ISSN(Online): 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Website: www.ijircce.com

Vol. 6, Issue 4, April 2018

State Management in Apache Apex using RocksDB incremental checkpointing

Akash Joshi [1], Ayushi Jain [1], Neeraj Goidani [1], Pranoti Bulakh [1]

U.G. Student, Department of Information Technology, MIT College of Engineering, Pune, Maharashtra, India[1]

ABSTRACT: Big data refers to data with the following characteristics- huge velocity, variety, veracity and volume. This data is dealt with in two ways – batch and stream processing. Apache Apex platform which runs on Hadoop Yarn, is used for big data processing which allows real time stream and batch processing. Fault tolerance is a very important feature needed while processing big data. Apache Apex uses the checkpointing mechanism to ensure fault tolerance. Checkpointing is a recovery process. RocksDB is the database proposed to support checkpointing. It is a library written in C++. It is an embedded key value data store. Keys and values are byte streams. It can store few terabytes of data on RAM or flash drives in local system. This paper is to propose a mechanism to reduce the latency and overcome other issues of checkpointing in Apache Apex.

KEYWORDS: Fault tolerance, Apache Apex, recovery, checkpointing, Hadoop Distributed Hash Table, latency, RocksDB

I. INTRODUCTION

Big data processing systems have a lot of concepts. Operators, for example, are important elements of a big data system that directly affect data. Operators are primary blocks to build an Apache Apex [1] application. Operators which are Java classes and are used to accept input data, process it and send the output to the other operators. The state of an operator maintains the history of the operations which occurred in the past and it plays a role in determining the logic for future computations.

In some scenarios an operator or an entire application fails due to reasons such as less resource availability, less memory capacity etc. Generally, the answer would be to restart the entire process from scratch. However this deals in loss of huge amounts of data already processed along with the results. It hampers resources like time and CPU capacity. This approach cannot even be considered in terms of dealing with big data as the losses would be substantial. Hence, the state must be kept in a reliable location to enable proper recovery. This is termed as fault tolerance. Fault tolerance is a definite requirement. The Apache Apex [1] system is made fault tolerant by using checkpointing. In checkpointing, the snapshots of a process are stored periodically and retrieved in case of error or failure. Apex treats checkpointing as an inbuilt functionality.

Apache Apex [1] stores the state of the operators in the local file system continuously. These states are periodically pushed into the HDFS. Once the state of an operator is pushed into HDFS, it is said to be checkpointed. Hence if a process fails, Apache Apex system looks into HDFS, retrieves the most recent state and pushes this into the local file system from where the process restarts.

II. RELATED WORK

Hbase [2] and Cassandra [3] are based on Log Structured Merge Trees but converting them into an embedded library would take a lot of effort. Thus embedded databases such as BerkleyDB, KyotoDB, SQLite3, LevelDB [4] were better alternatives for implementing checkpointing. In this lot, LevelDB was the fastest. LevelDB however did not strike the perfect tradeoff between read, write and space amplification. Hence RocksDb was created. RocksDB inherits its structural framework from LevelDB. In order to make the Apex checkpointing efficient, a Hadoop based data store is developed called Hadoop Distributed Hash Table (HDHT). The programming model of HDHT [5] is based on key-value stores applicable to various use cases. This embedded key-value store or hash table is based on HDFS. It is

ISSN(Online): 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Website: www.ijircce.com

Vol. 6, Issue 4, April 2018

assumed that the events that are being processed carry the key which will enable storage and retrieval of data from the database.

Since HDHT is embedded and requires no installing/managing operations, it was chosen over the other key value stores which are compatible with streaming applications. Apex checkpointing when integrated with HDHT provides exactly once guarantee. Exactly once guarantee means tuples are guaranteed to be processed only once by all the operators in the application, even in the case of any failures [5]. HDHT was based on HDFS as HDFS could perform scalable reads and writes. The key and values are byte arrays, by default stored in an alphabetical order. In scenarios where the volume of incoming tuples is more than the operator can handle, resulting in high latencies and reduced throughput, the operators replicate themselves on different nodes so that the load is shared by all the replicas. These replicas can be called as the partitions of that operator and this process is called as partitioning. The HDHT file system shows this partitioning of the operator. Each partition uses a separate directory having its own WAL (Write Ahead Log) and metadata file. Each of these partitions has a bucketKey which is the name of the subdirectory .

Supported file formats are Tfile - Hadoop file format, DTfile -written in Tfile format during HDHT lookup and reduces the disk I/Os and Hfile - written in Hbase format. HDHT achieves high throughput, low latency and fault tolerant by writing the updates in a write ahead log and only a few data files by key design. The timestamp can be used as prefix by the key. At the end of every application window, the WAL is pushed to the HDFS. Operator is blocked until the WAL is persisted to the HDFS successfully. Only after the flushing is performed the operator resumes its processing of the incoming tuples again. Limitations of HDHT are that since it is based on HDFS, continuously accessing HDHT was costly and it showed unpredictable speed variation in tuples processing at every checkpoint.

III. PROPOSED ALGORITHM

A. *Design Considerations:* Apex operators are represented in a Directly Acyclic Graph (DAG). These operators store their processed data in RocksDB on their local file system A checkpoint is taken after every thirty seconds (default).

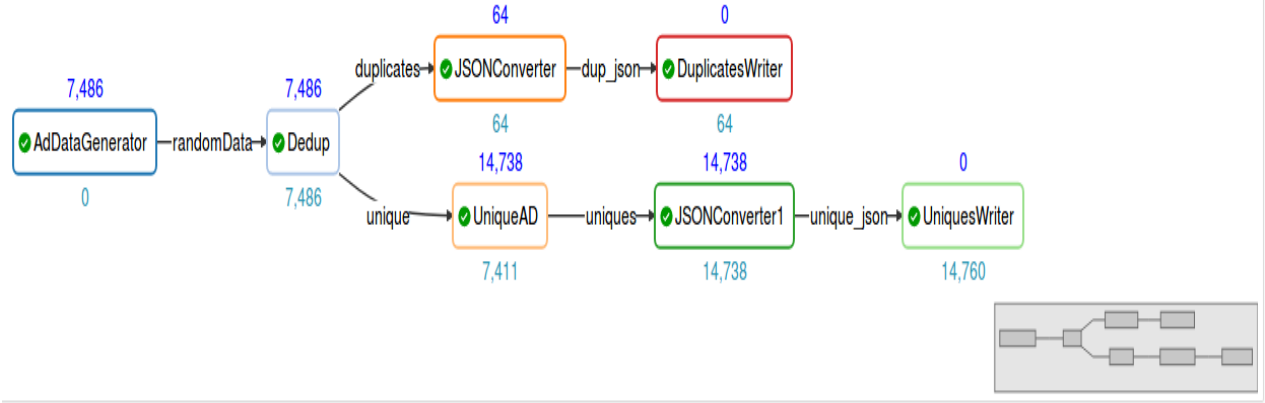

Fig.1.Operators represented in a DAG [Taken from Datatorrent Console]

B. *Description of the Proposed Algorithm:* To overcome the issues given by Hadoop Distributed Hash Table (HDHT), another key value store was proposed as an alternative. Since accessing the HDFS to serialize the state of the operators was a costly option, RocksDB [6] which is an embedded key-value store was chosen. In this system, the operators which are processing streamline data were storing processed tuples in RocksDB on the local file system. RocksDB has three storage structures- Memtable (in-memory structure, mutable), Sorted String Tables/SST (secondary storage structure, immutable) and Write Ahead Log/WAL (in-memory structure, mutable). Once the memtable reached its threshold, data got flushed to the secondary storage structures i.e. SSTs. These SSTs are based on the Log merge Structured Trees [6]. This facilitates a compaction background task where SSTs are checked for duplicates and merged into one single SST. This makes RocksDB better in storage efficiency and access speed. In the previous system, the whole state of the operator was being pushed on to the HDFS at every checkpoint. This is

ISSN(Online): 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Website: www.ijircce.com

Vol. 6, Issue 4, April 2018

not necessary since the state changes from one checkpoint to the other were not that large. This gave way to the idea of incremental checkpointing where only the change in the state of the operator is checkpointed instead of the whole state. In the proposed system, an API created in Java called RockDBStore is used which consists of methods that perform incremental push and full recovery of state of Apex operator. This API interfaces RocksDB instance with HDFS. The API maintains a hash map consisting of filename as the key and its timestamp as the corresponding value. A new file created by RocksDB is entered into the hash map with its timestamp and a modified file gets updated in the hash map with its new timestamp. Thus, a list of files which are newly generated along with the files that were modified is obtained which are eventually pushed onto the HDFS at checkpoint. After the event of a node failure, the Apex operator gets deployed on another node. It initializes a RocksDB instance on its local file system. The state of the operator before failing is retrieved from the HDFS and loaded into the RocskDB using the loadFromHDFS() method of the RockDBStore. The operator restarts its working from the state that was last checkpointed.

IV. PSEUDO CODE

Incremental checkpointing
Step 1: Initialize a RocksDB instance on local file system of each operator node.
Step 2: Import RockDBStore API
Step 3: Create Hash Map to store file name and timestamp of files created by RocksDB.
Step 4: if (timestamp==null)
File is new hence add it to the list of files to be pushed into HDFS.
File is added as new entry in hash map.
else if (stored_timestamp<current_timestamp)
File is modified hence added to list of files to be pushed into HDFS.
Timestamp is modified of the file in hash map.
Step 5: At checkpoint, push the list of files onto HDFS.
Step 6: Repeat steps 4, 5, 6 until operator fails or application is killed.
Step 6: End.

Retrieval of state of operator after failure of node
Step 1: Operator/Application killed and deployed on another node.
Step 2: Initialize RocksDB instance on local file system of the node.
Step 3: Retrieve all files from HDFS into RocksDB.
Step 4: Resume processing of operator.
Step 5: End

V. SIMULATION RESULTS

In order to analyse and compare the performance of the existing HDHT system with the proposed RocksDB system, a benchmarking application was carried out. First, the apex operator system used the HDHT store to serialize its state until the state was pushed into the HDFS. The apex system then used the RocksDB store to keep its operator history and then pushed only the changed state into HDFS implementing incremental checkpointing. The resulting behaviour of these two systems is represented using the number of tuples processed by each store per second in the graphs below. Figure 2 and 3 show time in minutes (mm:ss format) on the X-axis and number of tuples being processed per second on the Y-axis. The graphs help us in the comparative study in the behaviour of the two stores.

The optimal processing speed of HDHT was 49,342 tuples/second. This speed dropped to around 30,000-35,000 tuples/second at every checkpoint (after every 30 seconds). HDHT took around 25-35 seconds to come up to its optimal tuple processing speed. Thus, we can say that HDHT showed some latency during the time when the state was being pushed from HDHT to the HDFS. The nature of the graph in figure 2 represents the unpredictable variation in the processing speed of HDHT.

ISSN(Online): 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Website: www.ijircce.com

Vol. 6, Issue 4, April 2018

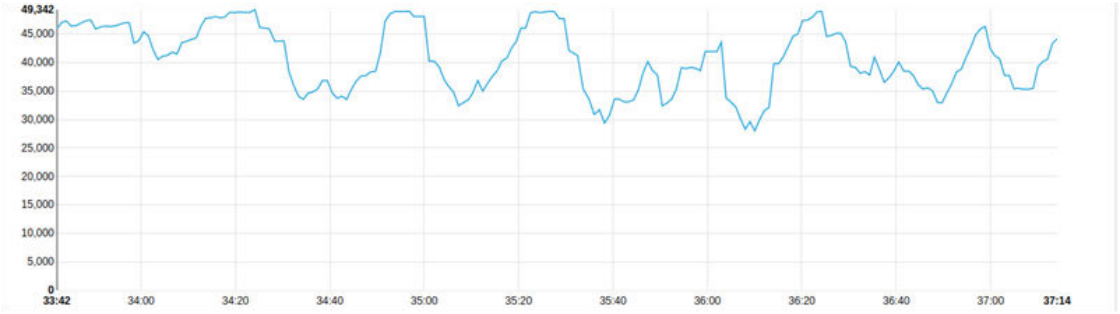

Fig.2. HDHT performance (Time in minutes (mm:ss format) on the X-axis and number of tuples being processed per second on the Y-axis)

On the other hand, incremental checkpointing using RocksDB had its optimal tuple processing speed as 58,376 tuples/second. During checkpointing it dropped down to around 48,000 tuples/second. This performance shown by RocksDB is better than HDHT in terms of tuple processing. RocksDB then took 12-15 seconds to come up to its optimal speed which was almost half of the time taken by HDHT.

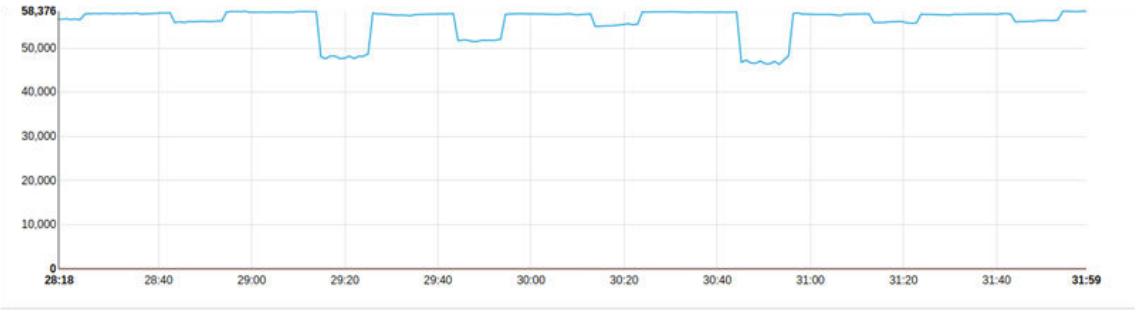

Fig.3. RocksDB performance (Time in minutes (mm:ss format) on the X-axis and number of tuples being processed per second on the Y-axis)

The simulation results thus prove that RocksDB is an efficient key value database to store the state of the Apex operators in the local file system. It resumes to its maximum speed of processing in much lesser time and is a better choice over the HDFS based store in terms of latency.

VI. CONCLUSION AND FUTURE WORK

The proposed system of incremental checkpointing using RocksDB showed substantial improvement in the system as compared to the system that uses complete state checkpointing. The results show that the process of storing only the delta of state of operators at every checkpoint made the mechanism more efficient in terms of processing speed of tuples and made the behavior of operators stable in nature. The scope of this proposed system is to develop an intelligent recovery system in place of the current full recovery system. In this intelligent recovery file system, a META file can be created which keeps a track of which files have been added at every checkpoint. This can help in retrieving only the SST files that were active at the time of operator failure instead of retrieving all the SST files.

ISSN(Online): 2320-9801
ISSN (Print) : 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(A High Impact Factor, Monthly, Peer Reviewed Journal)

Website: www.ijircce.com

Vol. 6, Issue 4, April 2018

REFERENCES

1. Apache Apex. https://apex.apache.org, 2017.
2. D. Carstoiu, A. Cernian, and A. Olteanu, "Hadoop hbase-0.20.2 performance evaluation", New Trends in Information Science and Service Science (NISS), 4th International Conference, pp. 84 –87, 2010.
3. E. Dede, B. Sendir, P. Kuzlu, J. Hartog, and M. Govindaraju, "An evaluation of cassandra for Hadoop", Proceedings of 2013 IEEE Sixth International Conference on Cloud Computing, pp. 494–501, 2013.
4. RocksDB. http://rocksdb.blogspot.in/
5. HDHT. https://www.datatorrent.com/blog/data-store-for-scalable-stream-processing/
6. RocksDB. https://github.com/facebook/rocksdb/wiki
7. Paris Carbone, Stephan Ewen, Gyula For a, Seif Haridi, Stefan Richter, Kostas Tzoumas ,"State Management in Apache Flink Consistent Stateful Distributed Stream Processing", Proceedings of the VLDB Endow., Vol. 10, No. 12,pp. 1718-1729,August 2017.
8. Q. To, J. Soto, V. Markl, "A survey of state management in big data processing systems," CoRR, vol. abs/1702.01596, 2017.
9. M. Manwal and A. Gupta, "Big data and hadoop — A technological survey," International Conference on Emerging Trends in Computing and Communication Technologies (ICETCCT), Dehradun, pp. 1-6, 2017.
10. Z. Zhang, Y. Yue ,B. He ,J. Xiong ,M. Chen ,L. Zhang and N. Sun, "Pipelined Compaction for the LSM-Tree," IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, pp. 777-786,2014.
11. S. J. Shepard, "Embedded databases can power emerging world of information to go," in IT Professional, vol. 2, no. 6, pp. 10-13, Nov/Dec 2000.
12. K. Singh and R. Kaur, "Hadoop: Addressing challenges of Big Data", IEEE International Advance Computing Conference (IACC), Gurgaon, pp. 686-689, 2014.