Scalable Complex Event Processing on Top of MapReduce

Jiaxue Yang, Yu Gu, Yubin Bao, and Ge Yu

Northeastern University, China
Yang.Jiaxue.Micheal@gmail.com, {guyu,baoyubin,yuge}@ise.neu.edu.cn

Abstract. In this paper, we propose a complex event processing framework on top of MapReduce, which may be widely used in many fields, such as the RFID monitoring and tracking, the intrusion detection and so on. In our framework, data collectors collect events and upload them to distributed file systems asynchronously. Then the MapReduce programming model is utilized to detect and identify events in parallel. Meanwhile, our framework also supports continuous queries over event streams by the cache mechanism. In order to reduce the delay of detecting and processing events, we replace the merge-sort phase in MapReduce tasks with hybrid sort. Also, the results can be responded in the real-time manner to users using the feedback mechanism. The feasibility and efficiency of our proposed framework are verified by the experiments.

1 Introduction

With the rapid development of various monitoring devices, simple event processing can not meet the requirements of continuous tracking and decision-making. Therefore, real-time analysis and processing over a series of events are desired in many applications. In recent years, complex event processing over streams is becoming a significant technique which needs to be urgently developed.

Most of the existing complex event processing *(CEP)* systems are centralized which transfer events collected by devices to a single node. The node utilizes the sliding window and automata model to identify and match events. Obviously, their processing ability and memory capacity are very limited. When facing massive source data, they may break down frequently. Some distributed systems have solved the problem to some extent, but fail to maintain the scalability. MapReduce[1,2] based systems are famous for the excellent scalability and flexibility, which offer the potential chance for the data-intensive event processing.

However, it is not feasible and efficient to apply the available MapReduce framework directly to the CEP scenarios. MapReduce framework was designed for online analytical processing. Therefore, there are mainly four challenges to adapt to the framework. First of all, because original events are often collected continuously, we need to solve the problem of storing data stream. Second, due to the high I/O cost, the merge-sort phase in MapReduce tasks is not suitable for many real-time CEP scenarios. And then, we always want to get results early

Q.Z. Sheng et al. (Eds.): APWeb 2012, LNCS 7235, pp. 529–536, 2012.

and gradually instead of simultaneously in massive source data. At last, how to cache partial matching results in the stream processing is also a problem.

In order to solve the potential challenges, this paper proposes a novel framework on top of MapReduce to support efficient complex event processing while retaining high scalability and fault-tolerance. By modifying the Hadoop[3,4], we implement a framework, which mainly consists of the event stream storage module and the event stream processing module. Specially, we design an uploading architecture to push the data stream to distributed file system *(DFS)*. Furthermore, our framework design a hybrid sort function instead of merge-sort phase. And then, we have improved the basic MapReduce framework by introducing an input cache for reducer to support continuous queries over streams, and by utilizing the feedback mechanism for users to get some results more quickly.

The remainder of this paper is organized as follows: The related work is described in Section 2. In Section 3, we introduce the event stream storage module. Section 4 describes how to execute a query in our framework and specifies the improvements on MapReduce. The experimental studies are presented in Section 5. Finally, the work is concluded in Section 6.

2 Related Work

The centralized CEP systems have been intensively studied in recently years. SASE[5] first proposes the CEP problem and optimizes the automata model for the efficient correlation. Furthermore, some systems[6,7] are designed for specified application scenarios of CEP. Also, some researchers have proposed some distributed CEP systems. For example, [8] introduces a distributed CEP system and the query rewriting and distribution schemes are proposed.

MapReduce framework is proposed to support parallel analysis and computing. In recent years, modifying MapReduce framework for different scenarios has become a hot topic. Some researchers have proposed a new type of system named Hadoop++[9], which proposes new index and join techniques, namely Trojan Index and Trojan Join to improve runtimes of MapReduce jobs. MRShare[10] transforms a batch of queries into a new one that will be executed more efficiently, by merging jobs into groups and evaluating each group as a single query. It provides a solution that derives the optimal grouping of queries. HaLoop[11] not only extends MapReduce with programming support, but also improves efficiency by the task scheduler loop-aware and various caching mechanisms. However, all these frameworks can not support CEP applications.

3 Event Stream Storage Module

3.1 Event Model

Event object could be expressed as a triple factor group (ID, Timestamp, EventType). Among them, the *ID* of an event is a unique identification. We can use

it to distinguish persons or goods from others. *ID* may be the article number in the supermarket application. *Timestamp* is the time when a simple event is collected(e.g. detected by a RFID reader). *EventType* means the event's category(e.g. it can be classified according to the *ID* or location of the RFID reader).

3.2 Centralized Upload

At first, we design a master-slave mode in the event stream storage module, including a master node and some data collectors as slave nodes shown in figure 1. Data collectors are responsible for collecting event information in real-time and sending them to the master node immediately. There is no need to have very strong processing ability and memory capacity in data collectors.

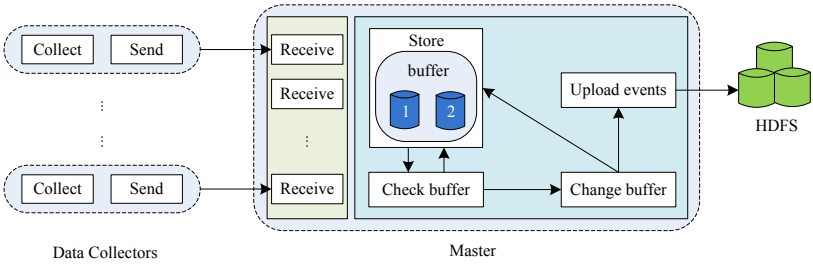


Fig. 1. Centralized upload architecture

We have adopted double buffer technology on the master node, which uses buffer *one* and buffer *two* alternately to receive events and preserve them. It is assumed that our framework is receiving events utilizing buffer *one* at T_0. At the moment of $T_1(T_1 > T_0)$, when buffer *one* is full of events or the number of them has exceeded the user-defined threshold, our framework will start to utilize buffer *two* instead of buffer *one*. Meanwhile, it will create a thread to upload all the events in buffer *one* to DFS. At $T_2(T_2 > T_1)$, events in buffer *two* have met the same uploading situation, and the processing method is similar to T_1. Taking turns using two buffers could guarantee that there is no need to add a synchronous lock to the buffer while uploading events in a batch.

3.3 Distributed Upload

If we only use a node as the master node to receive and upload events, it will become a bottleneck of our framework. The processing ability and memory capacity of a single node is very limited. To avoid that problem, we design and implement three-layer structure: data collection layer, event storage layer and path coordination layer. Our structure allows uploading events asynchronously in a batch. Path coordination layer usually stores the directory for uploading. Every node in the event storage layer is the same with the master node mentioned in the section 3.2. When any buffer is full, it will get a directory from the

path coordination layer and upload events to DFS immediately. We can increase some nodes to improve the scalability of our framework. Because of that, our framework could execute a query processing based on massive source data.

4 Event Stream Processing Module

Our framework will return the results with very low latency. The architecture of our event stream processing module is shown as figure 2. It is divided into four layers: application layer, parsing layer, computation layer and storage layer. In the application layer, we support different applications, such as SEQ() and COUNT(). As soon as receiving some queries, our framework will classify and parse them to the corresponding deterministic finite automatons*(DFAs)* by utilizing the query parsing engine in the parsing layer. And then, it will transfer a DFA to the MapReduce task and execute it immediately. Finally, MapReduce task will read events from the storage layer and results will be sent to users in real-time. We use DFS, RDBMS and local file systems as our storage layer.

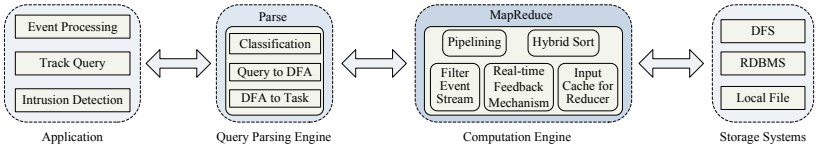


Fig. 2. Event stream processing module architecture

4.1 MapReduce Task

Different queries may have similar executions with each other in our framework. Map task is responsible for filtering some events which have no relationship with the query. Reduce task usually utilizes DFA to identify whether the result has been found or not. As an example, SEQ(A, B, C) is shown as figure 3.

Every file in DFS has many events. First, input files should be changed into event objects through the InputFormat phase. Map task is to filter events to reduce the cost of the sort and group phase. The output of map task is like *(ID + Timestamp, Event)*. After map tasks, the partition phase will begin. It uses a hash function only on the attribute *ID* to ensure that the events with the same *ID* will be processed in one reduce task. Before the reduce phase, our framework should sort the input data according to the key which has two attributes *ID* and *Timestamp*. And the group function will be only used on the attribute *ID* after sorting. Reduce task is to match and identify events gradually for each group. When the results are matched, our framework will return them to users immediately. There is no output in each reduce task, so our framework skips the OutputFormat phase. We also design the hybrid sort, feedback mechanism and input cache for the reducer. We will describe them later.

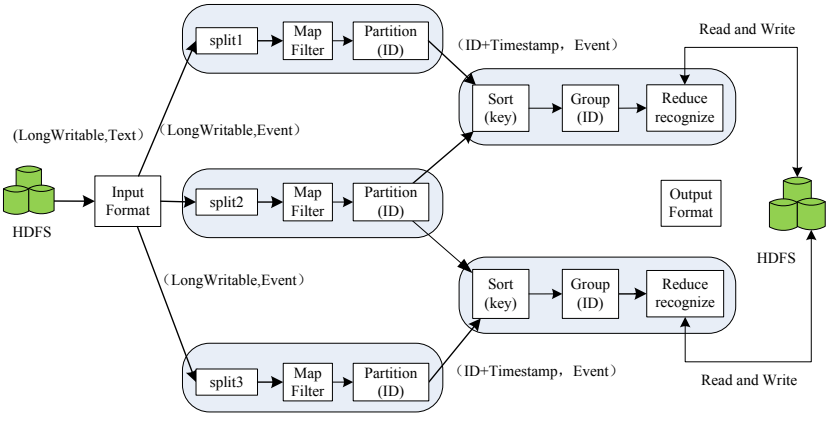


Fig. 3. MapReduce task of SEQ(A, B, C)

4.2 Hybrid Sort

We have to sort all the events by attribute *Timestamp* and group them by attribute *ID* in some query processings. After that, our system could begin to match and identify events. The basic MapReduce uses merge-sort in the sort phase. Although it can meet our requirements, merge-sort may cause very high I/O costs. It wastes too much time and the latency has to be delayed. To solve this problem, we have changed the MapReduce framework by replacing merge-sort with the hybrid sort. Hybrid sort is shown as figure 4, which is executed on the reduce node. And it blends a hash function and a merge-sort phase.

The output of map task is in the format of *(key, value)*. Key contains *ID* and *Timestamp* information. We will utilize the hash function on *ID* so that events with different *ID* could be set in different buckets. If the number of events in a bucket is beyond the memory capacity, we will save all the information of the bucket to local disks. For every bucket, our system uses merge-sort to sort events. After that, the events in a bucket have already been in the order of *Timestamp* and they will be processed as the same group by a reduce task.

When the data exceeds the memory capacity, the I/O cost of merge-sort may be as (1). F is the merge factor and B means the memory size. They could be configured in the files. N means the size of data set as the reduce input.

$$G(n) = \left(\frac{B^2}{2F(F-1)}n^2 + \frac{3B}{2}n - \frac{F^2}{2(F-1)}\right) \cdot B \tag{1}$$

$$F(n) = \frac{2n}{B} + \sum_{i=0} \alpha_i \cdot G(n_i) \qquad \left(\sum_{i=0} n_i = n\right) \tag{2}$$

The I/O cost model of our hybrid sort method is shown as (2). $\frac{2n}{B}$ is the cost of grouping by the hash function. The total I/O cost of merge is the sum of each bucket's I/O cost, which is $\sum \alpha_i \cdot G(n_i)$. Among them, n_i means the size

of events in bucket i and α_i is a binary function. If $n_i > B$, α_i equals to 1. Otherwise α_i equals to 0. Because events of each bucket are often less than the memory size, α_i usually equals to 0, and thus hybrid sort has less I/O cost.

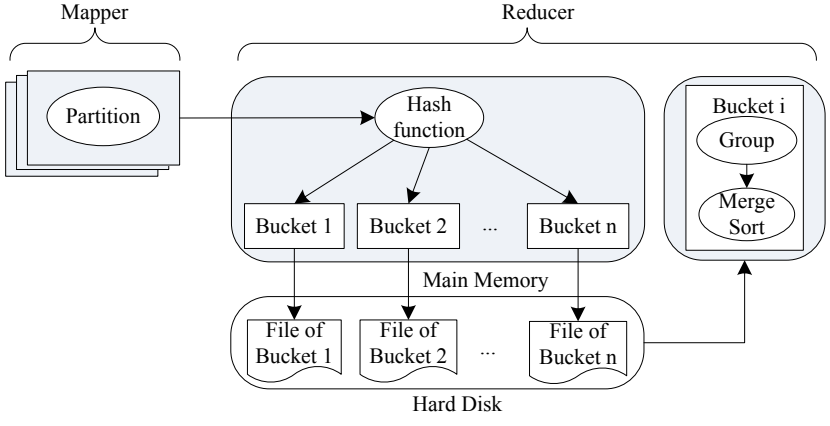


Fig. 4. Hybird sort

4.3 Feedback Mechanism

The basic MapReduce framework will save the output to DFS, so that our results have also to be stored into DFS. Users need to write another function to look up in all the results together. Our system utilizes the message mechanism and multi-thread technology in the reduce function to send results to users directly. Once matched, the results will be sent immediately. It makes users get the results gradually and users get them earlier by our feedback mechanism.

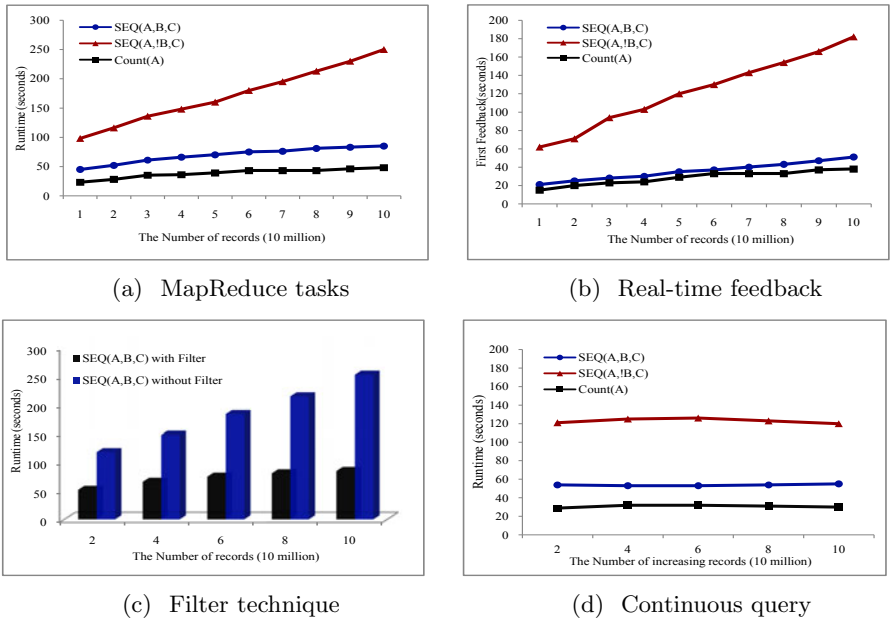
4.4 Input Cache for Reducer

We design and implement an input cache for reducer to support continuous query over event streams. When the next events have arrived, we only need to maintain the results incrementally, which are saved by the last batch processing. There is no need to scan all the events from the beginning to match the results. In order to keep the good fault-tolerance, we utilize the distributed file system or local disks as our cache mechanism. At the end of every MapReduce task, our framework will save the set of partial matched status of the DFAs to the input cache for reducer. When the next reduce task begins, it will read the input cache for reducer firstly, and then maintain the status incrementally.

5 Evaluation Performance

All the experiments are run on the cluster, which is composed of eleven nodes, including one master and ten slaves. Every node contains 2.00GHz Intel Xeon

CPU, 2GB RAM, 75GB SATA disks at 7,200rpm speed. All nodes run Hadoop-0.19.2 with the default parameters on Red Hat Enterprise Linux Server release 5.4 and are connected by gigabit Ethernet.



(a) MapReduce tasks (b) Real-time feedback

(c) Filter technique (d) Continuous query

Fig. 5. Evaluation performance with the simulated data set

Our experiments are performed on 2.7GB simulated data set including 10^8 records *(ID, Timestamp, EventType)*. We have simulated some applications about the theft in the supermarket, the monitor in the road network and the aggregation of clicks in the website, where SEQ(A,!B,C),SEQ(A,B,C) and Count(A) are separately used. We also have designed extensive experiments about runtime of MapReduce tasks on different records, filter technique, continuous query, and first feedback time. The experimental results are shown as figure 5.

According to the experiments, our system maintains the good scalability of Hadoop. As figure 5(a) shows, when the events are increasing in ten million level, the runtime of each query will not grow very rapidly. Even if we inquire on hundreds of millions events, Count() could return the results in one minute and SEQ() in five minutes. In figure 5(b), we improve our framework by the feedback mechanism. It will return all the results to users directly and asynchronously. Comparing with figure 5(a), users can get the first result of each level more than 50% earlier. Our framework firstly filter events which have nothing to do with queries. The selectivity of our data set is about 3.8% of Count(A) and 11.5% of SEQ(A,B,C) in figure 5(c). It shows that the runtime of the MapReduce tasks will reduce to 40% using filters, because we have fewer records to sort and match. It will have fewer I/O costs if memory usage is not beyond the buffer size. In

figure 5(d), our framework only matches and detects the newly coming twenty million events incrementally by utilizing the input cache for reducer. It is obvious that our performance has been improved a lot comparing to figure 5(a).

6 Conclusion

This paper has put forward a novel framework on top of MapReduce to support efficient complex event processing while remaining high scalability. It solves the problem of efficiently processing massive source data of the various complex event processing applications. We have designed an uploading architecture to support massive source data storage. The basic MapReduce framework has been improved by the hybrid sort, feedback mechanism and input cache for reducer. The experimental results verify our framework has good performance and scalability in CEP query processing. In the future, we will further design the effective share mechanism to optimize multi-users shared queries.

Acknowledgment. This research was supported by National Natural Science Foundation of China (No.61033007, No.61173028, No.61003058) and the Fundamental Research Funds for the Central Universities(N100704001).

References

1. Jeffrey, D., Sanjay, G.: Mapreduce: Simplied data processing on large clusters. In: OSDI (2004)
2. Jeffrey, D., Sanjay, G.: Mapreduce: a fiexible data processing tool. Communications of the ACM (2010)
3. Tom, W.: Hadoop: The Definitive Guide. O'Reilly, Yahoo! Press (2009)
4. hadoop (2011), `http://hadoop.apache.org/`
5. Eugene, W., Yanlei, D., Shariq, R.: High-Performance Complex Event Processing over Streams. In: SIGMOD (2006)
6. Kyumars, S.E., Tahmineh, S., Peter, M.F.: Changing Flights in Mid-air: A Model for Safely Modifying Continuous Queries. In: SIGMOD (2011)
7. Chun, C., Feng, L., Beng, C.O.: TI: An Efficient Indexing Mechanism for Real-Time Search on Tweets. In: SIGMOD (2011)
8. Nicholas, P., Matteo, M., Peter, P.: Distributed Complex Event Processing with Query Rewriting. In: DEBS 2009 (2009)
9. Jens, D., Jorge-Arnulfo, Q., Alekh, J.: Hadoop++: Making a Yellow Elephant Run Like a Cheetah. In: VLDB (2010)
10. Tomasz, N., Michalis, P., Chaitanya, M.: MRShare: Sharing Across Multiple Queries in MapReduce. In: VLDB (2010)
11. Yingyi, B., Bill, H., Magdalena, B.: HaLoop: Efficient Iterative Data Processing on Large Clusters. In: VLDB (2010)