 seconds and Query-4 is issued after 4 seconds. Even though Query-3 and Query-4 are issued at a later time with higher priority, Query-2 continues to execute, at a much lower rate, without being starved. The system is capable of handling many more queries, but only 4 are used in this experiment for the sake of simplicity. The various bends in the lines are because of queries being issued and queries completing their execution.
5.4 Hysteresis

What happens to the system if the user accidently or intentionally misuses the ability to change priority of a query at runtime? As already mentioned, another system can leverage the abilities of our execution engine. For example, consider we have gestureGB [22] feeding queries and priorities based on the user’s gesture. When the user of gestureGB performs a burst of rapid gestures, causing the priority of the query to change continuously, the execution engine will have problems executing other queries currently running in the system. Even though the system remains accurate in the execution of individual queries, it introduces unpredictability in the rate of execution of queries with respect to its priority. Hysteresis has to be dealt with at the execution level. There are two approaches that we discuss

- Hard Limit on the number of changes.
- Introducing a hysteresis factor into query priority.

**Hard Limit**

This approach sets a hard limit on the number of changes in priority for every query. This limit can be a universal constant for the entire system, which means each query gets the same limit. This could be unfavorable since the system will consist of users with different access levels. A privileged user will end up having the same authority as an unprivileged user. A way to deal with this would be to make it a user specific constant. A trusted or a privileged user is allowed more number of changes per query while an untrusted user is
allowed fewer changes per query. With this approach the limit has to be monitored per user basis by the administrator. Any changes to the limit for a user has to be done manually.

**Introducing a Hysteresis factor**

Hysteresis is still a possibility with the hard limit approach. The system is still affected if a trusted user accidently changes the priority of a query at a high frequency. In order to avoid this, variability needs to be introduced into the query priority. This variability should be a function of the number of changes in priority for each query. Let's call this hysteresis factor. The hysteresis factor should have a positive effect for the initial few changes in priority. Having a negative impact initially on the change in priority of a query contradicts the concept of changing priority at runtime. As the number of changes increases, beyond a threshold, the hysteresis factor should have a negative effect on the priority of the query. It should act like a penalty. The nature of this function suggests a Gaussian like distribution. The below formula is used to calculate the priority with hysteresis.

\[
P_H = P_u \times \frac{1}{e^{(2)(\text{Number of changes} - \text{CONSTANT})}}
\]

\[
P_H = \text{Priority with Hysteresis}.
\]

\[
P_u = \text{User defined priority}.
\]
The \emph{CONSTANT} variable decides how soon the hysteresis factor has a negative effect on the query priority. Higher value of the constant results in more tolerance, which means that the system will allow more changes in priority before having a negative effect on the priority. A low value for the constants results in less tolerance, allowing fewer changes before it has a negative effect. The value of \emph{CONSTANT} is configurable. Like the first approach it can be a universal constant or a user specific constant. In the later case, a privileged user is allowed higher tolerance, which denotes more changes in priority before being penalized.

Figure 8 shows how the hysteresis factor behaves for different values of \emph{CONSTANT}. The X-axis is the user-defined priority and the Y-axis is the priority with hysteresis factor. Each curve is for a different value of \emph{CONSTANT}. At a value of 20, the blue curve, the system allows more changes in a positive effect before having a negative effect on the priority. As the value of the \emph{CONSTANT} decreases so does the tolerance in the system. At a very high number of changes in the priority, the priority with hysteresis factor simply reduces to 0, which is the lowest priority.
5.5 Control Flow

Figure 9 describes the flow in our execution engine on the slave nodes. The execution engine first requests for a query from the query catalog. As mentioned before the queries are executed in round robin fashion. The weighted priority of the query is calculated using the formulae mentioned in section 5.2. The calculation is performed on the latest priority of the queries running in the system. An important point to note is that, the execution thread is separate from the thread that updates the priority of the query. This is needed for fast response time in case of change in query priority. With separate threads
and asynchronous model of execution, any change in query priority is reflected instantaneously.

Once the weighted priority is calculated, we determine the number of blocks to process for the current query in this cycle. Next, the engine acquires the execution information of

Figure 9: Control flow in the execution engine
the query from the execution catalog. This information consists of fields such as the number of blocks executed, next block that has to be executed etc. The execution information consists of more fields, which are covered in the following sections. Once the information is obtained, the query is executed in parallel. Each executing thread is assigned one block to be processed. After all threads complete their execution, the result is calculated and sent back to the master. The master will perform post aggregation and return the estimated result back to the user. On the slave side, the execution engine updates the execution catalog and the query catalog. The query catalog contains information needed to collect statistics about the query such as rate of execution, number of rows processed so far etc. The same procedure is followed for the next query.

5.5.1 Algorithm

Below are pseudo codes for the execution engine.

```
calculateWeightedPriority(P)
    for q in queries
        p = getQueryPriority(q)
        total = total + p
    Pw = P / total
    return Pw
```

Algorithm: Calculation of weighted priority.
Algorithm: The main execution.

5.6 Elevator Scan

In the previous sections we saw how the execution engine implements query reprioritization. We did not discuss about tables the queries were run on. All the queries were run on the same table, which means there was data overlap over all the queries. We observe that Query-2 and Query-3 ran over the same table, there was data overlap between them. Both the queries processed the same data to compute their results. When the entire table cannot fit in memory, pages are requested from disk. This is traditionally implemented in databases by using an LRU cache. The pages are cached after reading from disk. This constitutes an IO operation, which are expensive. The disadvantage of
the LRU approach is that if one query starts at a different time, the cached pages will already be swapped out before they can be reused.

Figure 10: Varying Priority with independent scans

In this case, Query-2 and Query-3 are making separate IO calls even though they operate over the same data. Is there a way to exploit this overlap? Is there a way to minimize the IO cost?

There are been work done in this area. Unterbrunner, Giannikis, et al. [7] talk about response time predictability using elevator scans. Wang, Das Sarma, et al. [8] and Agrawal, Kifer, Olston. [9] are recent work done in this area. They both examine how to
use shared scans to achieve faster processing for batch-oriented system such as MapReduce. This model can easily extended to an ad-hoc type of system, such as ours.

Elevator scan is simply a sequential scan, of the table, that is always running. Incoming queries attach themselves to the table scanner, which is scanning some part of the table, and begin executing. The queries detach from the scanner when they are done processing the entire table. This is a cache efficient approach since all queries are executed on the data block paged in from disk. Once processed, that block of data will never be paged in for those set of queries. This results in constant IO cost, for reading data, irrespective of the number of queries executing concurrently. Figure 11 illustrates how an elevator scan works. The table scanner is currently scanning from block A to block B. All queries that have not finished processing these blocks operate on the data.

![Elevator scan diagram](image)

Figure 11: Elevator scan
5.6.1 Query Priority With Elevator Scan

Elevator scan introduces the ability to share table scans between many queries. This implies that queries that share scans have to run simultaneously. Previously, with independent scans, each query processed different number of blocks per cycle of execution depending on its weighted priority. With shared scans this presents some problems. If each query that is sharing the scan has to process different number of blocks per cycle, then maintaining and calculating the different blocks processed for each query in the shared scan becomes a tedious process. To avoid this overhead, we bring in table priorities. In this technique, priorities are calculated per table. The effective priority of the table decides the number of blocks that will be processed for a given cycle. Any change in priority of a query affects the table priority as well. Below are the formulae for calculating the weighted priority of a table.

\[ P_{Table_t} = \sum_{i=0}^{m} P_i \ \forall \ Query \ i \ on \ Table \ t \]

\[ P_{Table_{Total}} = \sum_{t=0}^{T} P_{Table_t} \]
\[ P_{Table_{wt}} = \frac{P_{Table_t}}{P_{Table_{Total}}} \]

5.6.2 Algorithm

With shared scan there has to be different algorithm for execution. The algorithm for shared scan is given below. All operations are done at a table level. Like the previous approach, the tables are chosen in a round robin fashion. Once the table is decided, the execution engine calculates the weighted priority of the table. This is done using the formulae mentioned above. It can be argued that summing the query priorities for a table might not be an ideal method to calculate a table’s effective priority. For instance, let us consider a table, Table-1, having a large number of low priority queries and another table, Table-2, having a single high priority query. In a situation where the summation of the query priorities of Table-1 is greater than that of Table-2, Table-1 will result in having higher weighted priority. This might not be the desired effect with respect to the high priority query on Table-2, but considering the system as a whole, assigning higher effective priority to Table-1 would prove to be more beneficial. Another option is to use the mean of the query priorities, per table, instead of sum. The right approach depends on how the system is used and hence we have implemented this as a plug-and-play feature. A custom implementation for calculating the effective priority of a table can be used.
Algorithm: The main execution using shared scan

```java
execute()
    while(true)
        table = getNextTable()
        if table == NULL
            continue
        info = getTableInfo(table)
        Pweighted = calculateWeightedPriority(table)
        numBlocks = getNumBlocks(Pweighted)
        blocks = getTableData(table, numBlocks)
        queries = getQueriesForTable(table)
        for block in blocks
            executeQueriesInParallel(queries, block)
        updateTableCatalog(table, info)
        for q in queries
            if q is not complete
                putQuery(table, q)
        if table has queries
            putTable(table)
```

Algorithm: Calculating the weighted priority for shared scans

```java
calculateWeightedPriority(currentTable)
    for t in tables
        queries = getQueriesForTable(t)
        for q in queries
            p = getQueryPriority(q)
            tableTotal = tableTotal + p
        if t == currentTable
            currentTableTotal = tableTotal
        allTablesTotal = allTablesTotal + tableTotal
    Pweighted = currentTableTotal/allTablesTotal
    return Pweighted
```
5.6.3 Parallel Execution for Elevator Scan

In an independent scan, executing a query in parallel is a trivial task. Each block of data can be processed in a separate thread. If there are many blocks for a query to execute, a thread pool can be utilized to limit the number of threads created.

Figure 12 describes the idea behind parallel execution of a query. For each cycle, the number of blocks to process is calculated. The blocks of data are requested form the underlying datastore. In case an independent scan only the current query is passed to each executor thread and for shared scan all the queries for the particular table are passed to each executor thread.

Figure 12: Parallel Execution of query
Parallel execution for shared scan is a non-trivial task. Since each query can start and end at different blocks, extra care has to be taken during execution. The execution engine has to make sure that a block is processed only once for each query. Since the number of blocks to process per cycle, for any query, can change at runtime, we might wind up processing the same block twice, for a query. For example, let us assume a query starts at block 5000, and it has to end at block 4999. If the blocks per cycle for the table was 10, we would process 10 blocks in parallel. This would mean blocks 5000 - 5009, 5010 - 5019 and so on till blocks 4990 - 4999 are processed in parallel. With the ability to change the priority of a query, let us assume the blocks per cycle changed to 20. With this change, the execution engine could have to execute blocks 4990 - 5009 in parallel, but the query has already processed blocks 5000 – 5009 at the start. This condition has to be avoided.

An easy way to deal with this is to associate a query with all the remaining blocks that have to be executed for it to complete. Even though this solves the problem, it comes at a cost of high memory consumption. The memory consumption increases linearly as the number of concurrent queries increases. We develop a naïve algorithm using only two variables to identify if a block is already processed or not.
Our algorithm uses only two variables per query, \textit{startBlock} and \textit{maxBlock}. The \textit{startBlock} variable holds the starting block number of the query. This value can be different for different queries on the same table, depending on when the query is issued with respect portion of the table that is being scanned. The \textit{maxBlock} variable stores the highest block number processed so far. It is updated every time the execution of the query for each cycle is complete. The \textit{maxBlock} always lies between the \textit{startBlock} and the total number of blocks for the table. Only blocks between $0 - \textit{startBlock}$ and between $\textit{maxBlock} - \text{total block count}$, are processed in each cycle. In the previous example, \textit{startBlock} will have value 5000 and \textit{maxBlock} will have the value of 10000. When the execution engine executes blocks 4990 – 5009, only block 4990 – 4999 are executed, since other blocks, 5000 – 5009, do not meet the condition. Figure 13 describes the technique. The portion in red (between \textit{startBlock} and \textit{maxBlock}), are the already executed blocks of the table. The remaining portion, in blue, is the blocks yet to be executed. The algorithm is given below.

![Figure 13: Shared Scan Optimization](image)
Algorithm: Parallel execution method for shared scans

After applying all the above technique we can see an increase in the over performance of the system. Figure 14 describes the benefit of using shared scans. The same experiment, as described in Figure 10, is run again, this time with shared scans. As before, Query-1 is run on Table-1 while Query-2 and Query-3 is run on Table-2.

![Varying priority using Shared Scan](image)

Figure 14: Varying Priority with shared scans
This time we see that when the priority of Query-2 is increased, because of shared scan, the rate of execution of Query-3 also increases. We also observe that all three queries complete sooner when compared to the previous approach.

5.7 Lock-Free Aggregation

An aspect of the system that has not been mentioned so far is aggregation. Our system supports basic aggregations such as SUM, COUNT and AVERAGE. The execution engine makes use of hash aggregate to achieve it. This is implemented in java by using a HashMap. If the execution in single threaded, there are no locks. The downside of this is that the execution is slow. One of the approaches to improve performance is to have parallel executions. This is already discussed above.

When using threads to execute a query in parallel, the hash map that stores the aggregates has to be shared across numerous threads. This introduces a point of synchronization. If two threads modify the map simultaneously, the outcome can be wrong, inconsistent and unpredictable. In order to obtain the correct result, one thread has to lock the map when modifying any value, resulting in the second thread being blocked until the lock on the map is released. There are various data structures, such as concurrent maps, that have optimization to improve performance. Instead of locking the entire data structure, only the value that is accessed is locked. The optimization permits locking part of the data structure instead of the entire data structure. Consequently, two threads can simultaneously change values in the map as long as they access different keys. The
problem with such data structures is that, writes are blocking but reads are non-blocking. This gives rise to read-modified-write problems. This is the exact approach hash aggregates take. It reads a value, modifies it based on the value read and writes the new value back to the map. In between the read and the write operations, if another thread reads the old value, it results in a dirty read. Hence, we cannot use such optimized data structure and have to lock on the entire map.

**Locked Aggregation**

![Locked Aggregation Diagram]

**Lock Free Aggregation**

![Lock Free Aggregation Diagram]

Figure 15: Lock and Lock-free approach

We observe that the synchronization (lock) time is high, relative to the total time of execution. This time increases as the number of threads increase. In case of shared scans,
an obvious optimization is to shuffle the query order in different threads. This results in different threads executing different queries at the same time, which reduced the time a thread is blocked, waiting to acquire the lock. This helps performance to some extent, but the synchronization time is still considerably high.

Another approach, in case of shared scans, is to have individual maps per thread. This idea is derived of MVCC, Multi-version concurrency control. Each thread operates on its own aggregation map, one per query. With this approach there is no locking necessary. An additional step that is however required is to perform another aggregation, on the maps from different threads. The additional aggregation is not expensive and is observed to always be in the order of tens of milliseconds for each query in a cycle.

![Locked vs Lockfree Parallel Execution time](image)

Figure 16: Lock and Lock-free block execution time
In Figure 16 we look at a more detailed experiment. It shows the pure aggregation time needed for lock and lock-free aggregations. Each point in the graph signifies the time needed to aggregate 20 blocks of data. Each block has 10000 rows and there are 20 threads running in parallel. The X-axis is the number of blocks and the Y-axis is the time in milliseconds. The experiment is run with 8 queries, all having SUM as measure, each issued 5 seconds after the previous query. In both lock and lock-free aggregation, we see that the general trend, in aggregation time for 20 blocks of data, is to first increase and then decrease. The initial increase is because of issuing of new queries. It is the highest in the middle. It then decreases as queries complete their execution. We observe that lock-free approach always takes less time to perform the aggregation compared to the locked approach. The high variability in the time taken with the locked approached is because of the locking overhead. Most threads are blocked while a single thread holds the lock. The variability in the lock-free approach is discussed later.

Figure 17 shows the total query execution time for both lock and lock-free aggregation. As expected the lock-free aggregation is faster compared to the locked approach. On closer inspection, we see that it is only 2.25 seconds faster. Given that the time needed for aggregation in the lock-free approach was considerably smaller, the difference in total execution time is marginal. The lock-free approach is losing time in another part of the execution.
Figure 17: Lock and Lock-free total execution time

Figure 18 shows a detailed breakdown of the execution. We observe the time needed for aggregation is smaller for lock-free compared to locked approach. The main reason why the two approaches report very similar total execution time is because of the difference in garbage collection (GC) time. The garbage collection time signifies the time the application is paused, in order to reclaim memory. We notice that the GC time for lock-free approach is nearly double that of locked approach.
This is largely attributed to the individual maps, refer Figure 15, all the different threads create. In the locked approach there is a single map, per query, that is shared among all the threads. In the lock-free approach each thread creates a new map for every query. At the end of the final aggregation step, all these maps are dereferenced, which means that the garbage collector will claim all these to create some free space. In order to deal with memory fragmentation, the garbage collector pauses the application, moves all the referenced objects to a side of the heap, reassigns the pointer and frees the remaining contiguous memory space.
5.8 Storage

Work mentioned in the above sections was run on data stored in row-oriented form. Row oriented storage is the traditional storage format. In the recent past column oriented storage have emerged to be more useful under certain usage. For example column oriented are the preferred format in analytics. Column oriented storage are preferred because by principled, less data has to be read from disk. During query execution only the required columns are read from disk, unlike row store where all the columns of a row are read from disk. This provides a huge performance improve with wide tables. Column stores also have added benefits when it comes to compression. Since values in a column tend to be similar, they provide better compression ratio compared to row stores.

It is evident that column oriented storage format will benefit online aggregation system. Each query will read only the required columns during execution. The idea of shared scans for column storage is not intuitive. There are problems such as different rate of scans for different columns, choosing which column to scan etc. It is still worth performing experiments to compare row storage, with the different approaches mentioned, and column storage.

We perform an experiment for comparing execution time for row and column storage format. For all the experiments we use machines having 4 GB memory, intel xeon quad core processors and 720 GB of disk space. The cluster has a 1 Gbps switch. Data comprises of 5 million rows and 6 columns, consisting of strings, integers and floating-point data types. The data used is identical in both the formats. We plot the time it takes to complete executing the queries for both the formats.
The X-axis shows the number of queries and the Y-axis shows time in milliseconds. The case of row format, both the independent and the shared scan for considered. As the graphs reveals row storage with independent scans shows the least performance. The increase in time can be attributed to increased data read from disk as the number of queries increase. In comparing, row oriented shared scans and column store scale well with the number of queries. When there is less number of queries, column store performances better than shared scan, since data read from disk for column format is significantly less. As the number of queries increase the shared scans and column format lines cross each other. With increase in queries, column format starts reading more data.
from disk, whereas shared scan read the same amount of data from disk. The lines cross over when the data read by column format is more than the data read by the shared scan.
Conclusion And Future Work

We introduced a system that allows for a dynamic user defined reprioritization of online aggregation queries. The need for an online aggregation system is increasing as the size of data to process is increasing. With this rise, the demand for interactive querying is also growing. We introduced a distributed execution engines that can consider user input, at runtime, and show instant change in query execution will be a valuable addition to the field of interactive querying. We make use of online aggregation and shared scans to implement an optimized priority aware system that can execute numerous queries in parallel to provide an interactive experience.

With the capabilities that the system has shown, there are several avenues of future work. Once such possibility is to integrate gestureGB [22] with this system. GestureDB proposes a new way of querying by making use of gestures that translate into queries. By integrating gestureGB with this system, the results can be visualized in an online manner. Gestures can be used to compute the priority of query. Since a gesture, for generating a query, involves a sequence of motions, priority will play an important role. When the user starts a gesture the priority of the probable query can be low. As the user moves through the sequence, the priority can increase, which results in faster execution of the query. This is also useful when moving from one gesture to the next. As the user transits
from the previous gesture to next one, the priority of the previous query can decrease and
the priority of the next probable query can increase. A lot of sudden and fast gestures can
lead to hysteresis, which our system is equipped to handle.
References


[22] Querying Without Keyboards, CIDR, 2013