Optimizing Query Processing in Batch Streaming System

Peter Xiang Gao
EECS
University of California, Berkeley

Di Wang
EECS
University of California, Berkeley

ABSTRACT

With the growing need of processing "big data" in real time, modern streaming processing systems should be able to operate at the cloud scale. This imposes challenges to building large scale stream processing systems. First, processing tasks should be efficiently distributed to worker nodes with small overhead. Second, streaming data processing should be highly available, despite that failures are common in datacenters. In Spark Streaming [26], the DStream model is proposed to cope the problems aforementioned. DStream stands for discretized stream; data in the incoming stream is divided into small batches for processing. Compared with processing data at the granularity of a record, batch processing has much lower overhead and has a cheaper fault tolerance model. *Lineage* information of each batch is kept for recomputation when failure occurs. Therefore, fault tolerance can be achieved without duplicating processing nodes.

In this paper, we discuss how to optimize query processing in the DStream model. Specifically, we consider the case of Structured Query Language (SQL). SQL provides a declarative interface for the users query on the data. The declarative nature of SQL provides opportunity for query optimization as the execution is decoupled from the semantics of the query. In a streaming system, the same query is executed on similar data over and over again. Hence, the statistics of the data could be obtained for free, as long as the incoming data pattern is not changing abruptly. We study the performance of applying query optimization techniques in the DStream model, and show the advantage of dynamically optimizing stream processing.

1. INTRODUCTION

Processing big data in real time with bounded latency is becoming an important task in various application scenarios. With the incoming data and historical datasets in the data warehouse, decisions must be taken based on the analytic results. As an example, network intrusion detection systems [21] need to aggregate traffic information in the network to find out and drop the attacking flows. Twitter needs to quickly find out the topic trend from millions of tweets generated all over the world. Such workload must be processed by stream processing systems in cloud scale clusters.

In large clusters, a stream processing system must be fault tolerant, scalable and maintains a low processing latency. In Spark Streaming [26], incoming data streaming is divided into discretized batches for processing. Such a stream of discretized batches is called a DStream. The DStream model has several advantages. Since the data is processed in batch, the scheduling overhead is easily amortized, which increases the throughput of the system. In cloud environment, node failure is constant and hence, distributed computing engines must be highly fault tolerant. In DStream, lineage information is kept for fault tolerance. A job on a failed node can be easily relaunched on other nodes with lineage. Further, failed job usually blocks other jobs from proceeding when there is an execution barrier. Spark Streaming allows parallel recovery by splitting the failed job to smaller jobs and executing them on multiple nodes. This speeds up the entire execution upon failure.

Besides scalability and fault-tolerance, a streaming processing system should be robust enough to cope with the varying workloads. Since most of the streaming processing system is operating 24×7, the traffic pattern may change overtime. For example, Twitter streams from Britain, India and United States may show diurnal patterns with different peak period. An efficient execution plan needs to adapt itself to the change of traffic pattern. If we need to join three streams to obtain a common set of popular keywords in the above mentioned three countries, we should always first join the two streams that are offpeak and then uses the result to join with the third stream. However, note that the data rate of the streams varies over time, a static execution plan will not be efficient all the time.

In traditional stream processing systems, operators maintain states. Hence, to be fault tolerant, operators must be replicated, which doubles the resource consumption. Further, once the execution plan is decided at compile time, it is fixed over the lifetime of the stream query. However, at the compile time, the system usually do not have enough statistics on the data stream to decide the optimal query plan. Worse, the data streams are varying overtime; there is no single query plan that is optimal for the data stream all the time. Therefore, query plans must be adapted periodically to meet the ever changing data stream characteristics. This is difficult in traditional streaming system due to the internal states that operators needs to maintain.

Batch streaming system, however, is at an unique position for adaptive streaming query. As data is processed in batch, query plan can be rewritten between the execution of two consecutive batches. Furthermore, though data characteristics change over time, in a microscopic view, the data is

usually similar in two consecutive batches. Therefore, with the statistics collected in previous batches, we can optimize the query plan for future executions. In database systems, collecting statistics for query optimization is at the cost of sampling. A small fraction of data is sampled to obtain the statistics such as the size, mean, or histogram of the data. In batch streaming system, such data can be obtained almost for free. We can simply collect the statistics while processing the previous batch, and use them to optimize the following batches' execution.

In this paper, we consider the case of SQL-like streaming queries. We adopt SQL because it is declarative; users only specify what they want instead of how to obtain them. Query optimizer will figure out the cheapest way to perform the computation. The declarative nature of SQL gives freedom for re-optimizing the query online. In an imperative query engine such as Spark Streaming, optimization is difficult, as users already specified a static data flow for the streams.

We used several types of optimization for streaming queries.

- **Predicate Push-down** Predicates are operators that filter elements in a stream. For example, in the SQL query "SELECT hash(user_name), content FROM tweets WHERE location = 'USA'", the predicate *location = 'USA'* should be executed before the hash(user_name) to avoid hashing redundant user_name outside USA.
- **Window Push-up** Window operator is unique in streaming query. It caches recent several batches and output to another operator. Note that we can push-up window operator to reduce the amount of redundant work. However, note that we can not push a window operator beyond a "group by" operator or a join operator. We optimize the group by and join operators such that they execute data incrementally.
- **Join Operators Reordering** A set of join operators could be reordered to increase the performance while still maintain the semantic correctness. Usually, smaller tables should be joined before large tables.
- **Incremental Group By** When the input stream of a group by operator is windowed, we can do this incrementally to increase the efficiency of the operators.
- **Incremental Join** Similar to incremental group by, join can be also done incrementally to improve the performance.

We implement the streaming SQL query on top of Spark and Spark Streaming. Since we need a dynamic rewritable query plan, we implement all the operators instead of using Spark Streaming's operators. We only used Spark Streaming to obtain input data flows.

2. BACKGROUND

The system is built on top of Spark and Spark Streaming. Spark is a MapReduce-like execution engine that provides an abstract interface called Resilient Distributed Dataset (RDD). Computations are done by transforming RDDs to new RDDs. DStream is proposed in Spark Streaming, a stream processing system. Similar to RDD, DStream can be transformed to another DStream with a predefined transform function. Both Spark and Spark streaming provides efficient fault tolerance model for in memory computation.

2.1 Spark

Spark is the a cluster compute engine similar to MapReduce framework [9]. While MapReduce only allows a map stage followed with a reduce stage, Spark can represent the execution logic as a Directed Acyclic Graph(DAG). Each node in the DAG is a RDD and edges are functions that transform RDDs to new RDDs. A RDD is divided to multiple partitions such that computation can be done in parallel. For each partition, the *lineage* information is kept to trace how it is generated. When a partition is lost on node failure, it can be recomputed using the lineage information.

Hadoop spills intermediate data to disk to provide durability guarantee. In Spark, RDDs can be cached in memory for future use. This is reliable because lost data can be recovered by lineage information. This significantly improves performance as memory has much higher bandwidth than disk.

Recent research [20] shows that data analytic clusters are running shorter and smaller jobs. This is true for streaming tasks, as data is divided into small batches for processing. Spark optimizes the performance of small tasks by launching tasks efficiently. It is able to launch tasks with millisecond latency, while Hadoop takes several seconds to launch a task.

2.2 Spark Streaming

Spark Streaming introduces a new programming model called discretized streams (DStream). DStream treats streaming computation as batches of deterministic tasks on small discrete datasets. By batching the computation, the task scheduling overhead can be reduced and hence improves the throughput. Specifically, Spark Streaming has the following advantages.

The DStream model is highly fault tolerant, as it decouples operator from operator state. In traditional stream processing systems [5, 10], each operator needs to keep an internal state. To be fault tolerant, each operator needs to be duplicated, which doubles the resource consumption. Another approach is to use upstream backup. Each node buffers the data and resends it if a downstream node fails. Recovery takes a long time, since all buffered data needs to be reprocessed. DStream solves this problem by keeping lineage information on each data partition, and recover them by recomputing the task upon failure.

DStream can easily combine historical data with stream data. Traditional systems such as Hadoop and Dryad [15] fail to meet this goal as they keep historical data on disk. Loading these data takes significant time and hence delays the stream processing data flow. Spark allows data to be cached in memory and bounds the processing latency for each batch.

3. DESIGN

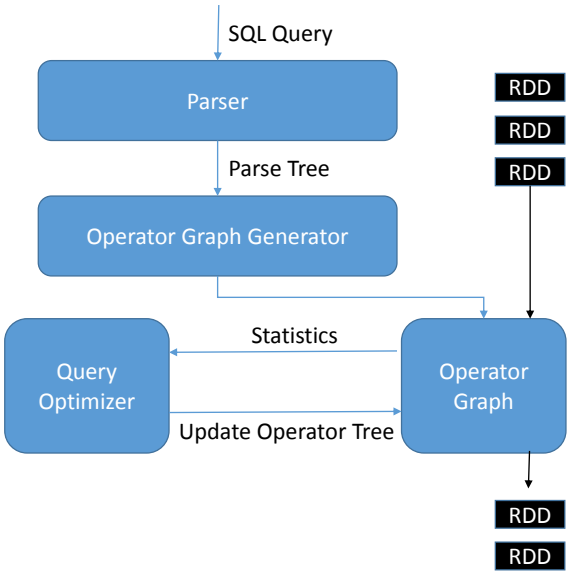

Figure 1: The architecture for dynamic query optimization.

We discuss the design of our streaming system in this section. The streaming system is composed of three components: a SQL parser, a query optimizer, and an operator graph. Queries are parsed by the parser to construct a parse tree. The parse tree is then translated into an operator tree. The operators in the operator tree collects the streams statistics while processing the streams. The statistics are feed back to the query optimizer after each batch. If necessary, the optimizer will reorganize the operator tree for more efficient execution. Figure 1 shows the architecture of our system.

3.1 Batch Processing

We process data in batch granularity instead of record granularity to improve scalability and fault tolerance. The main idea is to process streams in batches of small time intervals. All incoming data is cached in the cluster until the end of the time interval. At the end of each time interval, the cached data is stored as a RDD and is processed in a deterministic manner. A stream data query transforms input RDDs to output RDDs base on the query the user written, such as select, where, join, group by.

RDD keeps track of lineage information which describes how the data is computed. In the case of a failure, lost job can be recovered using the lineage information. For example, in Fig 2, it shows the lineage graph of a map followed by a reduce. If a map task fails, we can simply recover the data using the lineage graph by recomputing the data.

3.2 Grammar and Semantics

We adopt the grammar and semantics from SQL and extend the grammar to support window operations. In our grammar, there are three types of query; (1) input query, that parses the input to a stream of relational records (2) SQL query, that transforms a stream of relational records to another stream of records (3) an output query, that output the

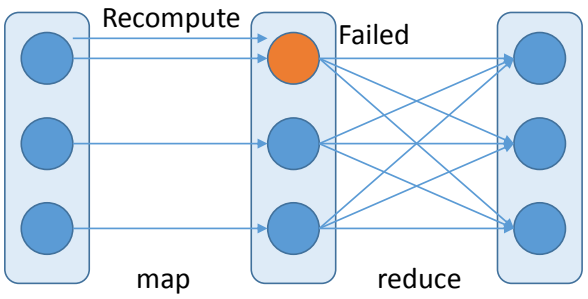

Figure 2: Lineage graph of a map and a reduce. The lost job can be simply recovered by recomputing the job.

relational records to string outputs.

Input Query: An input query parses strings in a stream to a relation. Consider the following input query.

```
twitterstream = input user_id:int, tweet:string
 from localhost:9999 delimiter ","
```

This query parses the content in the string to a user_id and the content of the tweet. The user_id is of type integer and the tweet is a string. The stream is split by a delimiter ",". The input source is a socket connection to server *localhost* at port *9999*. In general, an input query is composed in this way:

```
<stream name> = input <schema> from <data source>
delimiter <delimiter string>
```

For example, the following stream of string

```
01234, "Happy Thanksgiving"
02345, "Nice weather"
```

will be parsed to a stream of relations called *twitterstream*

user_id : Int	tweet : String
01234	"Happy Thanksgiving"
02345	"Nice weather"

Each line in the input stream is parsed according to the schema of the *twitterstream*. If one of the line in the input source is misformatted and causes a parse failure, that line will be skipped. In general, input query converts unstructured data to structured relations for further query.

SQL Query: A SQL query converts one structured relational stream to another structured relational stream. An example SQL query could be like this.

```
tg_tweets = select * from twitterstream
where tweet like '%Thanksgiving%' window 10 seconds
```

This query selects records from the *twitterstream* where the tweet content contains keyword "Thanksgiving". By adding the window clause, all records in the last 10 seconds will be retained. The result is assigned to another stream called "tg_tweets", which stands for Thanksgiving tweets. In general, a SQL query is composed in this way.

```
<stream name> = <standard SQL query> <window size>
```

A standard SQL query is applied on previous streams, the result is windowed with the given window size. The window clause is optional. If no window clause is specified, a default window size of 1 batch is applied. The name on the left side of the = sign is the identifier of the stream.

Output Query: Output query converts structured relation stream into unstructured string stream. Following is an example of an output statement:

```
output user_id, tweet from tg_tweets delimiter ","
```

This converts each relation back to a string. The user_id and tweet fields are converted to a string separated by ",".

The relation *tg_stream* is converted back to strings

user_id : Int	tweet : String
01234	"Happy Thanksgiving"

```
01234, "Happy Thanksgiving"
```

We build our own custom parser to parse the query. The query is then converted to an operator tree. Data stream flows through the operators for processing. Note that our underlying execution engine is Spark. All data is processed in batch granularity instead of record granularity.

3.3 Operators

We introduce the operators we have implemented in our system here. An operator takes a RDD as the input at the time t and outputs a RDD. A RDD may be totally stateless, such as a filter operator. The filter operator takes the input RDD on each batch interval, filter the records with a given predicate, and output the filtered RDD. Some other operators are "stateful", where some internal state is kept when processing the data stream. Note that the "states" of an operator are still RDDs, therefore, we do not need to duplicate operators for fault tolerance. Instead, when there is a failure, the "state" can be recovered by lineage graph. For example, a window operator is stateful, where input RDDs in the window are cached.

Each operator maintains an output schema that is defined when the operator object is initiated. A schema is an array of (column_id, type) pair. The column_id is a global unique identifier of that column. We can easily identify a column with column_id even if they appear in output schema of different operators. For example, when two streams are joined, the original column_id's are inherited from their parents. Sometimes we need to create new column_id. For example, consider the following query:

```
select item, price + tax as totalcost from shopping
```

While *item* still maintains the old column_id, *totalcost* will be assigned with a new column_id. With this global naming scheme, we can identify a column easily when operators are reordered at runtime. We discuss the details of the operators we implemented here.

Select Operator: Select operator filters records in the data stream with some predicate. Everytime a RDD is passed to the operator, the operator perform calls *.filter(predicate)* to filter the records in the RDD according to the predicate. The output schema of select operator is the same as its parent's output schema.

Project Operator: Project operator selects the required columns in the stream specified in the query. We implement this operator by calling *.map(func)* on the input RDD. The output schema of project operator is a subset of the output schema of its parent operator.

Join Operator: Join operator joins the records in two streams with the same join keys. In a join operator, records from the two input streams are first converted to key value pairs and then they are co-partitioned by the key. This shuffles all the data in the two input stream, and guarantees the records with the same keys are located on the same worker nodes. Since data has already been partitioned by key in the parent RDDs, the records in the corresponding partition can now be joined locally.

We collect statistics of the join operation when executing the join. The size of the two input streams, and the size of the output stream is counted. The selectivity of this operator is derived from the input and output cardinality. The statistics is collected using accumulator, that collects statistics of a RDD efficiently in an incremental manner. The output schema of the joined stream is the concatenation of the input stream schemas.

Window Operator: A window operator takes the input at the current timestamp and outputs the union of all the RDDs within the window. When a new RDD arrives the operator, we first persist it in memory. This is very important because it avoids recomputing this RDD at the future time. If the RDD is still within the window and need to be used, instead of recomputing it using lineage graph, we can simply get it from memory. The operator simply union all the RDDs in the window and output it. The output schema is the same as the input schema.

Aggregation Operator: Aggregation operators aggregates columns to a single value by a specific aggregation function (eg. SUM(), AVG()). Similar to join, aggregation operators

first partition the data by the aggregation key. The aggregation functions are then applied to the aggregation values to obtain the results. All columns that belongs to the aggregation key maintains the same *column_id*, while all aggregated columns uses a new *column_id*. The output schema is a combination of the aggregation keys and aggregation values.

Input Operator: Input operator parses the unstructured strings to construct a relation. The output schema is defined by the user as appeared in the input query.

Output Operator: Output operator converts structured data to strings separated by a delimiter. There is no output schema for this operator.

With the parse tree generated, the operator graph generator takes the parse tree to generate an operator graph. Batches of RDDs are generated by the stream receiver and the operator graph consumes the RDDs. All the RDDs generated at the same timestamp is processed by the operators. Finally, the output operators generate the processed data stream. The execution engine evaluates RDD lazily; If the stream is not output by any output operator, it will not be processed at all.

3.4 Query Optimization

Query optimization can be done efficiently in batch streaming system. Between each batch execution, there is a natural barrier to reoptimize the execution base on the statistics collected from previous iterations. This is difficult in traditional streaming systems [5, 7, 10, 12]. In traditional stream processing system, operators maintain states for incremental processing. To be fault tolerant, operators are replicated and states must be synchronized between the operators. In batch processing system, the cost of replication can be avoided by using lineage graph. All states are represented as RDDs, so they are reconstructable by recomputing the lost task. We consider applying several query optimization on the operator graph.

We implement predicate pushdown on the operator graph. Select operator can be pushed to the data source to reduce unnecessary calculation on filtered records. We implement this by swapping select operators with its parent until certain conditions:

- The select operator's parent has multiple children.
- The select operator's parent is an aggregation operator and the select predicate is on the aggregation columns.

Predicate pushdown can reduce the amount of computation if it is pushed below expensive operators such as join and aggregation.

Unlike select operators, window operators should be pushed up. Window operator unions all the RDDs in the window and outputs it. Therefore, a window operator can increase the amount of redundant computation. This is demonstrated in Figure3. If a stream is first windowed, the following project operation have to recompute certain amount of data every task. Therefore, instead of windowing

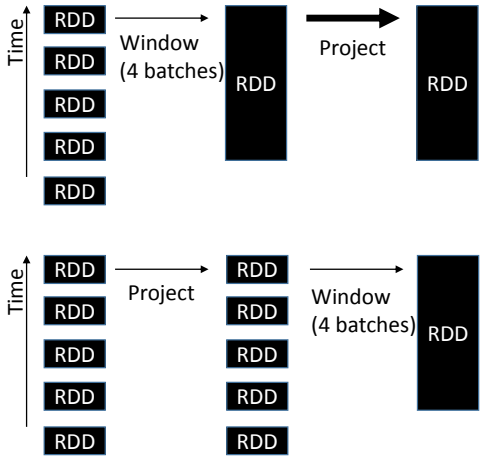

Figure 3: Window operators should be executed after other operators, if possible

the RDDs first, the project operation could be done before the window operation. The window operator simply unions the RDDs in the window, the operation cost is very low in Spark.

We should avoid pushing up window operators in the following condition to guarantee the semantic correctness.

- If there are multiple children for the window operator. In this case, we could split the window operator and push them to all children.
- If the child operator is an aggregation operator, the window size affects the aggregation result.
- Similarly, the window operator affects join result, so we should avoid pushing it below join operator.

Join operators could be reordered online base on the statistics we collected. Ideally, join operator with small selectivity should be applied first (Figure 4). We obtain the selectivity of join operator by estimating the input data size and output data size. These statistics are feedback to the query optimizer right away. The join optimizer reorders the operator according to the selectivity of the operators. Note that traffic pattern is usually static in a short time period. Therefore, it is unnecessary to obtain statistics every batch. Instead, we could estimate the statistics every several batches. We modify the query graph only if we are confident that the data characteristics have been changed.

We implement incremental aggregation to reduce the amount of redundant computation. As we have mentioned, window operators could not be pushed beyond aggregation operators. However, we can merge the window and aggregation operator such that only the new batch is aggregated. Consider the following query:

```
windowed = select topic, tweet from twitter_stream
```

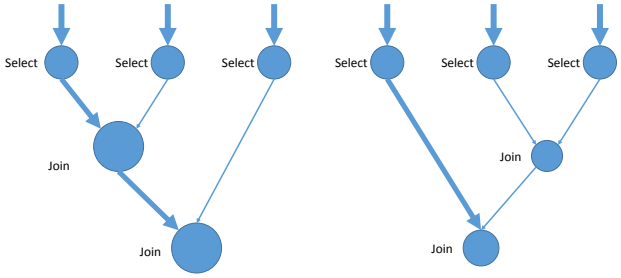

Figure 4: Reorder join operators by selectivity

```
window 300 seconds
topic_count = select topic, count(*) from windowed
```

By default, the last 300 seconds data will be windowed and counted repeatedly. The optimized version would perform a two level aggregation; Each batch in the time window is aggregated and cached. The aggregated value is being aggregated again as the final result. This method reduces the amount of redundant computation by using the cached aggregation value in the window.

The join operator can also be implemented incrementally. Consider joining two streams X and Y to stream W: let $X(t)$ denotes the records at time t. $X^+(t+1) = X(t+1) - X(t)$ and $X^-(t+1) = X(t) - X(t+1)$ are the records that added to and remove from X at time $t+1$. Then

$$W(t+1) = W(t) + W^+(t+1) - W^-(t+1)$$

Variable $W^+(t+1)$ can be obtained by

$$W^+(t+1) = Join(X^+(t+1), Y(t+1)) \\ +Join(Y^+(t+1), X(t) - X^-(t+1))$$

Similarly, variable $W^-(t+1)$ can be obtained by

$$W^-(t+1) = Join(X^-(t+1), Y(t)) \\ +Join(Y^-(t+1), X(t) - X^-(t+1))$$

Although the equations looks complicated, the semantic is quite simple; Everytime the time advances, we remove the records that are expired in the join results and add the new join results. We are currently studying how to efficiently implement incremental join. The current problem is Spark can only manipulate data at RDD level, record level modification may not be efficient.

4. EVALUATION

We evaluate the performance of the query optimization technique in Amazon EC2. We set up a cluster with 6 worker nodes using Spark 0.9.0 and deployed our system on top of it.

4.1 Micro Benchmark

We first evaluate the benefits of predicate pushdown. The query we execute is:

```
x = input key:int,value:int from localhost:9999
```

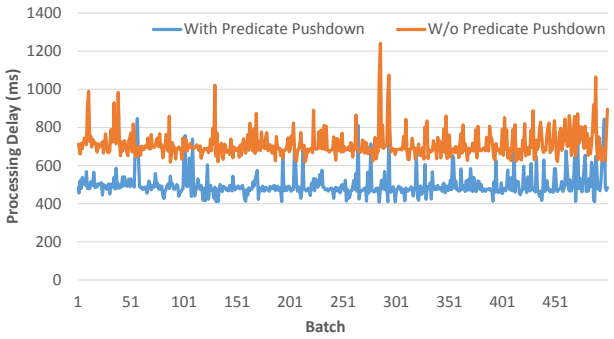

Figure 5: Processing delay with and without predicate pushdown

```
    delimiter ","
gb = select key, avg(value) as value from x
     group by key
pred = select key, value from gb where key > 10
output key, value from pred delimiter ","
```

The source stream generates 500,000 key value pairs per second. The key is generated from a random Gaussian distribution with mean equals to 0 and a standard deviation of 5. The predicate $key > 10$ filters about 97.5% records out. Without predicate pushdown, the group by operator is evaluated before the predicate. We evaluate 500 batches of data with a batch size of 1 second. Without predicate pushing, the average processing latency is 712ms. After predicate pushing, the average latency is only 497ms, which reduces the processing latency by 30%.

We evaluate the performance of window pushup using the same experimental setting. The cluster processes 500,000 records every batch, with batch size 1 second. Following is the query we executed:

```
x = input key:int,value:int from localhost:9999
    delimiter ","
win = select key, value from x window 10 seconds
pred = select key from win where key = 20
output key from pred delimiter ","
```

Even with very small window size of 10 seconds, it takes over 2 seconds to process 1 second worth of data without optimization (Figure 6 (A)). Since the processing rate is lower than the incoming data rate, the tasks soon queued up and the batch delay goes up to infinity. By using window pushup, the processing latency of a batch can be maintained at a low level (Figure 6 (B)). We reduce the data rate to 200,000 records per second and reevaluate the performance. By using window pushup, we can reduce the processing latency by 65%.

To evaluate the performance of dynamic join operator reordering, we execute the following query:

```
x = input key:int,value:int from localhost:9999
```

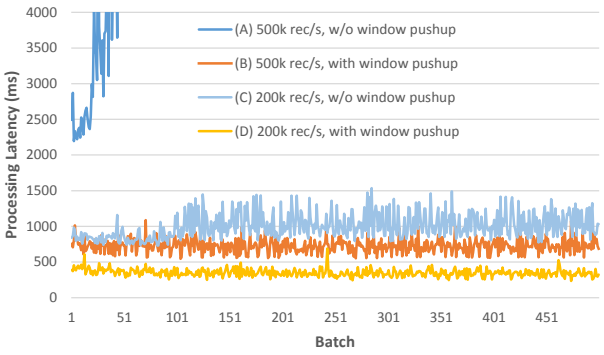

Figure 6: Processing delay with and without window pushup

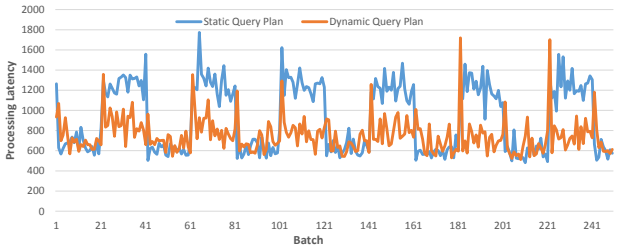

Figure 7: Join with/without operator reordering

```
    delimiter ","
y = input key:int,value:int from localhost:9998
    delimiter ","
z = input key:int,value:int from localhost:9997
    delimiter ","
joined = select x.key as xk, x.value as xv,
         y.key as yk, y.value as yv,
         z.key as zk, z.value as zv from x
         inner join y on x.key = y.key
         inner join z on x.key = z.key
ct = select count(xk) as c from joined
output c from ct delimiter ","
```

The key of streams y and z are generated from a Gaussian distribution with mean 0 and 30, respectively. The key of stream x is generated from a Gaussian Distribution with varying mean; Every 20 seconds, the mean of x switches between 0 and 30. Therefore, when x's mean is 0, we should join x and z first, and the join the result with y. The same rule applies when x's mean is 30. For all the three streams x, y and z, the standard deviation is 10 and the data rate is 2000 records per second. The *joined* stream produces around 400,000 records per second.

Figure 7 shows the processing latency of the execution with and without join operator reordering. With a static query plan, the latency follows a bimodal distribution, with an average latency of 921 ms. With dynamic query optimization, the latency distribution is more uniform, and the average latency is 748 ms. We observe that every time the mean of stream x switches, there is a spike on the latency even if the dynamic query optimization is applied. This is because

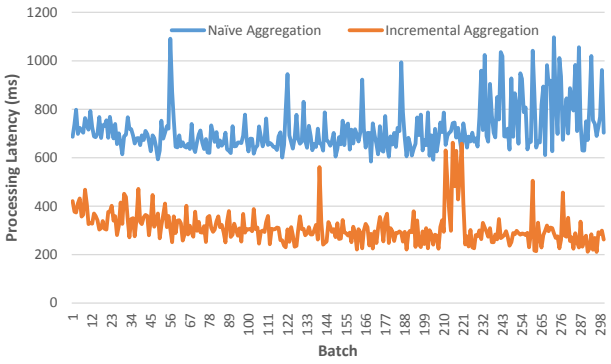

Figure 8: Processing delay using naïve aggregation and incremental aggregation

there is a one batch delay when obtaining the statistics. In practice, the stream characteristics changes less abruptly, hence, the processing latency would be more stable.

Window operators can only be pushed upto an aggregation operator. Further pushing the window operator above aggregation operator changes the semantic. However, by replacing the aggregation operator with an incremental aggregation operator, we can efficiently execute the aggregation without doing redundant work. We evaluate the following query:

```
x = input key:int,value:int from localhost:9999
    delimiter ","
xx = select key,vallue from x window 10 seconds
ct = select key, count(value) as count from xx
output key,count from ct delimiter ","
```

The *key*s in stream x are generated from a Gaussian Distribution with mean 0 and standard deviation 5. The data rate of stream x is 100,000 records per second. Figure 8 shows the processing delay using naïve aggregation and incremental aggregation. By using incremental aggregation, we can reduce the average processing delay from 715 ms to 310 ms.

We evaluate the performance of incremental join with following query:

```
x = input key:int,value:int from localhost:9999
    delimiter ","
y = input key:int,value:int from localhost:9998
    delimiter ","
xx = select key,value from x window 10 seconds
yy = select key,value from y window 10 seconds
joined = select xx.key as xk, xx.value as xv,
         yy.key as yk, yy.value as yv from xx
         inner join yy on xx.key = yy.value
ct = select count(xk) as count from joined
output count from ct delimiter ","
```

Window operators are pushed up to the join operator. The join operator are then replaced with an incremental join operator. The incremental join operator caches the join result

in the window. Figure 9 shows the processing latency of a naïve join and an incremental join. The incremental join operator checkpoints the data every 10 seconds. Therefore, the processing delay spikes every 10 seconds. Figure 10 shows the performance with varying window size. For window size varying from 10 seconds to 40 seconds, incremental join operator always performs better.

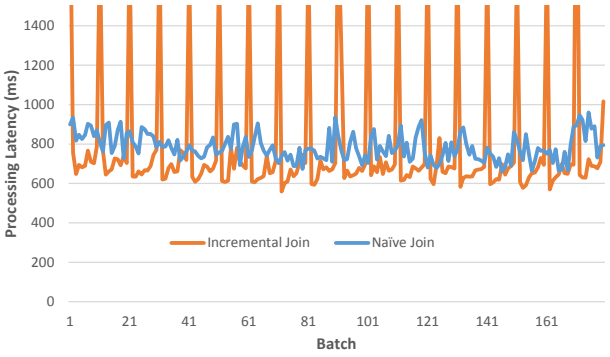

Figure 9: Processing delay using naïve join and incremental join

4.2 Road Traffic Benchmark

Linear Road Benchmark [6] is used in the STREAM [18] project. The benchmark has the locations of the cars in a simulated freeway system. The cars in the freeway system reports the location of the car to a central repository. Following is the schema of the data:

1. **time** The timestamp when the record enters the database.
2. **vid** An unique identification of the vehicle.
3. **speed** The speed of the vehicle.
4. **hwy** The highway ID. There are multiple highways in the system.
5. **lane** The lane the vehicle is on.
6. **dir** The direction the vehicle is going.
7. **seg** The segment the vehicle is in. The highway is divided into 1 mile segments.
8. **pos** The location the (in feet) the vehicle is from the starting point of the highway.

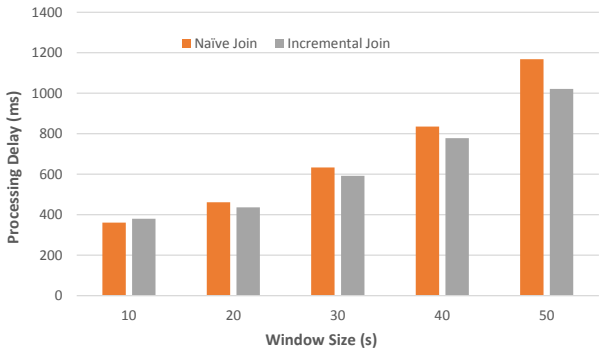

Figure 10: Average processing delay using naïve join and incremental join with varying window size

The data is three hours long with varying incoming data rates from 5 records per second to 1884 records per second. To increase the workload, we replay the data 5 times faster. The goal of the query is to count the number of cars that are in the congested segments (average car speed < 40mph in the last minute) in the last 15 seconds. Following is the query:

```
records = input time:int, vid:int, speed:int, hwy:int,
          lane:int, dir:string, seg:string, pos:string
          from localhost:8888 delimiter ","
recWindow60 = select vid, speed, seg, dir, hwy
          from records window 60 seconds
segSpeed = select seg, dir, hwy, avg(speed) as avgSpeed
          from recWindow60 group by seg, dir, hwy
congestedSeg = select seg, dir, hwy, avgSpeed
          from segSpeed where avgSpeed < 40
recWindow15 = select vid, speed, seg, dir, hwy
          from records window 15 seconds
congestedCar = select recWindow15.vid as vid
          from recWindow15 inner join congestedSeg
          on recWindow15.seg = congestedSeg.seg
          and recWindow15.dir = congestedSeg.dir
          and recWindow15.hwy = congestedSeg.hwy
numCongested = select count(vid) as c
          from congestedCar
output c from numCongested delimiter ","
```

Stream *records* are the raw input records with the location of the vehicle and speed of the car. *recWindow10* is a windowed stream with the location and speed of the vehicle in the last. *segSpeed* is a stream storing the average speed of the vehicles in a segment. *congestedSeg* is the segments that are congested (i.e. avgSpeed < 40). *recWindow3* is a windowed stream with the location and speed of the vehicle in the last 15 seconds. By joining *congestedSeg* and *recWindow15*, we obtain *congestedCar*, the cars that are in the congested segments. *numCongested* counts the number of cars in the congested segments.

The optimizer automatically applies predicate pushdown, window pushup and incremental aggregation in this query. Figure 11 shows the results of the experiment. At the beginning, both approaches have similar performance as the input data rate is small. With the increase of input data rate, the query plan without query optimization has larger latency than the optimized query plan. The average execution latency of the query plan without optimization is 960 ms with a standard deviation of 445 ms. The optimized query plan has an average latency of 679 ms and the standard deviation is 250 ms.

4.3 Twitter Dataset

We use the Twitter API to sample small fraction of Tweets and dump them to a local file. We replay the Tweets with

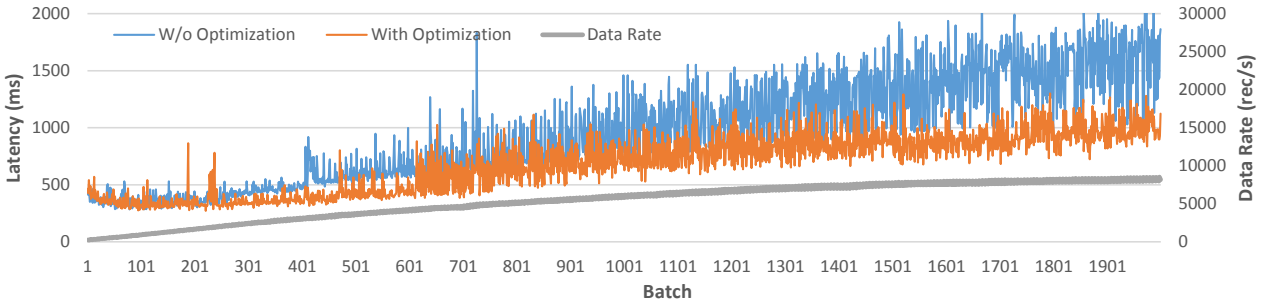

Figure 11: Executing Query with/without optimization on Linear Road Benchmark

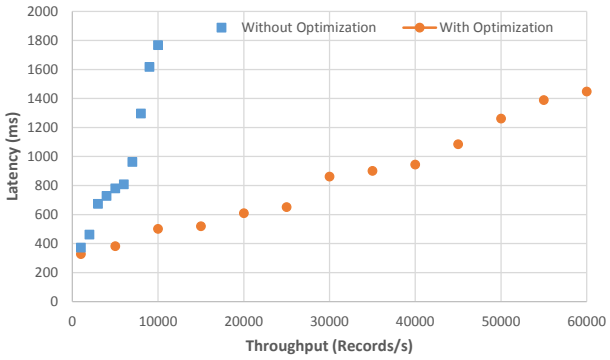

Figure 12: Throughput vs Latency. The blue squares are from the query plan without optimization. The orange dots are from optmized query plan.

varying data rate to measure the throughput of the query plan. We evaluate the following query:

```
raw = input time:string, tweetId:string,
    scrName:string, language:string, userId:string,
    userLoc:string, userName:string, text:string
    from localhost:7777 delimiter "\t"
win = select lang from raw window 10 seconds
g = select language, count() as c from win
    group by language
output language, c from g delimiter ","
```

Figure 12 shows the evaluation results. Without query optimization, latency grows quickly with the increase of throughput. To maintain a latency bound of 1 second, the query plan without optimization can only process about 7,000 records per second. With optimization, the throughput can be as high as 40,000 records per second.

5. RELATED WORK

Stream Processing Systems: Stream processing engines, such as STREAM [18], TelegraphCQ [10], TimeStream [22], Storm [1], Naiad [19], and Borealis [2] are all continuous operator model. The data flow is expressed as a graph of operators. Data is processed by operators and sent to other operators for further processing. In fact, in these systems, batching is a common technique to increase the performance of stream processing. In these systems, operators maintain states. When an operator fails, the state of the operator is gone. To reconstruct the state, either the operator needs to be duplicated or the upstream backup needs to be applied. In DStream, operator is decoupled from operator state by keeping the lineage information. Therefore, lost state can be simply recovered by recomputing the lost partition. All these stream process system applies a static execution graph. Our system allows the operators to be reordered base on the execution information from previous batches. Statistics of the stream data is collected from executed batches, and these data is applied to optimize the execution of new batches.

Adaptive Streaming: Existing papers have extensive discussion on optimizing query execution in traditional record-based stream process engine. [23] discusses how to place the operators in the query plan. When the processing capacity is not enough to processing the incoming data, some fraction of the data needs to be discarded [8,24]. While this technique guarantees the processing time, it is lossy in nature and the quality of the query may be sacrificed. Other papers have proposed to elastically adjust the amount of resource according to the incoming workload [4,16,17,25]. We would like to study how to adjust the degree of parallelism in the data processing flow in response to the varying incoming data rate. Deciding the optimal degree of parallelism is a difficult question. Making the degree of parallelism too small increases the processing time of the tasks, while making it too large increases the task launching overhead. We will incorporate this in our future work.

Query Optimization: Optimizing query execution is a well studied topic in database management system [11, 13, 14]. To estimate the cost of an operator, sampling is necessary to obtain the statistics of the data. However, sampling is costly, especially for "big data". A recent proposal [3] argues that if a task is executed on the same data again, statistics can be obtained from the first execution. In our system, although data between batches is not the same, they often show similar statistical properties. Therefore, we could collect data statistics from previous executions to optimize future execution. In this paper, we implement this idea and show the benefits of optimizing the execution online.

6. CONCLUSION

We presented several techniques to optimize query execution in batch streaming system. We implemented incremental operators and a query optimization framework that dynamically reoptimize the data parallel execution on the fly. We show the benefits of such optimization with micro benchmark and real dataset. In the future work, we would like to investigate elastically adjust the parallelism of the operators to provide performance guarantee in batch streaming system.

7. REFERENCES

[1] Storm, distributed and fault-tolerant realtime computation.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[3] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *Proc. of USENIX NSDI*, pages 21–35, 2012.

[4] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 71–71. IEEE, 2006.

[5] A. Arasu, S. Babu, and J. Widom. Cql: A language for continuous queries over streams and relations. In *Database Programming Languages*, pages 1–19. Springer, 2004.

[6] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491. VLDB Endowment, 2004.

[7] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *ACM sigmod record*, volume 29, pages 261–272. ACM, 2000.

[8] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 350–361. IEEE, 2004.

[9] D. Borthakur. The hadoop distributed file system: Architecture and design, 2007.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[11] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pages 190–200. IEEE, 1995.

[12] N. Conway, M. J. Franklin, S. Krishnamurthy, A. Li, A. Russakovsky, and N. Thombre. Trusql: A stream-relational extension to sql.

[13] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[14] J. C. Freytag. *A rule-based view of query optimization*, volume 16. ACM, 1987.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[16] A. Ishii and T. Suzumura. Elastic stream computing with clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 195–202. IEEE, 2011.

[17] A. Jain, E. Y. Chang, and Y.-F. Wang. Adaptive stream resource management using kalman filters. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 11–22. ACM, 2004.

[18] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. CIDR, 2003.

[19] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[20] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[21] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[22] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14. ACM, 2013.

[23] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 250–258. ACM, 2005.

[24] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 309–320. VLDB Endowment, 2003.

[25] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 791–802. IEEE, 2005.

[26] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.