Trigger: A Hybrid Model for Low-Latency Processing of Large Data Sets

by

Min Xiang
B.Eng., Beijing University of Posts and Telecommunications, 2010
M.Eng., Beijing University of Posts and Telecommunications, 2013

An Industrial Project Report in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Min Xiang, 2015
University of Victoria

All rights reserved. This report may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Trigger: A Hybrid Model for Low-Latency Processing of Large Data Sets

by

Min Xiang
B.Eng., Beijing University of Posts and Telecommunications, 2010
M.Eng., Beijing University of Posts and Telecommunications, 2013

Supervisory Committee

Dr. Yvonne Coady, Supervisor
(Department of Computer Science)

Dr. George Tzanetakis, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Yvonne Coady, Supervisor
(Department of Computer Science)

Dr. George Tzanetakis, Departmental Member
(Department of Computer Science)

ABSTRACT

Large data sets now need to be processed at close to real-time speeds. For example, video hosting sites like Youtube [18] and Netflix [22] have a huge amount of traffic every day and large amounts of data needs to be processed on demand so that statistics and analytics or application logic can generate contents for user queries. In such cases, data can be *stream processed* or *batch processed*. *Stream processing* treats the incoming data as a stream and processes it through a processing pipeline as soon as the stream is gathered. It is more computationally intensive but grants lower latency. *Batch processing* tries to gather more data before processing it. It consumes fewer resources but at the cost of higher latency. This project explores an adaptable model that allows a developer to strike a balance between the efficient use of computational resources and the amount of latency involved in processing a large data set. The proposed model uses an event triggered batch processing method to balance resource utilization versus latency. The model is also configurable, so it can adapt to different tradeoffs according to application specific needs. In a very simple application and a extremely best case scenario, we show that this model offers a 1 second latency when applied to a video hosting site where a traditional batch processing method introduced 1 minute latency. When the initial system has low latency, this model will not increase the latency when appropriate parameters are chosen.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vi**

List of Figures **vii**

Acknowledgements **viii**

1 Introduction **1**
1.1 The Application 2

2 Literature Review **4**
2.1 Live Data Streams 4
2.2 Stream Processing 5
2.3 Batch Processing 5
2.4 Deferred Evaluation 7
2.5 Summary 7

3 Design of the Trigger Framework **9**
3.1 The Data Event Model 9
3.2 Processing Model Component 10
3.2.1 Event Queue 10
3.2.2 Processing Scheduler 11
3.2.3 Data Processing Logic 12
3.3 Summary 13

4 Implementation 14
4.1 Libraries Used . . . 14
4.2 Event Generation and Queueing . . . 15
4.3 Processing Scheduler . . . 16
4.4 Data Processing Logic . . . 18
4.5 Summary . . . 19

5 Analysis 20
5.1 Environment Settings . . . 20
5.2 Event Delay Limit . . . 20
5.3 Buffer Size . . . 21
5.4 Data Rate . . . 23
5.5 Application Performance . . . 24
5.6 Summary . . . 25

6 Conclusions and Future Work 26

A Additional Information 27

Bibliography 32

List of Tables

Table 2.1 Popular Data Processing Models Comparison Table 8

Table 5.1 Performance Comparison Table 24

List of Figures

Figure 1.1 Original Data Processing Model of the Website 3

Figure 2.1 Stream Processing Model 5
Figure 2.2 Batch Processing Model 6

Figure 3.1 Event System and Processing Model Architecture 10

Figure 4.1 An Example of Data Event 16
Figure 4.2 Processing Scheduler Logic 17

Figure 5.1 Event Delay Time under Different Event Delay Limit 21
Figure 5.2 Processing Cost under Different Event Delay Limit 21
Figure 5.3 Event Delay Time under Different Buffer Size 22
Figure 5.4 Processing Cost under Different Buffer Size 22
Figure 5.5 Event Delay Time under Different Data Rate 23
Figure 5.6 Processing Cost under Different Data Rate 23

ACKNOWLEDGEMENTS

Thank you to everyone helped me complete this project and gave feedback during the design of this project. Thank you to my supervisor, Dr. Yvonne Coady for her support. And thanks to my wife Xin Zhang for her support, encouragement and patience during my project term.

Chapter 1

Introduction

In many modern network applications, users generate a great deal of content very quickly. It is important to react to this data on demand, in close to real-time speed. These data sets are called *live data sets*. Task like *summarize*, *aggregate* and *analyze* on user generated content previously was considered to be a periodic tasks that only executed offline when all data is collected and run in a *batched* processing manner. However, in modern web applications the need to show user statistics or summaries for *live data sets* at a close to real-time speed is increasing. This trend encourages research on methods involving close to real-time processing on a continuous flow of user generated data.

There are two main categories of methods that approach the problem. One is *stream processing* [1] and the other is *batch processing* [6] [26]. *Stream processing* treats the data as a continuous flow and processes the data as soon as it comes into the system, while batch processing accumulates data and processes by larger quantities of data as a group..

Stream processing tries to get the best performance on this problem and handle data in a continuous manner. In applications where data needs to be processed as a flow of data records, each data record can be treated as an event and the data flow can be treated as an event stream. Many approaches treat the problem as if they are processing the stream in an event driven way where each event is handled by the processing system as soon as it is received. These methods usually have very responsive results where the latency between processing the result and the incoming events is low. But for applications that do not require this low latency, batch processing can suffice.

Batch processing tries to group data records. In some scenarios, the data process-

ing can be done in a batch manner to get better performance by using technology that helps accelerate processing large sets of data such as MapReduce [11]. In this scenario, batch processing is a more favoured way to process the data as it can result in savings in terms of resource utilization. But the drawback of using batch processing is sometimes larger latency in response time.

This project proposes a hybrid model for doing on demand processing on live data sets. The resulting framework intends to fill the gap between balancing on demand performance and efficiency of live data processing, and has following objectives:

1. Define a processing model for live data sets that addresses both efficiency of processing and response latency.

2. Offer an adjustable framework for developers to explore tradeoffs between performance and resource utilization.

This model is explored in a prototype implementation of a framework we call *Trigger*. Trigger offers a flexible design allowing developers to adjust the latency after the system is implemented. The goal of Trigger is focused on live data set processing where there is a potential to benefit from batch processing such as MapReduce. It could also benefit systems designed to use batch processing for analytics that wish to shift to live data processing.

1.1 The Application

This model was originally designed for a video sharing website where user can replay videos on demand through a web server. The problem we needed to solve was to generate analytics information for video uploaders. Analytics information consists of statistics such as total click count on each video and total user time spent on each video. In order to calculate these statistics, many user interactions with the website are recorded through a data collection API and written to a database. Statistics are then generated by processing the data using MapReduce.

The video sharing site originally relied on data processing through a timed *cron* job to execute the MapReduce processing batch. A *cron* job is a time-based job scheduler which can be used to schedule periodic executions. Due to the limitations of the cron job scheduler, the maximum frequency of batch processing execution cannot be faster than 1 execution per minute. This resulted in a human perceivable delay between

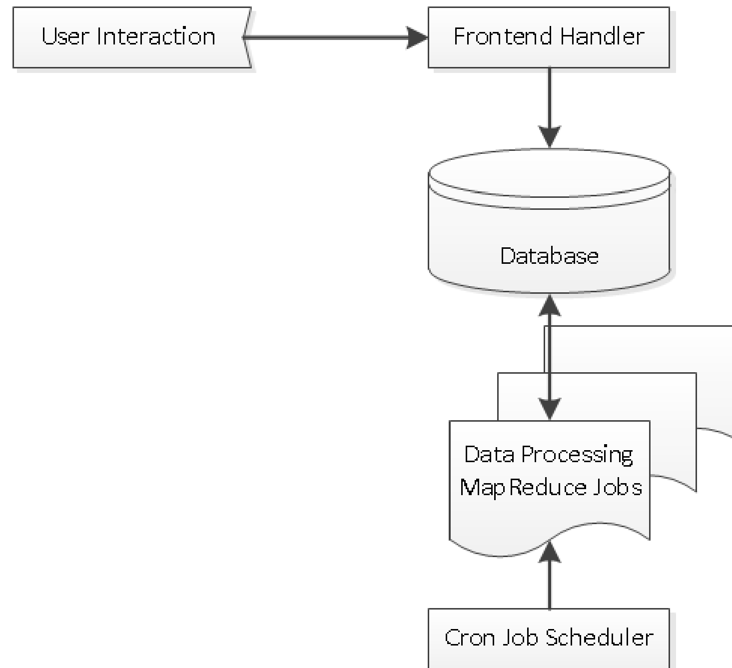

Figure 1.1: Original Data Processing Model of the Website

the data generation and the statistics availability in the database. This scenario is shown in Figure 1.1.

Applying the model designed in this project to the video site, the goal is to reduce the delay and make the data processing more interactive while still allowing the MapReduce processing logic to be executed in a batch. With the new model applied to the site, we expect to get:

1. Reduced delay between data collection and result generation.

2. Increased overheads in terms of processing logic but not prohibitive or a severe performance penalty.

The remainder of this report is organized as follows. In Chapter 2, we review the traditional processing models and methods used for live data sets . In Chapter 3, the design of the hybrid processing model is explained. In Chapter 4, we show the actual implementation of the hybrid processing model. Chapter 5 gives the evaluation of the hybrid processing model. Chapter 6 gives our conclusions and possible future work.

Chapter 2

Literature Review

Live data processing is a very popular topic. Many processing models widely used in industry for live data processing are stream processing engines, such as Storm [16], Spark [15] and S4 [14] have recently gained a lot of attention due to their performance in industrial applications. While some other systems use batch processing applied to a distributed system to gain even better performance. Systems, such as BigTable [9] and Hadoop [27], all fall into this category and they can also be used to process live data. This review will introduce live data sets and go over popular methods for live data processing, briefly describing the features and strengths of each method.

2.1 Live Data Streams

Live data streams are continuous unbounded flows of data. Live data streams mostly exist in web services [3]. In these systems, data is generated from users by interacting with the application, such as clicking on buttons in user interfaces or accessing URLs from a browser. Processing live data streams is different from processing data that already resides in databases or data files, since the pattern of data streams arrival is uncertain and data processing can only happen after data arrival. Therefore, scheduling data processing is important. One important performance metric for live data processing is the latency, which is how much time is required to generate the result of processing after the data arrives. The other performance metric used in this project is efficiency, which is how much of a general computing resource is used to process each piece of data in the live data stream.

2.2 Stream Processing

Stream processing is a very popular way to process live data. One representative framework is Spark [15]. Data resources are considered streams and are processed through a process pipeline as shown in Figure 2.1. Data streams in Spark are represented as a sequence of Resilient Distributed Datasets (RDD), which is a data abstraction in Spark. Spark processes the stream on the fly. The processing begins as soon as data reaches the pipeline. Although Spark does have batch concept, Spark only processes micro-batches that break stream data into small groups as RDDs so it is packed in a way that can be passed through a pipeline. The strength of stream processing is low latency, but the resource usage is high. It usually cannot apply complex analytical logic on live data sets, since the data arrival pattern is uncertain [20]. Approaches can be applied to improve the efficiency of stream processing by trading off response speed, such as predicting the data arrival pattern by probabilistic methods [7] [12] or machine learning methods [2] [17] [21] and then applying better scheduling for the stream processing.

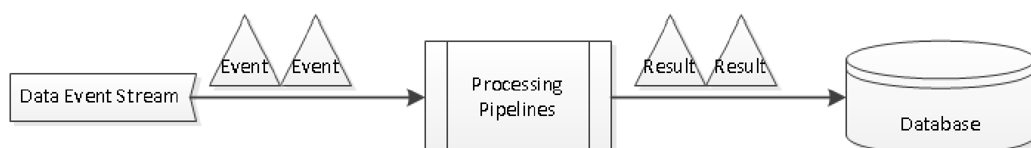

Figure 2.1: Stream Processing Model

2.3 Batch Processing

Batch processing is also an attractive way to process live data sets. The traditional way to deal with large data sets is still batch processing, which includes distributed computing [26] [24] and cluster computing [29]. Batch processing is a good way to accelerate large data set processing [4], as batch processing reduces a lot of overheads compared to processing data one by one. Batching methods like MapReduce are

widely used in both archived data and live data. Not only are batch methods good for live data, but they are also easier to use when creating complex logic for data processing and have better scalability [6] than stream processing methods.

MapReduce is a programming model for processing large data sets in parallel, distributed algorithms on clusters [11]. It uses two procedures to implement the processing algorithm which are *map()* and *reduce()*. *Map()* procedure can perform single data event related processing, such as filtering and sorting. *Reduce()* procedure performs a summary operation which generates the result. Because the *map()* procedure can run simultaneously on all machine in a cluster, it provides very good scalability over large data sets. We show a simple batch processing system workflow in Figure 2.2.

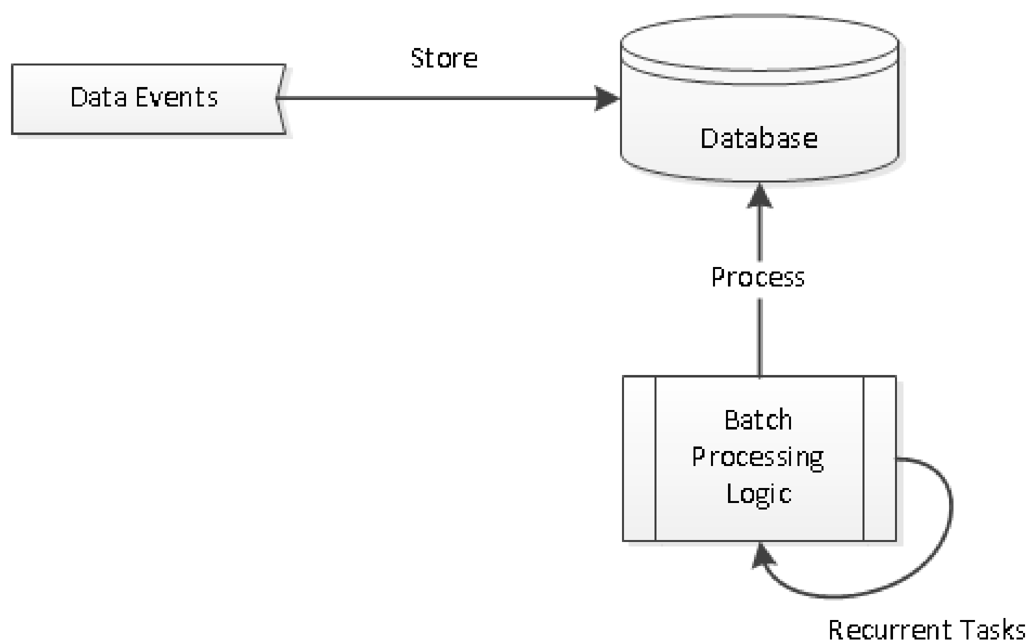

Figure 2.2: Batch Processing Model

Timed cron job is also a very common way to do calculations on live data sets in batch format. The processing is not triggered by data events or the incoming data stream, but periodically executes processing logic on any new data that appeared in the database. The cron job model is very good at data processing when delay does not matter, since cron jobs cannot be triggered very frequently due to the consumption

of computing resources. But as a cron job, it is isolated from the data collecting procedure and do not depend on event systems; this advantage is useful to decouple the processing system with the data collecting procedure. An example of cron job system widely used is Crontab, which is Unix-like system built-in job scheduler [19]. While in Crontab, the minimal time span can be set between two cron job executions is 1 minute, which is typically not good enough for time sensitive live data applications.

2.4 Deferred Evaluation

Deferred evaluation [13] [8] is also a good way to reduce resource consumption for live data processing. Since in most cases the generated result is only needed when the user is trying to retrieve the result, the data processing is only necessary to be completed before the actual result is required. Deferred evaluation uses this property to delay the processing until the first time a particular result is actually needed. Delaying the computation reduces the overhead cost of setting up processing for each data event by processing a set of items together. But the deferred evaluation can cause a lengthy processing period when there is a large set of data to be processed, and may cause delay when a result is needed.

2.5 Summary

As shown in this chapter, there are advantages and disadvantages for each of these data processing models when applied to live data sets. We list advantages and disadvantages in Table 2.1. Stream processing provides best latency at the cost of efficiency. However, batch processing and deferred evaluation give good efficiency on resource utilization but have higher latency. The scalability for stream processing is average since it depends on how fast the data stream can be distributed to the processing pipeline. The scalability for batch processing is good since it is isolated from the data collection procedure and can use scalable processing methods, such as Mapreduce. Deferred evaluation is not good idea for large data sets since deferring too much data and processing in a short time does not scale very well. There is not a single processing model that is strong at all aspects. Tradeoffs must be made before designing a data processing system unless a hybrid model can be made to provide a balance between these aspects.

Model	**Latency**	**Efficiency**	**Scalability**
Stream processing	Good	Average	Average
Batch processing	Bad	Good	Good
Deferred Evaluation	Average (Bad on worst case)	Good	Bad

Table 2.1: Popular Data Processing Models Comparison Table

Chapter 3

Design of the Trigger Framework

This chapter describes the structure of the processing model and how it works to process live data sets. The main goal of this model is making batch data processing logic be able to work on live data sets. We first explain the basic assumption of live data sets. Then we describe the components of the model and how the data processing logic works with it.

3.1 The Data Event Model

Similar to the stream processing model, Trigger is based on the event stream model. The live data set source is treated as a stream of events where each data record incoming is regarded as an event sent to the system. Based on the data format in our applications, we assume the data events have same chunk size and there is no dependencies between data records.

Each event sent to the system is considered an integral data record, so the data sent to the system does not contain incomplete or invalid data records. This can be ensured by the data generating module by only sending out events containing a complete valid data record. If the data originally generated does not guarantee integrity, we can always add a translation layer that transforms the data to a valid form and handles issues with incomplete data records.

Each event can either carry the data record it represents or just a notifier. Since the processing system is focusing on batch processing, it is not efficient to pass the data record through the message passing system because it will involve additional data copy overhead. Because data will eventually be stored in a persistent storage

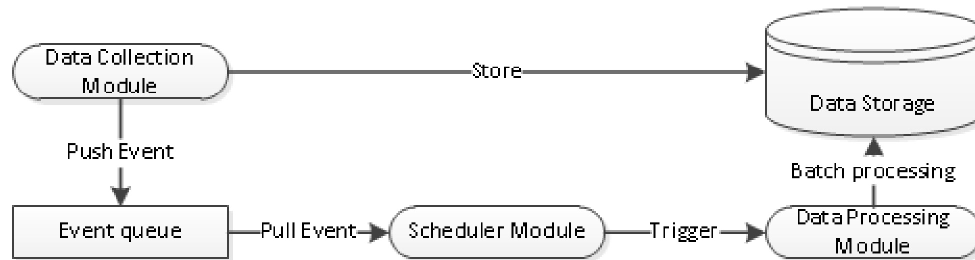

Figure 3.1: Event System and Processing Model Architecture

such as databases or files, loading a batch of records from persistent storage is usually faster than copying the records one by one through the messaging system. If a data events is just a notifier, the processing logic will the relevant data from databases when processing data.

3.2 Processing Model Component

The processing model consists of three components: the event queue, the processing scheduler and the data processing logic. The event queue temporarily stores events and maintains their order. The processing scheduler reads from the event queue and arranges data batch processing to achieve latency requirements. The data processing logic is only a consumer of the processing model, it is triggered by the processing scheduler. The data processing logic has to follow the interface provided by the processing model to benefit from it, and it must cooperate with the processing scheduler to be efficient.

3.2.1 Event Queue

The event queue is used to accept data events from the data source. We assume the data source is an event stream in the processing model. The event queue holds data events and waits for the processing scheduler to trigger and remove data. The event queue has storage space for data events and can persist data events until the processing scheduler dequeue events. In case of failures, the event queue can also be implemented to have persistent storage so the data event will not lose through system

failures.

For getting events from the queue, the event queue should support block waiting, so that the processing scheduler can wait on the event queue and does not use busy waiting when there is no event sent to the queue. If each data event carries the data record, the data record will be sorted in the event queue encoded and the event queue will ensure the processing scheduler will get a complete data event and data record when retrieving events.

3.2.2 Processing Scheduler

The processing scheduler is the component that collects data events from the event queue and arranges data processing according to latency limit and efficiency requirements. The scheduling therefore should achieve following objectives.

1. Each batch process executed should be efficient, and each single batch should not have too few events to process when possible.

2. The scheduled processing should meet the latency requirement. The delay time between a data event received at the event queue and its corresponding result coming out must be within the specified limit..

To achieve the first objective, the processing scheduler holds the events received from the event queue until enough data events are collected to run a large enough batch process. The processing scheduler has its own storage, besides the storage of the event queue, and it uses the storage to hold data events. To achieve the second objective, the processing scheduler will have timing logic such timer callback, that allows it to trigger the processing when the deadline of a data event is due. Finally, the processing scheduler triggers the data processing logic to process the data according to both principles above. Data processing logic is triggered when the number of data events stored in the processing scheduler reaches the limit in the first objective or one of the stored data events is about to be due. Each time the processing scheduler triggers batch processing it sends all stored data events to the data processing logic. After the processing scheduler sends out data events, it clears all stored data events.

There are two key parameters that are adjustable for the processing scheduler, one is the due time for each data event and the other is the number of data events that are considered efficient for a single batch. By adjust these two parameters, the

model can be configured to balance between faster reaction to data events or more efficient batch processing.

3.2.3 Data Processing Logic

The data processing logic collects the data records batched by the processing scheduler and performs the data processing. It is trivial when data records are carried with data events and then passed to the data processing logic from the processing scheduler. The data processing logic will be able to process the data it received along the data events without accessing the persistent storage that stores the data. But if the data record is not attached with the data event and stored in persistent storage such as databases or files, the data processing logic has to find these data records in persistent storage first.

The problem is if the data processing logic has to retrieve the data record from persistent storage, it has to determine which records are new and need processing and which records are processed so it can avoid processing a previously processed data record. One way to filter the data record is to have a reference on each data event that points to the corresponding data record in persistent storage and the data processing logic fetches them one by one when processing them. But this way the efficiency of processing is low because the lookup for the data records take resources. Another method to filter out data records is to query the database on an indexed key to retrieve a batch of records that will be processed; this requires the cooperation from data collection module to put special tags on each data record in the database. A common way to put tags on records is to put a timestamp or a sequence number on each data record, and the data processing logic fetches records for timestamps between the current time and the timestamp of the last record of the last batch to get all records needed for the current batch.

Data processing logic benefits from preventing data records from being copied through modules. By making the data event not contain the data record, the system prevents the processing framework from creating redundant copies through the messaging system. Data processing logic can further prevent data copying if it can run distributed processing such as MapReduce which sends logic to where the data is, instead of fetching the data to where the data processing logic is. Fortunately, there are databases such as HDFS [5] and MongoDB [10] that support this sort of MapReduce processing where the processing is done inside the database where the

data records are and users just send a piece of logic to the database through proper interface. By doing so, the entire data processing system will only send notification events and will not incur extra data record copying costs other than the initial storage cost by the data collection module which is always necessary even when the data does not require processing.

3.3 Summary

The hybrid processing model is based on components introduced in this chapter. First of all, the live data sets are converted into data events by the front end module according to the data event model introduced in this chapter. The front end logic stores data records in database and notifies the event queue when data events arrive from live data sets. The processing scheduler waits on the event queue; when a data event is captured by the processing scheduler, it is temporarily stored in the processing scheduler. The processing scheduler triggers the batch data processing on two conditions. The first condition is enough data events have been stored. The second condition is any data event has been waiting for enough time. The two triggering conditions make the processing model a hybrid model. When the live data set is generating data at a fast rate, the batch processing is usually triggered by the first condition and the processing model works like a stream processing model offering low latency data processing. When the data generating rate is low, the batch processing is usually triggered by the second condition and efficiency of resource utilization is ensured with the latency remaining in an acceptable range. Finally, when the batch processing is triggered, a scalable batch processing method such as MapReduce is applied with minimal data copying overheads.

Chapter 4

Implementation

This chapter discusses some details of the implementation of Triggers hybrid processing model. Trigger leverages some of the popular out-of-box tools and framework such as event queues and databases as part of its implementation.

4.1 Libraries Used

Trigger is developed primarily in Python, using Flask [23] as the framework for front end event handling, RabbitMQ [25] as the event queue service and MongoDB [10] as the persistent data storage. Flask offers a simple routing service which can be used as a entry for the data events collector as well as a result viewer in the project. Recently Flask has became more popular and widely used, it is also well documented. RabbitMQ is an open source and easy to use message queue system that supports a number of platforms. RabbitMQ offers both blocking and non-blocking methods to access the queue. It also has a message persistence feature and is known for its reliability. RabbitMQ has no restrictions for the message format so it is easier for the processing model to reformat the data event structure when needed. MongoDB is used for the data storage. It is an easy to use database and has an active user community. More importantly MongoDB has a build-in MapReduce feature that allow user to specify data processing logics that run inside the database saving the cost of data transferring. This kind of batch processing feature is a major benefit within the proposed processing model.

4.2 Event Generation and Queueing

The event source is a front end program accepting HTTP requests and converting the requests to data events. We use a Python dictionary object to store each data event in the front end program. It contain a branch of key value pairs, and the schema depends on the application. A special key named 'time' is always contained in a data event which is the unique timestamp of the event. Timestamps is also stored in database with data records for applications use data events only as notifications. This makes sure the data processing logic can later fetch the corresponding data records correctly. Event sequence numbers can also be used as the identifier for each data record. We did not use event sequence numbers in this model. If event sequence number is used, additional consistency protocols will be needed to generate and maintain sequence numbers in case of front end failures.

The event generation logic creates a named blocking queue for each data processing logic and populates the queue when data events are generated. Data events are serialized and packed as a string when pushed to the message queue. This ensures the processing scheduler can retrieve messages from the message queue and deserialize back to event objects. The event generation logic always puts data records in the database before populating the message queue, preventing the processing logic triggering before data records are stored in database. This project did not use the popular JSON format for the serialization. The Python built-in serialization library is used to serialize the dictionary object which contains all event information including the timestamp, since it provides more efficient deserialization and has shorter serialized strings.

The code for the front end to use is very flexible. The framework offers two helper libraries that help front end applications to use the datastore and the event queue without directly calling the underlying library and cover details such as queue creation and database connection. Helper library 'rmq' is used for using message queue and helper library 'Store' is for the database storage. An example of a front end application that collects key value pairs and stores them in database while populating the event queue is shown in Listing 4.1. The 'time' property for the data record is the timestamp mentioned above that helps the data processing later.

Listing 4.1: A Simple Application front end

```
app = Flask(__name__)
@app.route('/insert/<string:key>/<int:value>')
def insert(key, value):
    connection = rmq(queue_ip)
    doc = {'time': datetime.utcnow(), key':␣key,␣value': value}
    s = Store(db_ip)
    s.store(db_name, collection_name, doc)
    connection.push(queue_name, doc)
app.run()
```

The code in Listing 4.1 is only a sample of using the processing model through the framework which handles both event queue operation and database operation. In a real application, data events can have more attributes and they can have different names other than ‘key’ and ‘value’. An example of a data event object generated in application looks like Figure 4.1.

```
{
  'time': datetime.datetime(2015, 7, 7, 5, 7, 10, 419060),
  'name': 'item',
  'data': 30,
  ...
}
```

Figure 4.1: An Example of Data Event

4.3 Processing Scheduler

The processing scheduler retrieves each data event from the message queue as a Python program. Each component of processing logic is assigned a processing scheduler. The processing scheduler waits on the named message queue. Each time the processing scheduler successfully retrieves a serialized data event from the message queue it deserializes the message and recovers the data event as an object. After the data event object is recovered, the processing scheduler triggers its logic.

Before introducing some of the implementation details of the processing scheduler logic, some data structures used in processing scheduler are introduced here. Each processing scheduler instance holds a reference to the corresponding data processing logic and a buffer to put pending event objects. Each processing scheduler instance

also holds two parameters, one is the max delay time for a single data event which denoted as t and the buffer size which denoted as k. The processing scheduler also holds timers to deal with deadline triggers.

The processing scheduler has very simple logic. When a data event hits the scheduler, it checks if the buffer for pending events is empty; if it is empty then the scheduler sets up a timer with duration of t. The scheduler then appends the newly received event into the pending events buffer. Finally it check if the buffer is full. The scheduler will call data processing logic to process the data events when the buffer is full or the timer goes off. When the scheduler triggers data processing logic, it sends all the pending data events in the buffer to the processing logic, then clears the buffer and resets all timers.

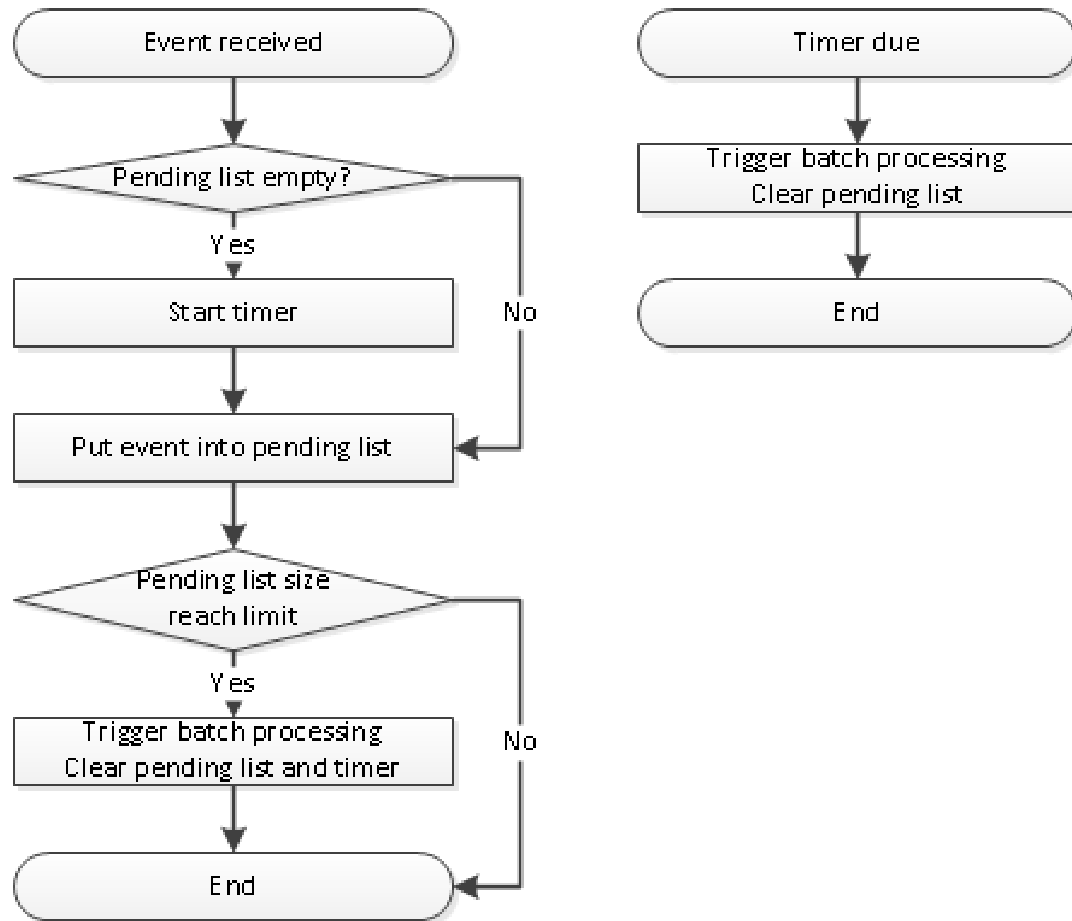

Figure 4.2: Processing Scheduler Logic

Timers in the processing scheduler ensure the max delay time of each event. The

buffer size prevents the processing from holding too many events for a single batch. Both limits are parameters to each processing scheduler instance, and are configured at the initiation of the instance.

4.4 Data Processing Logic

Data processing logic accesses databases through a Python library for corresponding databases. The processing logic is implemented as a MapReduce job, each type of data processing logic is implemented as a class in Python and the MapReduce logic is implemented inside the class. Executing the processing logic require two timestamps, the timestamp of the earliest data event and the timestamp of the latest data event. The processing logic is then applied on the data record with a timestamp between the two timestamp bound. The processing logic does not pull any actual data record from the database. Instead it uses the MapReduce function with a query argument to achieve the timestamp filtering, the final result is also stored in the same database instance so the processing logic avoid all data record copy between processing system and the database.

In the framework, each data logic is implemented as a single class, which all inherit from a base class called *worker* which is listed in Listing 4.2. Data processing logic can be created easily using the Worker class. For example an processing logic tries to add all 'data' field of the data record and generate a total can be written as listed in Listing 4.3. This processing logic utilize the MapReduce function in database and provide timestamp arguments that allow the data processing on data events within time window:

Listing 4.2: The Worker Class

```
class Worker(object):
  def __init__(self, url, db, collection):
    self.client = MongoClient(url)
    self.db = db
    self.collection = collection
  def _run(self):
    return
        self.client[self.db][self.collection].map_reduce(self.mapper,
        self.reducer, self.out, **self.args)
```

Listing 4.3: The Sum Worker

```
class Sum(Worker):
  def __init__(self, url, db, collection, output):
    super(Sum, self).__init__(url, db, collection)
    self.mapper = Code('function() {emit(this.name, this.data)}')
    self.reducer = Code('function(key, values) {return 
       Array.sum(values)}')
    self.out = {'reduce': output}
  def run(self, begin, end):
    self.args = {'query': {'$and': [{'time': {'$gte': begin}}, {'time':
       {'$lte': end}}]}}
    return self._run()
```

4.5 Summary

The implementation includes helper libraries, a front end program, a processing scheduler and batch processing logic. Helper libraries simplify the access to the event queue and database storage. The front end program converts live data sets into data events and populates the event queue. The processing scheduler implements the hybrid model logic described in Chapter 3. The batch processing logic implements the logic needed to process the data through a MapReduce method supported by the database. The full source code is hosted on Github [28]. Some of the key code also involved in the Appendix.

Chapter 5

Analysis

To analyze the hybrid model, different parameters are configured and tested with a large load of live data. The tests determine that the model will have different performance between processing efficiency and latency in different configurations. To achieve this, additional code is added to the processing logic to record delay time and amortized processing time for each event.

5.1 Environment Settings

In this analysis test, simple processing logic is used. The processing logic adds up all values in 'value' field for data records that have same 'key' field, and stores the total for each 'key' field back in the database using MapReduce. Different delay limits and buffer sizes in the scheduler logic are applied. Maximum and average delay on data events and average processing time cost for each data record is calculated to compare performance of different settings. Different data generating rates are tested to show the performance under different velocities of live data set.

5.2 Event Delay Limit

Different event delay limits are tested through a fixed event generating rate and fixed buffer size. The maximum and average delay and the amortized processing cost for each data event is changing through different delay settings.

In the delay limit analysis, data event generating rate is 10 per second and the buffer size is 100. As shown in Figures 5.1 and 5.2. The processing latency increases

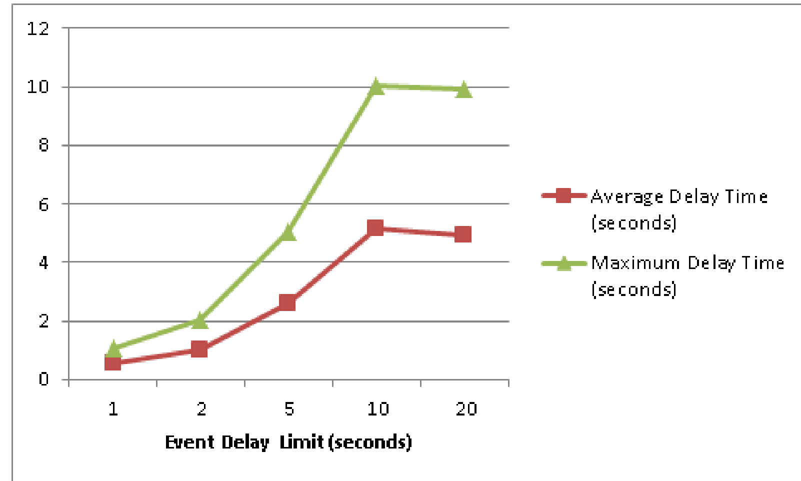

Figure 5.1: Event Delay Time under Different Event Delay Limit

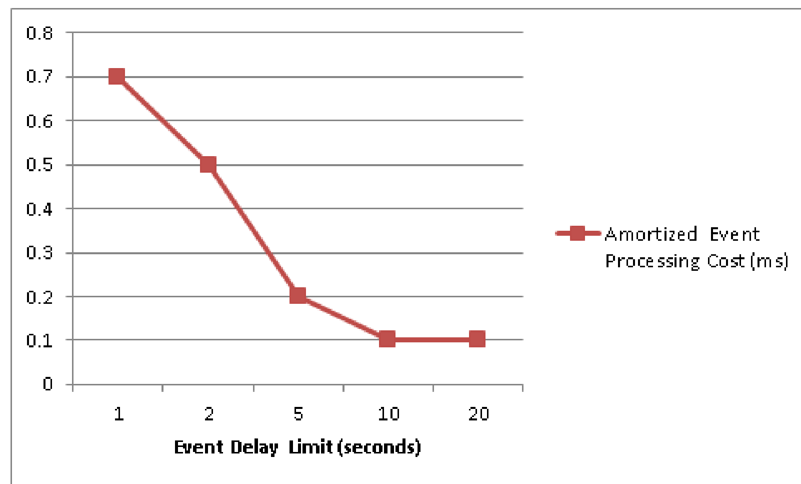

Figure 5.2: Processing Cost under Different Event Delay Limit

and processing cost decreases on data events when the delay limit increases. Change capped at 10 seconds because the buffer size limit dominates performance at that point. This test result shows that the processing model is able to adjust its performance and latency by tuning the delay limit. More delay limits provides better processing efficiency.

5.3 Buffer Size

For buffer size analysis, different buffer sizes are chosen while other parameters remain the same as the delay limit test. Processing cost and event latency are observed to

show the difference between settings.

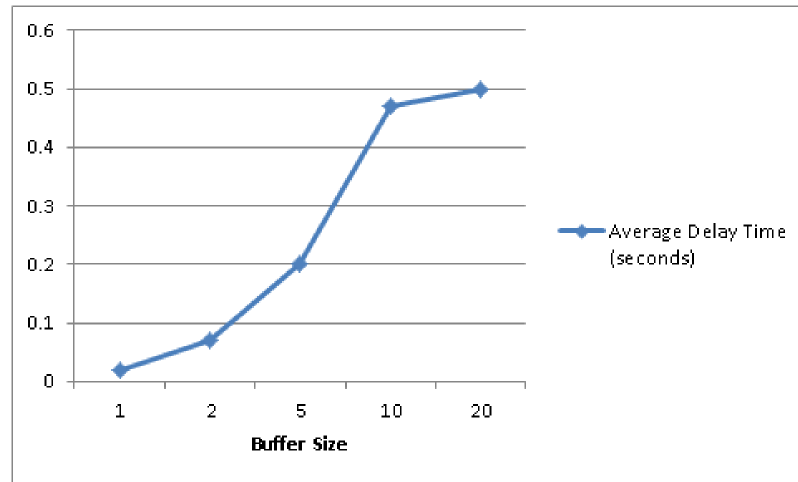

Figure 5.3: Event Delay Time under Different Buffer Size

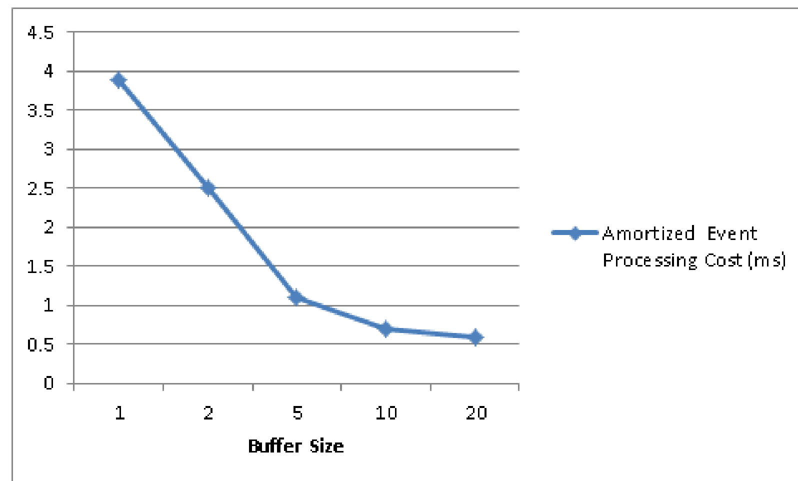

Figure 5.4: Processing Cost under Different Buffer Size

In this test, data event generating rate is 10 per second and the delay limit is 1 second. As shown in Figures 5.3 and 5.4, the actual delay increases as the buffer size increases. The efficiency of the data processing also increases, as we can see amortized processing cost on each data event decreases. This result shows that increasing the size of the buffer can benefit the processing efficiency at the cost of higher latency.

5.4 Data Rate

It is also good to know whether the characteristics of the data source can affect the performance of this model, so this test goes with a fixed setting on buffer size and event delay limit. The performance under multiple data source rates are observed and recorded.

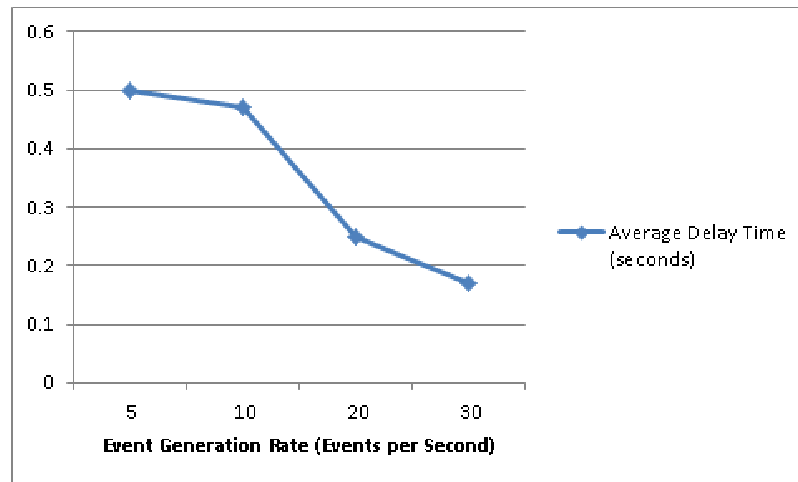

Figure 5.5: Event Delay Time under Different Data Rate

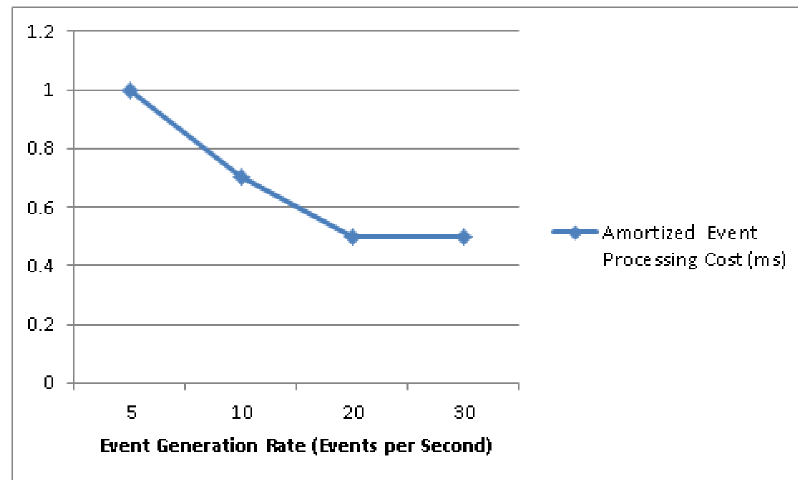

Figure 5.6: Processing Cost under Different Data Rate

In this test, the delay limit is 1 second and the buffer size is 10. As shown in the Figures 5.5 and 5.6, the latency reduces as the event generating rate increases before it reaches 10 events per second. This is because at 10 events per second the buffer size

Model and event rate	Cost per event (ms)	Avg. delay (seconds)	Max delay (seconds)
Original model, 10 evt/sec	0.02	29.9	59.8
Hybrid model, 10 evt/sec	0.6	0.5	1.0
Original model, 5 evt/sec	0.02	30.5	59.9
Hybrid model, 5 evt/sec	1.1	0.5	1.0
Original model, 2 evt/sec	0.05	29.9	59.8
Hybrid model, 2 evt/sec	1.5	0.5	1.0

Table 5.1: Performance Comparison Table

will cap and pending data events in the processing scheduler will trigger the processing because buffer size instead of delay timer. When the event generating rate is low the 1 second delay limit takes effect and ensures the event delay time. Therefore, the amortized processing cost for each data event is decreasing as the event generating rate increases but capped at 0.5ms per event. This is also because the buffer size limits the number of events in a single batch. Even if the event generating rate is high enough, a single batch will only contain 10 events according the parameter of this test. The processing efficiency can not improve unless buffer size is enlarged to enable larger batch.

5.5 Application Performance

This hybrid data processing model is designed for a video sharing website that requires low-latency data processing for statistics. After the new model is applied some performance metrics are collected to compare with the original data processing model. The original model used a timed cron job model that tries to process any incoming data events every minute for all available events. The hybrid model we applied uses parameters with 1 second delay limit with a buffer size of 20 events.

From Table 5.1, the amortized cost for processing each event is increased approximately 1 millisecond and the delay time is reduced from 1 minute to 1 second after applying the new processing model. This result meets our goal, as the delay time of events is limited to a level that users of the website can rarely sense statistics delay. Since the processing is becoming more frequent and the batch size is relatively small in the new model, it is expected to have the processing cost increased. It is, however, still efficient as within a few milliseconds. Furthermore, the parameters can

be tuned in the new model to fit different event arrival patterns and different latency requirements.

5.6 Summary

Through testing, it is shown that the hybrid processing model can be configured to prefer either low latency or high processing efficiency. Reducing the delay limit or the buffer size will allow lower latency on event reaction time with less efficient resource utilization. Increasing the delay limit or the buffer size will make the batch processing more efficient at the cost of higher latency.

In cases the data generation can be very fast at some period, larger buffer size will provide the chance to have larger batch size and better processing efficiency. It will also take more space to store events in the processing scheduler model. For data source having fast generating speeds, the buffer size dominates the performance preference. At a fast generating rate, the events never wait the duration of the event timer but instead are triggered because the buffer is full. For a slow data source, the delay limit dominates the performance, because the buffer is rarely fully filled.

Chapter 6

Conclusions and Future Work

This project presents a prototype for low latency batch processing live data streams based on an adaptable framework. It achieves the goal of processing the data efficiently and also with a low delay time. This processing model strikes a balance between stream processing and batch processing. It is suitable for processing live data sets and is also able to adjust through parameters to have better efficiency or reduced delay time. This feature enables developers to explore tradeoffs between performance and reaction speed for data processing, making live data processing more adaptable.

The processing model proposed by this project is rich in possibilities for further improvement. One way to improve the processing model is making it more adaptive to live data streams, since in many real cases the live data streams do not necessarily have a fixed generating rate. As different rates are proven to have different optimized parameters in this model, it will be better to have an adaptive model that will adjust the parameters based on the rate and throughput of specific live data streams.

Another way to improve the processing model is by focussing on how to more efficiently utilize the resources for the data processing. The model can be smarter if it detects how busy the data hosting machine is and tries to schedule the processing a relatively idle period. It will make the hosting machine processing the data more effective and result in less impact on normal database accesses such as data gathering routines.

There is also more analysis can be done to discover optimal configurations for various type of live data streams that have different data generation patterns.

Appendix A

Additional Information

Listing A.1: Application Frontend

```
from flask import Flask
from store import Store
from datetime import datetime
from rmq import rmq
app = Flask(__name__)

connection = rmq('localhost')

@app.route('/insert/<string:name>/<int:data>')
def insert(name, data):
    global connection
    doc = {'time': datetime.utcnow(), 'name': name, 'data': data}
    print doc
    s = Store('localhost:27017')
    s.store('test', 'doc', doc)
    connection.push('stats', doc)
    return 'Inserted [%s:%d]' % (name, data)

@app.route('/total/')
def total():
    result =
        ['<head/><body><table><tr><td>Name</td><td>Total</td></tr>']
    s = Store('localhost:27017')
    for record in s.find('test', 'total', {}):
      result.append('<tr><td>%s</td><td>%d</td></tr>' % (record['_id'],
          record['value']))
```

```
    result.append('</table></body>')
    return '\n'.join(result)

if __name__ == '__main__':
    s = Store('localhost:27017')
    s.create_index('test', 'doc', 'time')
    app.debug = True
    app.run('0.0.0.0')
```

Listing A.2: Aggregation Worker

```
from pymongo import MongoClient
from bson.code import Code

class Worker(object):
  def __init__(self, url, db, collection):
    self.client = MongoClient(url)
    self.db = db
    self.collection = collection
  def _run(self):
    return self.client[self.db][self.collection].map_reduce(self.mapper,
                                                            self.reducer,
                                                            self.out,
                                                            **self.args)


class Sum(Worker):
  def __init__(self, url, db, collection):
    super(Sum, self).__init__(url, db, collection)
    self.mapper = Code('function() {emit(this.name, this.data)}')
    self.reducer = Code('function(key, values) {return 
        Array.sum(values)}')
    self.out = {'reduce': 'total'}
  def run(self, begin, end):
    self.args = {'query': {'$and': [{'time': {'$gte': begin}},
                                    {'time': {'$lte': end}}]},
                 'full_response': True}
    return self._run()
```

Listing A.3: Processing Scheduler

```
import rmq, sys
from datetime import datetime, timedelta
```

```
import worker

class SumJob():
  def __init__(self, wait_time, max_size):
    self.worker = worker.Sum('localhost:27017','test','doc')
    self.wait_time = wait_time
    self.max_size = max_size
    self.q = []
    self._process_count = 0
    self._process_cost = 0
    self._event_count = 0
    self._total_delay = timedelta(0)
    self._max_delay = None
  def push_event(self, event):
    self.q.append(event)

  def run(self):
    if len(self.q):
      self.worker.run(self.q[0]['time'], self.q[-1]['time'])

  def _timer(self, connection):
    self.run()

  def onevent(self, connection, data):
    if len(self.q) == 0:
      connection.set_timer(self.wait_time, self._timer)
    self.push_event(data)
    if len(self.q) >= self.max_size:
       connection.clear_timer()
       self.run()

delay = float(sys.argv[1])
buffer_size = int(sys.argv[2])
sumjob = SumJob(delay, buffer_size)

connection = rmq.rmq('localhost')
connection.consume('stats', sumjob.onevent)
connection.start()
```

Listing A.4: Database Access Helper Library

```
from pymongo import MongoClient
```

```
def cached(cls):
    _cache = {}
    def new(*args, **kw):
      kw_tp = tuple(sorted(kw.items()))
      args_tp = tuple(args)
      if (args_tp, kw_tp) not in _cache:
        _cache[(args_tp, kw_tp)] = cls(*args, **kw)
      return _cache[(args_tp, kw_tp)]
    return new

@cached
class Store(object):
  def __init__(self, url):
    self.client = MongoClient(url)

  def store(self, db, collection, doc):
    self.client[db][collection].insert(doc)

  def create_index(self, db, collection, key):
    self.client[db][collection].create_index(key)

  def find(self, db, collection, query):
    return self.client[db][collection].find(query)

  def __del__(self):
    self.client.close()
```

Listing A.5: Event Queue Access Helper Library

```
import pika
import pickle

class rmq(object):
  def __init__(self, pram):
    self.connection =
        pika.BlockingConnection(pika.ConnectionParameters(pram))
    self.channel = self.connection.channel()
    self.timer_id = None

  def push(self, queue, data):
    self.channel.queue_declare(queue=queue)
```

```
    self.channel.basic_publish(exchange='',
                               routing_key=queue, body=pickle.dumps(data))
  def consume(self, queue, callback, no_ack=True):
    self.callback = callback
    self.no_ack = no_ack
    self.channel.queue_declare(queue=queue)
    self.channel.basic_consume(self._callback, queue=queue,
        no_ack=no_ack)

  def get(self, queue):
    res = self.channel.basic_get(queue=queue, no_ack=True)
    if res[0]:
      return pickle.loads(res[2])
    else:
      return None

  def _callback(self, ch, method, properties, body):
    self.callback(self, pickle.loads(body))
    if not self.no_ack:
      ch.basic_ack(delivery_tag = method.delivery_tag)

  def set_timer(self, seconds, timer):
    if self.timer_id:
      self.clear_timer()
    self.timer = timer
    self.timer_id = self.connection.add_timeout(seconds, self._timer)

  def clear_timer(self):
    if self.timer_id:
      self.connection.remove_timeout(self.timer_id)
    self.timer_id = None

  def _timer(self):
    self.timer(self)

  def start(self):
    self.channel.start_consuming()

  def __del__(self):
    self.connection.close()
```

Bibliography

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[2] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. Learning-based query performance modeling and prediction. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 390–401. IEEE, 2012.

[3] Masiar Babazadeh, Andrea Gallidabino, and Cesare Pautasso. Liquid stream processing across web browsers and web servers. In *Engineering the Web in the Big Data Era*, pages 24–33. Springer, 2015.

[4] 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*, pages 1–16. ACM, 2002.

[5] Dhruba Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop.apache.org/common/docs/current/hdfsdesign.pdf*, 2008.

[6] Andrey Brito, Andre Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, and Christof Fetzer. Scalable and low-latency data processing with stream mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 48–58. IEEE, 2011.

[7] Michael Carney, Pádraig Cunningham, Jim Dowling, and Ciaran Lee. Predicting probability distributions for surf height using an ensemble of mixture density net-

works. In *Proceedings of the 22nd international conference on Machine learning*, pages 113–120. ACM, 2005.

[8] Petrus Kai Chung Chan, Roberta Jo Cochrane, Sam Sampson Lightstone, Mir Hamid Pirahesh, Richard Sefton Sidle, Tuong Chanh Truong, Michael J Winer, and Calisto Paul Zuzarte. System and method for selective incremental deferred constraint processing after bulk loading data, September 17 2002. US Patent 6,453,314.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[10] Kristina Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 337–348. ACM, 2011.

[13] Craig E Foster and Lawrence M Bain. Deferred processing of continuous metrics, August 31 2010. US Patent 7,788,365.

[14] Apache Software Foundation. Apache s4. `https://incubator.apache.org/s4/`. Accessed: 2015-07-13.

[15] Apache Software Foundation. Apache spark. `https://spark.apache.org/`. Accessed: 2015-06-01.

[16] Apache Software Foundation. Apache storm. `https://storm.apache.org/`. Accessed: 2015-06-17.

[17] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Data Engineering,*

2009. ICDE'09. IEEE 25th International Conference on, pages 592–603. IEEE, 2009.

[18] Inc. Google. Youtube. `https://www.youtube.com/`. Accessed: 2015-07-15.

[19] Michael S Keller. Take command: cron: Job scheduler. *Linux Journal*, 1999(65es):15, 1999.

[20] Alireza Khoshkbarforoushha, Rajiv Ranjan, Raj Gaire, Prem P Jayaraman, John Hosking, and Ehsan Abbasnejad. Resource usage estimation of data stream processing workloads in datacenter clouds. *arXiv preprint arXiv:1501.07020*, 2015.

[21] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *Proceedings of the VLDB Endowment*, 5(11):1555–1566, 2012.

[22] Inc. Netflix. Netflix. `https://www.netflix.com/`. Accessed: 2015-07-15.

[23] Armin Ronacher. Flask. `http://flask.pocoo.org/`. Accessed: 2015-07-10.

[24] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[25] Pivotal Software. Rabbitmq. `http://www.rabbitmq.com/`. Accessed: 2015-06-28.

[26] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank Van Harmelen. *Scalable distributed reasoning using mapreduce*. Springer, 2009.

[27] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[28] Min Xiang. Hybrid model source code. `https://github.com/cff29546/EventTrigger`. Accessed: 2015-07-20.

[29] Fan Zhang, Junwei Cao, Samee U Khan, Keqin Li, and Kai Hwang. A task-level adaptive mapreduce framework for real-time streaming data in healthcare applications. *Future Generation Computer Systems*, 43:149–160, 2015.