HBase-0.20.0 Performance Evaluation

Anty Rao and Schubert Zhang, August 21, 2009

{ant.rao, schubert.zhang}@gmail.com，http://cloudepr.blogspot.com/

Background

We have been using HBase for around a year in our development and projects, from 0.17.x to 0.19.x. We and all in the community know the critical performance and reliability issues of these releases.

Now, the great news is that HBase-0.20.0 will be released soon. Jonathan Gray from Streamy, Ryan Rawson from StumbleUpon, Michael Stack from Powerset/Microsoft, Jean-Daniel Cryans from OpenPlaces, and other contributors had done a great job to redesign and rewrite many codes to promote HBase. The two presentations [1] [2] provide more details of this release.

The primary themes of HBase-0.20.0:

- Performance
  - Real-time and Unjavafy software implementations.
  - HFile, based on BigTable's SSTable. New file format limits index size.
  - New API
  - New Scanners
  - New Block Cache
  - Compression (LZO, GZ)
  - Almost a RegionServer rewrite
- ZooKeeper integration, multiple masters (partly, 0.21 will rewrite Master with better ZK integration)

Then, we will get a bran-new, high performance (Random Access, Scan, Insert, …), and stronger HBase. HBase-0.20.0 shall be a great milestone, and we should say thanks to all developers.

Following items are very important for us:

- Insert performance: We have very big datasets, which are generated fast.
- Scan performance: For data analysis in MapReduce framework.
- Random Access performance: Provide real-time services.
- Less memory and I/O overheads: We are neither Google nor Yahoo!, we cannot operate big cluster with hundreds or thousands of machines, but we really have big data.
- The HFile: Same as SSTable. It should be a common and effective storage element.

Testbed Setup

Cluster:

- 4 slaves + 1 master
- Machine: 4 CPU cores (2.0G), 2x500GB 7200RPM SATA disks, 8GB RAM.
- Linux: RedHat 5.1 (2.6.18-53.el5), ext3, no RAID

- 1Gbps network, all nodes under the same switch.
- Hadoop-0.20.0 (1GB heap), HBase-0.20.0 RC2 (4GB heap), Zookeeper-3.2.0

By now, HBase-0.20.0 RC2 is available for evaluation: http://people.apache.org/~stack/hbase-0.20.0-candidate-2/. Refer to [2] and [4] for installation.

Hadoop-0.20.0 configuration important parameters:

core-site.xml:

parameter	*value*	*notes*
io.file.buffer.size	65536	Irrelevant to this evaluation. We like use this size to improve file I/O. [5]
io.seqfile.compress.blocksize	327680	Irrelevant to this evaluation. We like use this size to compress data block.

hdfs-site.xml

parameter	*value*	*notes*
dfs.namenode.handler.count	20	[5]
dfs.datanode.handler.count	20	[5] [6]
dfs.datanode.max.xcievers	3072	Under heavy read load, you may see lots of DFSClient complains about no live nodes hold a particular block. [6]
dfs.replication	2	Our cluster is very small. 2 replicas are enough.

mapred-site.xml

parameter	*value*	*notes*
io.sort.factor	25	[5]
io.sort.mb	250	10 * io.sort.factor [5]
mapred.tasktracker.map.tasks.maximum	3	Since our cluster is very small, we set 3 to avoid client side bottleneck.
mapred.child.java.opts	-Xmx512m -XX:+UseConcMarkSweepGC	JVM GC option
mapred.reduce.parallel.copies	20	
mapred.job.tracker.handler.count	10	[5]

hadoop-env.sh

parameter	*value*	*notes*
HADOOP_CLASSPATH	${HADOOP_HOME}/../hbase-0.20.0/hbase-0.20.0.jar:${HADOOP_HOME}/../hbase-0.20.0/conf:${HADOOP_HOME}/../hbase-0.20.0/lib/zookeeper-r785019-hbase-1329.jar	Use HBase jar and configurations. Use Zookeeper jar.
HADOOP_OPTS	"-server -XX:+UseConcMarkSweepGC"	JVM GC option

HBase-0.20.0 configuration important parameters:

hbase-site.xml

parameter	*value*	*notes*
hbase.cluster.distributed	true	Fully-distributed with unmanaged ZooKeeper Quorum
hbase.regionserver.handler.count	20	

hbase-env.sh

parameter	*value*	*notes*
HBASE_CLASSPATH	${HBASE_HOME}/../hadoop-0.20.0/conf	Use hadoop configurations.
HBASE_HEAPSIZE	4000	Give HBase enough heap size
HBASE_OPTS	"-server -XX:+UseConcMarkSweepGC"	JVM GC option
HBASE_MANAGES_ZK	false	Refers to hbase.cluster.distributed

Performance Evaluation Programs

We modified the class org.apache.hadoop.hbase.PerformanceEvaluation, since the code has following problems: The patch is available at http://issues.apache.org/jira/browse/HBASE-1778.

- It is not updated according to hadoop-0.20.0.
- The approach to split maps is not strict. Need to provide correct InputSplit and InputFormat classes. Current code uses TextInputFormat and FileSplit, it is not reasonable.

The evaluation programs use MapReduce to do parallel operations against an HBase table.

- Total rows: 4,194,280.
- Row size: 1000 bytes for value, and 10 bytes for rowkey. Then we have ~4GB of data.
- Sequential row ranges: 40. (It's also used to define the total number of MapTasks in each evaluation.)
- Rows of each sequential range: 104,857

The principle is same as the evaluation programs described in Section 7, Performance Evaluation, of the Google BigTable paper [3], pages 8-10. Since we have only 4 nodes to work as clients, we set mapred.tasktracker.map.tasks.maximum=3 to avoid client side bottleneck.

Performance Evaluation

Report-1: Normal, autoFlush=false, writeToWAL=true

Experiment	Eventual Elapsed Time(s)	row/s	rows/s per node	ms/row per node	Total throughput (MB/s)	Google paper (rows/s in single node cluster)
random reads	948	4424	1106	0.90	4.47	1212
random writes(i)	390	10755	2689	0.37	10.86	8850
random writes	370	11366	2834	0.35	11.45	8850
sequential reads	193	21732	5433	0.18	21.95	4425
sequential writes(i)	359	11683	2921	0.34	11.80	8547
sequential writes	204	20560	5140	0.19	20.77	8547
scans	68	61681	15420	0.06	62.30	15385

Report-2: Normal, autoFlush=true, writeToWAL=true

Experiment	Eventual Elapsed Time(s)	row/s	rows/s per node	ms/row per node	Total throughput (MB/s)	Google paper (rows/s in single node cluster)
sequential writes(i)	461	9098	2275	0.44	9.19	8547
sequential writes	376	11155	2789	0.36	11.27	8547
random writes	600	6990	1748	0.57	7.06	8850

Random writes (i) and sequential write (i) are operations against a new table, i.e. initial writes. Random writes and sequential writes are operations against an existing table that is already distributed on all region servers, i.e. distributed writes. Since there is only one region server is accessed at the beginning of inserting, the performance is not so good until the regions are split across all region servers. The difference between initial writes and distributed writes for random writes is not so obvious, but that is distinct for sequential writes.

In Google's BigTable paper [3], the performance of random writes and sequential writes is very close. But in our result, sequential writes are faster than random writes, it seems the client side write-buffer (12MB, autoFlush=false) taking effect, since it can reduce the number of RPC packages. And it may imply that further RPC optimizations can gain better performance. If we set autoFlush true, they will be close too, and less than the number of random writes in report-1. The report-2 shows our inference is correct.

Random reads are far slower than all other operations. Each random read involves the transfer of a 64KB HFlie block from HDFS to a region server, out of which only a single 1000-byte value is used. So the real throughput is approximately 1106*64KB=70MB/s of data read from HDFS, it is not low. [3] [8]

The average time to random read a row is sub-ms here (0.90ms) on average per node, that seems about as good as we can get on our hardware. We're already showing 10X better performance than a disk seeking (10ms). It should be major contributed by the new BlockCache and new HFile implementations. Any other improvements will have to come from HDFS optimizations, RPC optimizations, and of course we can always get better performance by loading up with more RAM for the file-system cache. Try 16GB or more RAM, we might get greater performance. But remember, we're serving out of memory and not disk seeking. Adding more memory (and region server heap) should help the numbers across the board. The BigTable paper [3] shows 1212 random reads per second on a single node. That's sub-ms for random access, it's clearly not actually doing disk seeks for most gets. (Thanks for good comments from Jonathan Gray and Michael Stack.) [9]

The performance of sequential reads and scans is very good here, even better than the Google paper's [3]. We believe this performance will greatly support HBase for data analysis (MapReduce). In Google's BigTable paper [3], the performance of writes will be better than reads, includes sequential reads. But in our test result, the sequential reads are better than writes. Maybe there are rooms to improve the performance of writes in the future.

And remember, the dataset size in our tests is not big enough (only average 1GB per node), so the hit ratio of BlockCache is very high (>40%). If a region server serves large dataset (e.g. 1TB), the power of BlockCache would be downgraded.

Bloom filters can reduce the unnecessary search in HFiles, which can speed up reads, especially for large datasets when there are multiple files in an HBase region.

We can also consider RAID0 on multiple disks, and mount local file system with *noatime* and *nodiratime* options, for performance improvements.

Performance Evaluation (none WAL)

In some use cases, such as bulk loading a large dataset into an HBase table, the overhead of the Write-Ahead-Logs (commit-logs) are considerable, since the bulk inserting causes the logs get rotated often and produce many disk I/O. Here we consider to disable WAL in such use cases, but the risk is data loss when region server crash. Here I cite a post of Jean-Daniel Cryans on HBase mailing list [7].

> *"As you may know, HDFS still does not support appends. That means that the write ahead logs or WAL that HBase uses are only helpful if synced on disk. That means that you lose some data during a region server crash or a kill -9. In 0.19 the logs could be opened forever if they had under 100000 edits. Now in 0.20 we fixed that by capping the WAL to ~62MB and we also rotate the logs after 1 hour. This is all good because it means far less data loss until we are able to append to files in HDFS.*
>
> *Now to why this may slow down your import, the job I was talking about had huge rows so the logs got rotated much more often whereas in 0.19 only the number of rows triggered a log rotation. Not writing to the WAL has the advantage of using far less disk IO but, as you can guess, it means huge data loss in the case of a region server crash. But, in many cases, a RS crash still means that you must restart your job because log splitting can take more than 10 minutes so many tasks times out (I am currently working on that for 0.21 to make it really faster btw)."*

So, here we call put.setWriteToWAL(false) to disable WAL, and expect get better writing performance. This table is the evaluation result.

Report-3: NonWAL (autoFlush=false, writeToWAL=false)

Experiment	Eventual Elapsed Time(s)	row/s	rows/s per node	ms/row per node	Total throughput (MB/s)	Google paper (rows/s in single node cluster)
random reads	1001	4190	1048	0.95	4.23	1212
random writes(i)	260	16132	4033	0.25	16.29	8850
random writes	194	21620	5405	0.19	21.84	8850
sequential reads	187	22429	5607	0.18	22.65	4425
sequential writes(i)	241	17404	4351	0.23	17.58	8547
sequential writes	122	34379	8595	0.12	34.72	8547
scans	62	67650	16912	0.06	68.33	15385

We can see the performance of sequential writes and random writes are far better (~double) than the normal case (with WAL). So, we can consider using this method to solve some bulk loading problems which need high performance for inserting. But we suggest calling admin.flush() to flush the data in memstores to HDFS, immediately after each bulk loading job, to persist data and avoid loss as much as possible. The above evaluation does not include the time of flush.

Conclusions

Compares to the metrics in Google's BigTable paper [3], the write performance is still not so good, but this result is much better than any previous HBase release, especially for the random reads. We even got better result than the paper [3] on sequential reads and scans. This result gives us more confidence.

HBase-0.20.0 should be good to be used:

- as a data store to be inserted/loaded large datasets fast
- to store large datasets to be analyzed by MapReduce jobs
- to provide real-time query services

We have made a comparison of the performance to run MapReduce jobs which sequentially retrieve or scan data from HBase tables, and from HDFS files. The latter is trebly faster. And the gap is even bigger for sequential writes.

HBase-0.20.0 is not the final word on performance and features. We are looking forward to and researching following features, and need to read code detail.

- Other RPC-related improvements
- Other Java-related improvements
- New master implementation
- Bloom Filter
- Bulk-load [10]

In the world of big data, best performance usually comes from tow primary schemes:

(1) Reducing disk I/O and disk seek.
    If we come back to review the BigTable paper [3] Section 6, Refinements, we can find the goals of almost all those refinements are related to reducing disk I/O and disk seek, such as Locality Group, Compression, Caching (Scan Cache and Block Cache), Bloom filters, Group commit, etc.

(2) Sequential data access.
    When implementing our applications and organize/store our big data, we should always try our best to write and read data in sequential mode. Maybe sometimes we cannot find a generalized access scheme like that in the traditional database world, to access data in various views, but this may be the trait of big data world.

References:

[1] Ryan Rawson's Presentation on NOSQL. http://blog.oskarsson.nu/2009/06/nosql-debrief.html

[2] HBase goes Realtime, The HBase presentation at HadoopSummit2009 by Jonathan Gray and Jean-Daniel Cryans http://wiki.apache.org/hadoop-data/attachments/HBase(2f)HBasePresentations/attachments/HBase_Goes_Realtime.pdf

[3] Google paper, Bigtable: A Distributed Storage System for Structured Data http://labs.google.com/papers/bigtable.html

[4] HBase-0.20.0-RC2 Documentation, http://people.apache.org/~stack/hbase-0.20.0-candidate-2/docs/

[5] Cloudera, Hadoop Configuration Parameters. http://www.cloudera.com/blog/category/mapreduce/page/2/

[6] HBase Troubleshooting, http://wiki.apache.org/hadoop/Hbase/Troubleshooting

[7] Jean-Daniel Cryans's post on HBase mailing list, on Aug 12, 2009: Tip when migrating your data loading MR jobs from 0.19 to 0.20.

[8] ACM Queue, Adam Jacobs, 1010data Inc., The Pathologies of Big Data, http://queue.acm.org/detail.cfm?id=1563874

[9] Our another evaluation report: http://docloud.blogspot.com/2009/08/hbase-0200-performance-evaluation.html

[10] Yahoo! Research, Efficient Bulk Insertion into a Distributed Ordered Table， http://research.yahoo.com/files/bulkload.pdf