2012 IEEE 28th International Conference on Data Engineering

M^3: Stream Processing on Main-Memory MapReduce

Ahmed M. Aly*, Asmaa Sallam*, Bala M. Gnanasekaran*, Long-Van Nguyen-Dinh*,
Walid G. Aref*, Mourad Ouzzani†, Arif Ghafoor*

*Purdue University, West Lafayette, IN, USA
†Qatar Computing Research Institute, Qatar Foundation, Qatar

***Abstract*—The continuous growth of social web applications along with the development of sensor capabilities in electronic devices is creating countless opportunities to analyze the enormous amounts of data that is continuously steaming from these applications and devices. To process large scale data on large scale computing clusters, MapReduce has been introduced as a framework for parallel computing. However, most of the current implementations of the MapReduce framework support only the execution of fixed-input jobs. Such restriction makes these implementations inapplicable for most streaming applications, in which queries are continuous in nature, and input data streams are continuously received at high arrival rates. In this demonstration, we showcase M^3, a prototype implementation of the MapReduce framework in which continuous queries over streams of data can be efficiently answered. M^3 extends Hadoop, the open source implementation of MapReduce, bypassing the Hadoop Distributed File System (HDFS) to support main-memory-only processing. Moreover, M^3 supports *continuous* execution of the Map and Reduce phases where individual Mappers and Reducers never terminate.**

I. Introduction

In recent years, the MapReduce framework [1] has proved to be successful in processing large datasets on large clusters of machines, particularly after the massive deployment reported by companies like Facebook, Google, and Yahoo!. However, most of the current implementations of the MapReduce framework target batch processing where MapReduce jobs operate on fixed input data that is usually stored in some file system integrated into the framework. For instance, in Hadoop [2], input data to a MapReduce job is stored in the Hadoop Distributed File System (HDFS) before the job is run. While acceptable for batch processing applications, HDFS introduces significant disk delays that make Hadoop inapplicable for streaming applications in which input streams are received at high arrival rates.

In [3], [4], *MapReduce Online* is introduced to support continuous query processing on MapReduce. However, since it is based on HDFS, it is not suitable for streaming applications, in which data streams have to be processed without any disk involvement. Moreover, MapReduce Online requires that each set of input is processed through a fresh MapReduce job, in which new Mappers and Reducers are invoked. This, implies significant setup-time overhead for the multiple jobs being invoked.

To support the processing of continuous queries over large scale streaming data, we introduce M^3, a main-memory implementation of MapReduce, based on Hadoop. In M^3 data gets processed only through a main-memory-only data-path. Moreover, Mappers and Reducers never terminate, hence, there is only one MapReduce job per query operator that is continuously executing. M^3 is fault tolerant and incrementally evaluates continuous queries. It also extends the SQL interface of Hive [5], providing a way of issuing continuous streaming queries on MapReduce.

II. Stream Processing on MapReduce

A straightforward implementation of a streaming application using Hadoop is shown in Fig. 1a. Input data streams accumulate into HDFS up to a certain size, at which a MapReduce job is started to process the accumulated data. The Map phase starts with all the Mappers reading input splits from HDFS, processing the data, and finally writing intermediate key-value pairs into local disks. The Reduce phase starts with all reducers reading the intermediate key-value pairs from local disks, processing these pairs, and then finally writing the output into HDFS. Afterwards, a fresh MapReduce job is started for the next accumulated data.

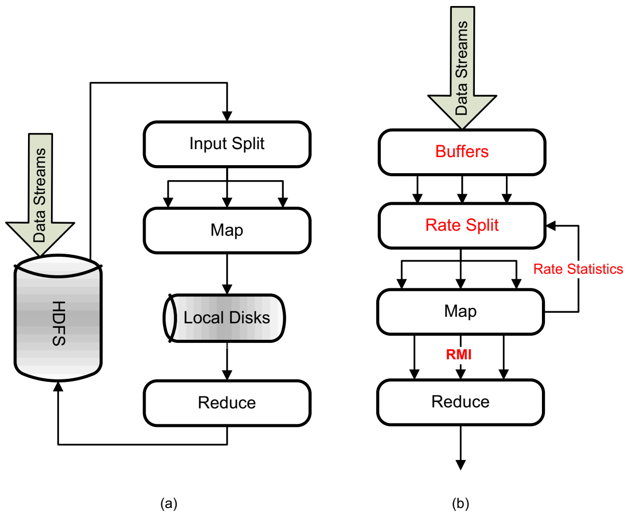

Fig. 1. (a) Dataflow in Hadoop. (b) Dataflow in M^3.

With such multiple reads/writes from/to HDFS as well as local disks, the architecture of Hadoop is inapplicable to most streaming applications. Therefore M^3 extends Hadoop as in Fig. 1b such that data processing is performed only through main-memory, avoiding HDFS and local disk access. In M^3, direct communication between Mappers and Reducers takes place using Java *Remote Method Invocation* (RMI), which is a main-memory communication protocol.

Query processing in M^3 is incremental [6], [7], i.e., only the new input is processed, and the change in the query answer is represented by three sets of inserted (+ve), deleted (−ve), and updated (u) tuples. The query issuer receives as output a stream that represents the deltas (incremental changes) to the answer. Whenever an input tuple is received, it is transformed into a *modify* operation (+ve, −ve, or u) that propagates in the query execution pipeline, producing the corresponding set of modify operations in the answer.

Supporting incremental query evaluation requires that some intermediate state be kept at the various operators of the query execution pipeline. For instance, consider the query: *"Continuously monitor the identifiers of cars that are in the parking lot"*. At every point in time, the execution of that query has to store the current number of cars that are in the parking lot so it can incrementally modify that number according to the updates received from the streams. If Hadoop is to be used as in Fig. 1a, then that intermediate state has to be stored into HDFS before a MapReduce job completes, so that the next MapReduce job can read that state back from HDFS. However, in Hadoop, Mappers and Reducers terminate after the job finishes, leaving no chance to maintain any state in main-memory. In M^3, Mappers and Reducers run continuously without termintation, and hence can maintain main-memory state throughout the execution.

III. Features of M^3

The main features M^3 are summarized as follows:

- ***Main-memory-only data flow.*** Throughout the query execution, data flows only through main-memory, totally avoiding any disk access. Input streams are temporarily stored into main-memory buffers. Moreover, Mappers and Reducers directly communicate through Java RMI.
- ***Continuity of MapReduce jobs.*** In M^3, there is no need to issue a separate MapReduce job per batch of stream updates since Mappers and Reducers never terminate. In consequence, operators' states are kept in main-memory, supporting incremental evaluation.
- ***Rate-based split.*** In contrast to splitting the input data based on its size as in Hadoops *Input Split* functionality, M^3 splits the streamed data based on arrival rates. The *Rate Split* layer, between the main-memory buffers and the Mappers is responsible for balancing the stream rates among the Mappers. This layer periodically receives rate statistics from the Mappers and accordingly redistributes the load of processing amongst Mappers. For instance, a fast stream that can overflow one Mapper should be distributed among two or more Mappers. In contrast, a group of slow streams that would underflow their corresponding Mappers should be combined to feed into only one Mapper.
- ***Fault tolerance.*** In M^3, each Mapper and Reducer has two states: *running* and *commited*. To support fault tolerance, input data is replicated inside the main memory buffers and an input split is not overwritten until the corresponding Mapper commits. When a Mapper fails, it re-reads its corresponding input split from any of the replica inside the buffers. A Mapper writes its intermediate key-value pairs in its own main-memory, and does not overwrite a set of key-value pairs until the corresponding reducer commits. When a reducer fails, it re-reads its corresponding sets of intermediate key-value pairs from the Mappers. Also the state of the execution (which is stored with the Reducers as we will show next) is replicated across more than one Reducers. When a Reducer fails, it reconstructs its state from the corresponding replica.
- ***Support of incremental processing.*** In M^3, Query processing in M^3 is incremental [6], [7], i.e., only the new input is processed.
- ***Support of multi-operator query plans.*** In M^3, a query is compiled into a tree of MapReduce jobs (a job per query operator), with the output of a job directly connected to the input of its parent. The leaves of the tree plan have the streams as input and the root of the tree produces the output of the query.
- ***Support for general window predicates.*** M^3 supports both predicate [8], [6] and sliding window predicates. Consider for example the query: *"Continuously monitor the identifiers of cars that were in the parking lot within the last 5 minutes"*; a time-based window query in which the window of interest is limited to the last 5 minutes. Consider another query: *"Continuously monitor the identifiers of cars that are exceeding the speed limit (50 mph)"*, in which the window of the query is based on a predicate on the speeds of cars. For more details on window predicates, the reader is referred to [9], [8], [6].
- ***Support of SyncSQL through Hive.*** Hive [5] provides an interface to transform SQL queries into MapReduce jobs. M^3 extends the syntax of Hive to support continuous streaming queries on data streams. The extended SQL query language is based on *SyncSQL* [6].

IV. Continuous Query Processing in M^3

In this Section, we show how continuous streaming queries are incrementally evaluated inside M^3. We show the operation the Join operator as a representative for all binary operators.

A. Stream Tagging

M^3 supports incremental evaluation of streaming queries by using a *Tagger* operator [6] for each input stream. Typically, the Tagger operators would be at the leaves of a query plan to tag and filter the tuples of the input streams. A Tagger operator ensures the proper assignment of tags (signs) to tuples (e.g., +ve, −ve, or update (u)). For example, if the input tuple corresponds to an object that has appeared before, the Tagger operator would assign a u sign (for update) to indicate that this tuple corresponds to an object that has been previously processed. In M^3, the query compiler pushes the window predicates inside the Tagger operators to filter unqualified tuples early on.

Consider the previous query: *"Continuously monitor the identifiers of cars that are exceeding the speed limit (50 mph)"*. This query can be expressed using SyncSQL as follows:

```
CREATE STREAMED VIEW SpeedyCarsView AS
CSELECT STREAMED CarID, Speed
FROM CarsSpeedStream CSS
WHERE CSS.Speed > 50;
```

M^3 creates a view for the above query so that its streamed output can be used as input to another query (as we show in the next subsections). The intermediate key between the Map and Reduce phases is set to be $CarID$, i.e., the object ID. This implies that each Reducer will be responsible for a set of the objects and that set never changes throughout the execution. Each Reducer keeps a main-memory hash table for the objects that satisfy a given window predicate. For example, in the above query, the Tagger operator will store the IDs of the cars that are exceeding the speed limit, along with their speeds.

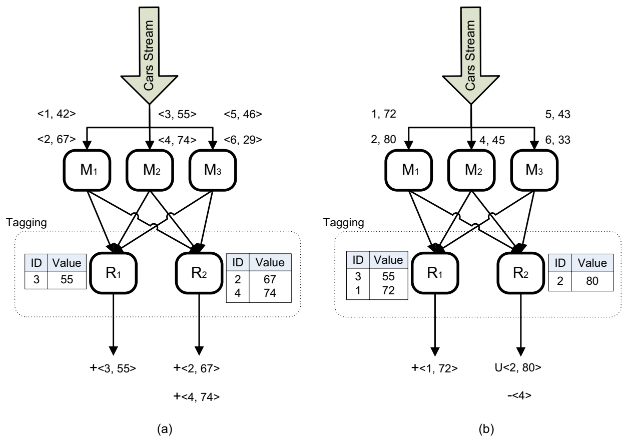

Fig. 2. Stream Tagging. (a) Execution at time t. (b) Execution at time $t+1$.

Fig. 2 shows the MapReduce job for the above query. The input and output tuple format is $< CarID, Speed >$. Three Mappers M_1, M_2, and M_3 and two Reducers R_1 and R_2 are running. R_1 is designated to receive the odd IDs while R_2 is designated to receive the even IDs. The filter $Speed > 50$ is applied at each Reducer to pass only cars with $Speed$ greater than 50. At time t, R_1 produces $+ < 3, 55 >$ as the only speeding car R_1 encounters. The sign of the output tuple is $+$ as ID 3 was not in the state of R_1 at the time it is received. R_1 records in its hash table that it had encountered ID 3 with $Speed$ 55. Similarly, R_2 produces $+ < 2, 67 >$ and $+ < 4, 74 >$ and records in its hash table that IDs 2 and 4 have speeds 67 and 74, respectively.

At time $t + 1$, R_1 receives an update for the speed of ID 1 to be 72, so it produces $+ < 1, 72 >$. R_2 detects that the speed of ID 4 drops below the speed limit, so R_2 removes ID 4 from the hash table and outputs $- < 4 >$. R_2 also detects a change in the speed of ID 2, which happens to be still above the speed limit, so R_2 produces an update tuple $u < 2, 80 >$ and updates that value in the hash table.

B. Stream Joins

In M^3, joins among streams are supported using Repartition Join [10], in which the intermediate key between Map and Reduce is set to be the same as the join key.

Consider the query: *"Continuously monitor the identifiers of cars on Highway I-65 that are exceeding the speed limit (50 mph)"*. The query can be expressed using SyncSQL as:

```
CREATE STREAMED VIEW SpeedyI-65View AS
CSELECT STREAMED CarID
FROM SpeedyCarsView SC, I-65Stream S65
WHERE SC.ID == S65.ID;
```

This query joins the stream of cars on interstate highway I-65 with the speedy-cars view, S_c, discussed in Section IV-A. A MapReduce job is initiated for the above query in which the intermediate key between the Map and Reduce phases is set to be $CarID$. This implies that IDs from different streams that should join would go to the same Reducer all the time. Each Reducer keeps three main-memory states (hash tables), the first is the join state, the second is the S_{65} state, and the third is S_c state.

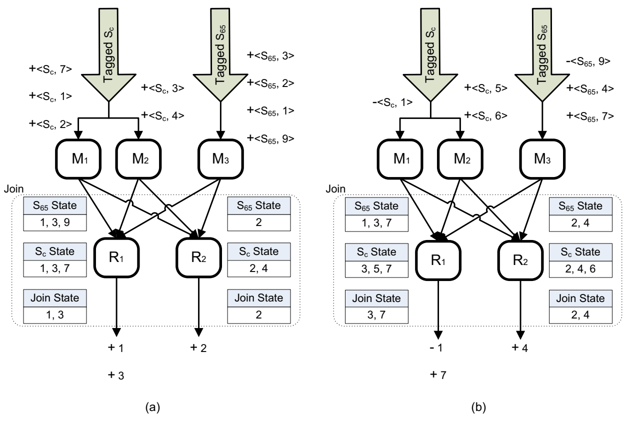

Fig. 3. Joining streams. (a) Execution at time t. (b) Execution at time $t+1$.

Fig. 3 shows the MapReduce job for the above query. The input format is $Sign < StreamID, CarID >$. The output format is $Sign < CarID >$. Three Mappers M_1, M_2, and M_3 and two Reducers R_1 and R_2 are running. R_1 is receiving the odd IDs (from both streams), while R_2 is receiving the even IDs.

At time t, R_1 produces $+1$ and $+3$ since IDs 1 and 3 are received from both input streams. Also R_1 records 1 and 3 in its three states. IDs 9 and 7 from streams S_{65} and S_c respectively are also recorded in the corresponding states. Similarly, R_2 produces $+2$ and adds ID 2 to the three states, and adds ID 4 to the state of S_c.

At time $t+1$, R_1 receives a negative update for ID 1 from S_c, so, it removes it from both the join and S_c states, and produces -1. It also produces $+7$ in response to receiving $+ < S_{65}, 7 >$, as ID 7 was in the state of S_c at time t. Similarly, R_2 produces $+4$ after receiving $+ < S_{65}, 4 >$.

C. Stream Aggregates

In M^3, grouping the output of a query is achieved through a separate MapReduce job in which the intermediate key between the Map and Reduce phases is set to be the same as the grouping key. Each Reducer keeps a state of the local aggregates (SUM, AVERAGE, ...) for the grouping keys it is responsible for. The *HAVING* condition is applied at each Reducer.

Consider the query: *"Continuously monitor how many cars on Highway I-65 are exceeding the speed limit (50 mph), grouped by the car make. Report the total number of speeding cars only if the count per car make is* > 2*"*. The query can be expressed using SyncSQL as:

```
CSELECT STREAMED Make, Count(CarID)
FROM SpeedyI-65View
GROUP BY Make
HAVING Count(CarID) > 2;
```

M^3 initiates a MapReduce job for that query, in which the intermediate key between the Map and Reduce phases is set to be the car make. Each Reducer keeps the count of the car make(s) it is responsible for.

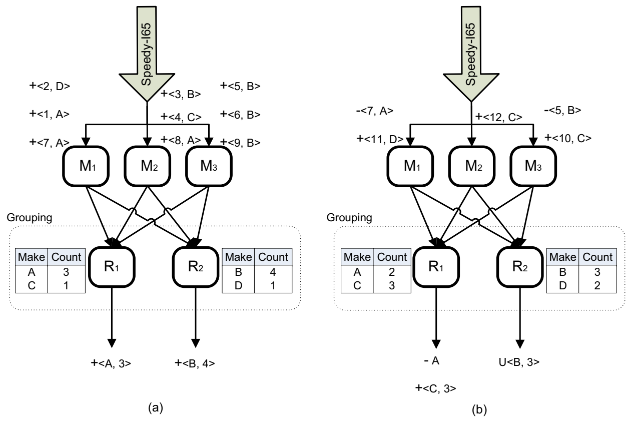

Fig. 4. Group By. (a) Execution at time t. (b) Execution at time $t+1$.

Fig. 4 shows the MapReduce job for the above query. The input format is $Sign < CarID, Make >$. The output format is $Sign < Make, Count >$ Three Mappers M_1, M_2, and M_3 and two Reducers R_1 and R_2 are running. R_1 handles car makes A and C, while R_2 handles car makes B and D.

At time t, R_1 produces $+ < A, 3 >$ as the sum of the cars received of car make A is exceeding 2. Since the number of car make C cars received so far is less than 2, R_1 reports nothing for car make C. R_1 also records in its state the counts of car makes A and C as 3 and 1 respectively. Similarly R_2 produces $+ < B, 4 >$ and records in its state the counts of car makes B and D.

At time $t+1$, R_1 detects that the count of car make A drops to 2, i.e., A no longer qualifies the *HAVING* condition, so, it produces $- < A >$. R_1 also detects that the count of car make C jumps to 3, and hence qualifies the *HAVING* condition, so, it produces $+ < C, 3 >$. Similarly, R_2 detects that the count of make B cars changed to 3, yet still qualifies the *HAVING* condition, so it produces the update tuple $u < B, 3 >$.

V. Demo Scenario

We will demonstrate M^3 and showcase its performance. More specifically, we will show how M^3 performs when processing continuous queries over streaming data that arrive at a wide variety of stream rates. We will use streams of spatio-temporal data, generated using BerlinMOD [11], in which 2 million cars that are making about 300 million trips over 28 days report their location updates in the city of Berlin. We will compare M^3 against Hadoop, and also demonstrate the functionalities and performance gains the various components of M^3 can achieve. We will show how the main-memory-only data path of M^3 can support higher stream rates, when compared to Hadoop.

Acknowledgment

This research is supported in part by QCRI, and the National Science Foundation under Grants III-1117766, IIS-0964639, and IIS-0811954.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.
[2] "Apache hadoop: http://hadoop.apache.org/."
[3] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears, "Online aggregation and continuous query support in mapreduce," in *SIGMOD Conference*, 2010, pp. 1115–1118.
[4] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *NSDI*, 2010, pp. 313–328.
[5] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - a warehousing solution over a map-reduce framework," *PVLDB*, 2009.
[6] T. M. Ghanem, A. K. Elmagarmid, P. Å. Larson, and W. G. Aref, "Supporting views in data stream management systems," *ACM Trans. Database Syst.*, 2010.
[7] T. M. Ghanem, M. A. Hammad, M. F. Mokbel, W. G. Aref, and A. K. Elmagarmid, "Incremental evaluation of sliding-window queries over data streams," *IEEE Trans. Knowl. Data Eng.*, 2007.
[8] T. M. Ghanem, W. G. Aref, and A. K. Elmagarmid, "Exploiting predicate-window semantics over data streams," *SIGMOD Record*, 2006.
[9] W. G. Aref, "Window-based query processing," in *Encyclopedia of Database Systems*, 2009.
[10] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," in *SIGMOD Conference*, 2010.
[11] C. Düntgen, T. Behr, and R. H. Güting, "Berlinmod: a benchmark for moving object databases," *VLDB J.*, 2009.