Distributed in-memory JVM cache

Nikhil Kurhe, Suyash Chaudhari, Onkar Shedge

***Abstract*—** Big Data analysis and processing is a very interesting field with plenty of ongoing research. One of the major constraints in this field is that a single machine's processing power is not growing in proportion with the size of the ever growing dataset sizes. In order to handle this problem, one workaround that has been in place for a while now are distributed systems such as Hadoop for analysis. However most of the existing solutions are largely disk based, with low latency solutions being even fewer. With a distributed in-memory JVM cache, the aim to achieve multi column query results quickly is fulfilled. A performance study of data analysis in java heap vs. java off heap on row store and column store was conducted, which lead to the conclusion that column store off heap java cache will deliver the best performance. Existing hadoop design is referred for communication between nodes and handling failure cases; and some improvements are done in the existing design.

***Keywords*—** *Allocation/de-allocation strategies, Distributed memories, Garbage collection, Main memory, Operating Systems, Software, Storage Management.*

*****

I. INTRODUCTION

Consider a database table which we want to load into java memory. This table is huge consisting of a large number of rows and 50 columns. Since this whole table won't fit into a single Java Virtual Machine's memory we would like to divide it into small partitions and load them in multiple Java Virtual Machines situated on different machines. Since we are now relying on multiple machines we need to introduce replication of these partitions so that failure cases are handled. We can follow the HDFS concepts of name node and data nodes.

II. EXISTING DESIGN DETAILS

A. *Abbreviations and Acronyms*

HDFS - Hadoop Distributed File System
JVM - Java Virtual Machine
DSL - Distributed Storage Layer
REST – REpresentational State Transfer

B. *Ehcache, Redis and Memcached*

Ehcache is an open source, standards-based cache that boosts performance, offloads the database, and simplifies scalability. Ehcache scales from in-process caching, all the way to mixed in-process/out-of-process deployments with terabyte-sized caches. It has an in-memory data structure and key-value store. It is a multi-user system.

Memcache is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering. It's intended for use in speeding up dynamic web applications by alleviating database load.

Redis is an open source (BSD licensed), in-memory data structure store, used as database, cache and message broker. It has built-in replication, Lua scripting, transactions and different levels of on-disk persistence, and provides high availability.

The following figures compare various cache[1].

This work was supported by eQ Technologic.

Nikhil Kurhe is currently a final year computer engineering student at Pune Institute of Computer Technology, Pune under University of Pune (e-mail: Nikhil.kurhe@gmail.com).

Suyash Chaudhari is currently a final year computer engineering student at Pune Institute of Computer Technology, Pune under University of Pune (e-mail: suyashc1295@gmail.com).

Onkar Shedge is currently a final year computer engineering student at Pune Institute of Computer Technology, Pune under University of Pune (e-mail: shedge31onkar@gmail.com).

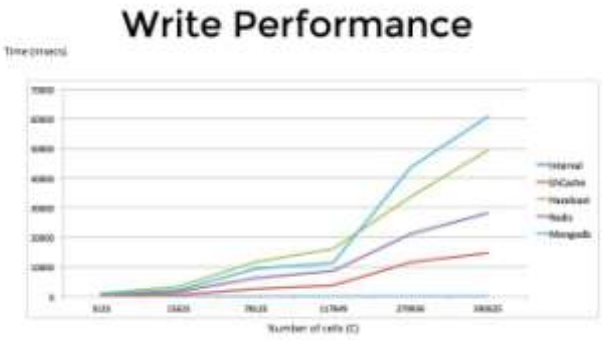

Figure II-1: Comparision of write performance [1]

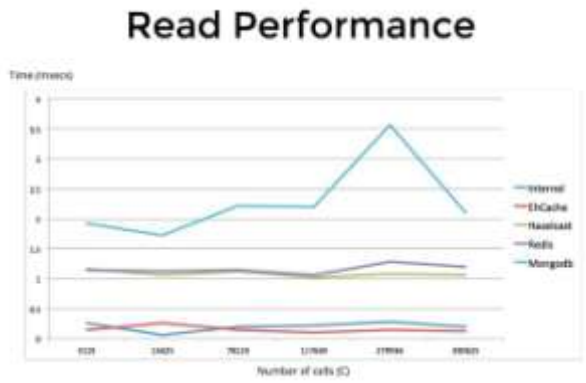

Figure II-2: Comparision of read performance [1]

Local caches (internal HashMap and EhCache) have the best write performance as expected. Redis performed better than Hazelcast and MongoDB for writes. Local caches have the best read performance, again as expected.

C. HDFS architecture

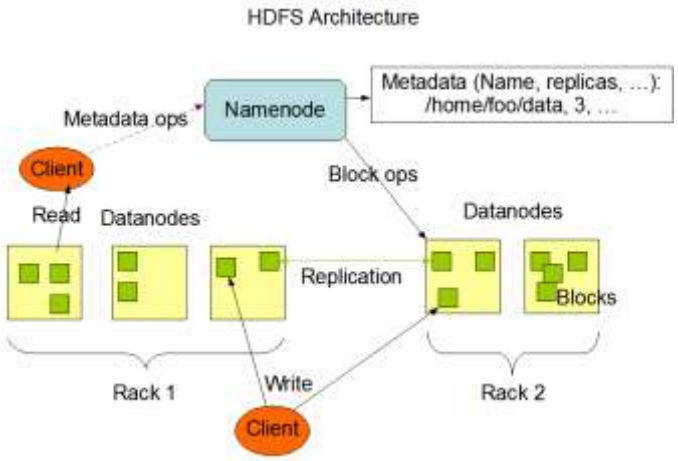

Figure II-3: HDFS architecture [2]

HDFS has master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes.

The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

D. Comparison of row store and column store

The conclusion that off heap column store java cache will deliver the best performance was based on the following test[2]:

Test details:
Machine configuration: i5 – 4 cores, 3.20 GHz, 6 Gb RAM

Worker/JVM configuration:

Min JVM memory	2 GB
Max JVM memory	2 GB
JVM mode	Server
JDK	1.6.0.27- 64 bit

DataCube Information:

No. of dimensions	1	
No. of measures	3	[numeric, string, string]

Level and number of rows	Number of mappers/reducers	Number of threads to be spawned
Level 0 (1 row)	1 reducer	1 thread
Level 1 (50 rows)	1 mapper, 1 reducer	1 thread for each pair
Level 2 (1500 rows)	2 mappers, 2 reducers	2 threads
Level 3 (15 lac rows)	4 mappers	4 threads

Total 8 threads are required for executing a rollup of a user.
Observations:

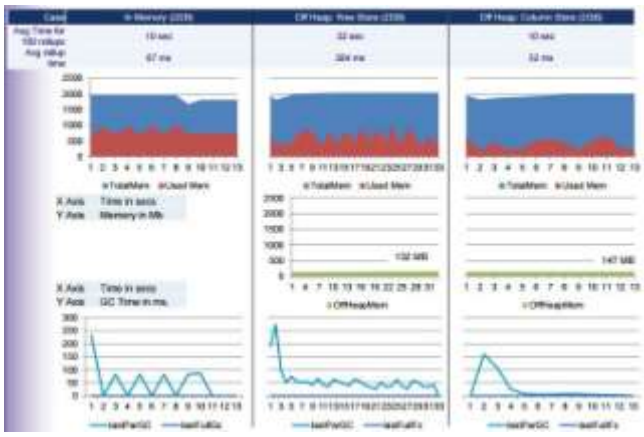

Figure II-4: 1 user rollup

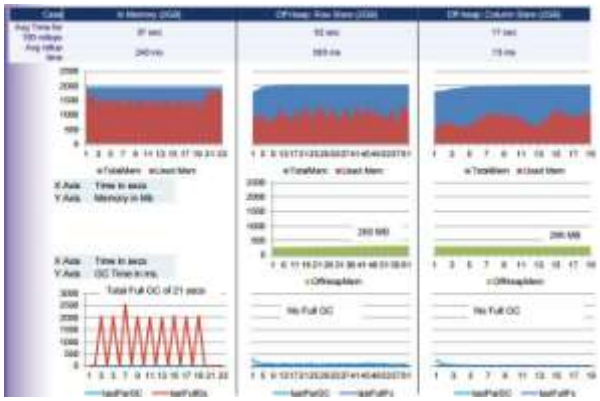

Figure II-5: 2 user rollup

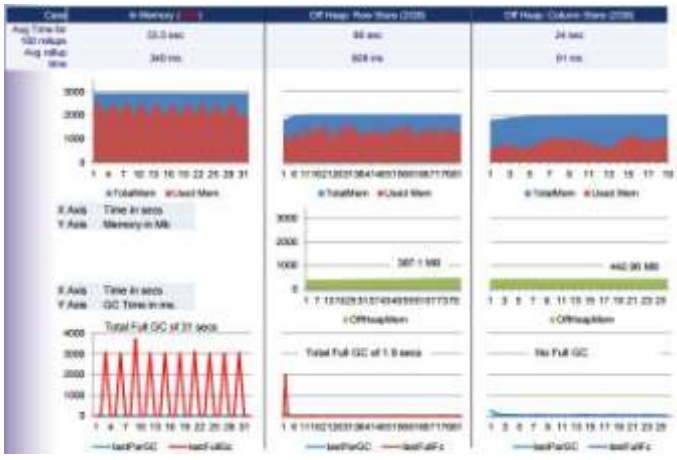

Figure II-6: 3 user rollup

Figure II-7: 4 user rollup

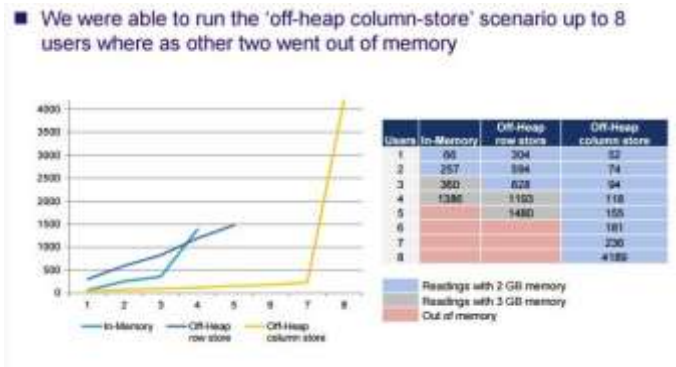

Figure II-8: Further observations

III. DESIGN

Communication between nodes:

The communication between nodes is achieved using netty. [3]Netty is an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients. It greatly simplifies and streamlines network programming such as TCP and UDP socket server. It has a unified API for various transport types - blocking and non-blocking socket. The communication is passing of events over the wire using Kryonet. KryoNet is a Java library that provides a clean and simple API for efficient TCP and UDP client/server network communication using NIO. KryoNet uses the [4]Kryo serialization library to automatically and efficiently transfer object graphs across the network. Each event has a handler which handles the particular event. The communicator has a queue of events which are to be processed. These events are executed in a global eventexecutor which is a fixed/scheduled thread pool. This thread pool contains a limited number of threads. Events for example on NameNode can be DATA_NODE_REGISTRATION, HEARTBEAT, LOCATE_BLOCK, ADD_BLOCK, CHECK_REPLICATION. Asynchronous event when submitted a Future object[5] is returned and hence the current thread is not blocked. A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.

Incomplete replication is handled by NameNode. Each node has eventhandlers which are mapped to events.

Storage:

The blocks of data are stored on off heap. The on-heap store refers to objects that will be present in the Java heap (and also subject to GC). On the other hand, the off-heap store refers to (serialized) objects, but stored outside the heap (and also not subject to GC). As the off-heap store continues to be managed in memory, it is slightly slower than the on-heap store, but still faster than the disk store. Off heap memory is represented as buckets which contain pages. There could be memory holes (Internal defragmentation). Internal fragmentation occurs when extra space is left empty inside of a block of memory that has been allocated for a client. This usually happens because the processor's design stipulates that memory must be cut into blocks of certain sizes -- for example, blocks may be required to be evenly be divided by four, eight or 16 bytes. When this occurs, a client that needs 57 bytes of memory, for example, may be allocated a block that contains 60 bytes, or even 64. The extra bytes that the client doesn't need go to waste, and over time these tiny chunks of unused memory can build up and create large quantities of memory that can't be put to use by the allocator. Because all of these useless bytes are inside larger memory blocks, the fragmentation is considered internal.

Data Storage on DataNodes:

On the datanodes, the data is stored in the offheap, as mentioned above. However the organisation of the data both in the offheap and on the wire is somewhat different. Using two main interfaces, Data and DataBlock and their corresponding abstract classes, the data is stored and transported in the form of DataBlocks. Each DataBlock has DataBlockMetaInfo and handlers. The blocks are generated using a DataBlockFactory. Furthermore, each datanode maintains a localcache wherein, the most recent and frequently accessed blocks are already retrieved from offheap to the main memory so as to avoid wasting time during computations on that data.

MetaData Storage on NameNode:

The storage of metadata is done on namenode using an improvisation of Hadoop Gset structure known as DSet along with BlocksMap, a structure that is used to deal with failure cases. The Dset is a combination of linked list and array that can be traversed in multiple ways. Using a hash function each block is assigned to a particular bucket. Each block also stores the next and previous block address along with the bucket it belongs to, known collectively as blockInfo. Now in case a node goes down, i.e no heartbeat has been received, then using the Dset, it is easy to understand which blocks were stored on that node and now need to be replicated elsewhere on the cluster.

IV. EXPERIMENTAL SETUP

Multiple JVM's in a cluster with one JVM acting as NameNode and others as DataNodes.
Min JVM memory: 2 GB
Max JVM memory: 2 GB
JVM mode: Server
JDK: 1.6.0.27-64 bit

V. CONCLUSION

A comparative study of row store and column store multi-user rollup on paper lead to conclusion that column store leads to better results. A design is proposed based on existing HDFS design. The design is successfully implemented using the above mentioned structures. Various test cases have been considered during the implementation and by running the experimental test cases it has been proven that column store implementation has lead to better results.

References

[1] Nomis Labs, "Evaluation of Caching frameworks"
Available:http://blog.nomissolutions.com/labs/2015/03/10/evaluation-of-caching-frameworks/
[2] Apache, "HDFS architecture"
Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[3] The Netty project, "Netty"
Available: http://netty.io/
[4] EsotericSoftware, "Kryonet"
Available: https://github.com/EsotericSoftware/kryonet
[5] Oracle, "Future"
Available:
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html