Fast MapReduce over Apache HBase

Technologies and Techniques Developed by Explorys Inc.

Keith Wyss
Software Engineer, Explorys Inc.

September, 24 2012

Agenda

1. Introduction
2. Design
3. Running MapReduce directly over HFiles

Why you should use HBase

- HBase is a columnar storage DataBase designed to facilate web scale write throughput by accessing on disk storage structures as a batch process and performing a scan and merge of columnar results from disparate "Store Files".
- HBase harnesses the fault tolerance of a Hadoop cluster and provides a tool to perform low latency reads upon that data by presenting an interface that unifies in memory caching, sorted key-values on disk and administration for managing the size of the files and garbage collecting data based on TTL and delete operations.

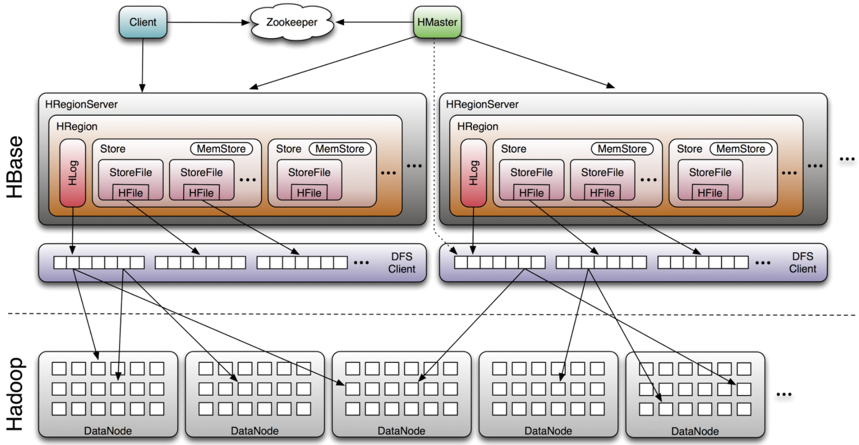
More reasons to use HBase

- HBase provides a database that may be used a source for data processing via MapReduce. HBase makes it easy to perform batch processing at scale upon data that processes writes in real time.
- HBase interoperates with the rest of the Apache distributed stack. It utilizes zookeeper efficiently to spread the load of database operations evenly across a cluster if you structure the data properly.
- HBase allows you to control your own destiny. Never have I seen interfaces that include so many byte arrays. HBase will not interfere with your custom serialization and does not require you to communicate with it via Json or web service calls. A formal schema is not required, but this does not mean structuring your data isn't important.

Why MapReduce Over HBase Detracts from Both

- Running MapReduce over an HBase table using the TableInputFormat is slow. It is many times slower than processing the same data sourced from raw HDFS files.
- Running MapReduce over an HBase table will destroy your read performance. The scanners that the MapReduce job employs flood the block cache with data that does not reflect your random read access patterns. Also, be prepared to deal with thrashing as your HBase table is probably quite large and none of it will be read twice, so none of this cache management is beneficial.

Architecture of an HBase Scan [4]



explorys

Running the Diagnostics

- At Explorys we realized that the problems with running MapReduce over HBase stem from a common root.
- The batch processing data access pattern is conflated with a typical client read scenario. TableInputFormat is not some glorious miracle of backend data structures. It is actually very similar to how you might write it if it didn't exist.
- Each map task uses the client API to scan over the Results in the HBase table. The primary advantage over typical use is merely concurrent scans allowing parallelism. The overhead of the client server architecture is offset by a large amount of buffering to reduce the impact of Remote Procedure calls, but it is far from the mark of a direct file-system access even if the data is local to the MapReduce task.

Direct Access

- We wanted to design an InputFormat that allowed direct, read-only subscription to the StoreFiles for an HBase Table.
- Through some conversation with the HBase committers, especially St. Ack, we were told
  1. We were crazy for thinking of this.
  2. More direct data access is something that's been requested before.
  3. We might have a viable approach by spinning up our own Region objects and using those to gain access to the underlying HFiles.
- There are a few contraindications to the approach that we will discuss later.

How to make this fast

- Each storeFile is a column-family specific sorted list of KeyValues. The storefiles themselves are ordered by the timestamps.
- The approach is to treat each HFile as a heap of KeyValues. We then built an outerheap of HFiles where the comparator is the KeyValue.COMPARATOR applied to the top of the heap.
- Then we merge the results into a Result object as they are read in to provide the familiar pair of rowkey and Result object.
- Due to reverse timestamp sorting, Delete KeyValues (tombstone markers) will be encountered before the older records that they nullify.

Anatomy of an HFile [1, 2]

A Typical HFile	
1	Data Block 0
2	Data Block 1
3	Meta Block
4	File Info
5	Data Block Index
6	Meta Block Index
7	Trailer

A Typical DataBlock	
1	Data Block Magic
2	Key Length
3	Value Length
4	Key
5	Value
6	Key Length
7	Value Length
8	Key
9	Value
10	...

explorys

Implementation

- Here at Explorys, we implemented this proposed InputFormat to be used as a plug and play replacement for TableInputFormat.
- Almost...
- The code is hosted on github at https://github.com/ExplorysMedical/Apothecary
- Problems? Compaction and Region Splitting. Memstore access. Why?

Dealing with HBase Data Administration

1. Turn off data source or funnel into temporary storage.
2. Trigger flush and compaction.
3. Wait for flush and compaction to finish. All the data is on disk.
4. Create a log of the existing storefiles.
5. Run the MapReduce Job
6. Check the log to see that the state of the table didn't change (region splits or compaction).
7. Possible do-over.

- That's a lot. What can we skip?
- Hard links?

Performance

	TableInputFormat	HFiles
Copying a Table	5:06	0:26
Explorys Indexing Job	1:50	0:30

- Better over large tables where cache doesn't have an impact.
- Worth the operation overhead in some circumstances.
- Major motivation for hard links and/or table snapshotting.

- Indexing Job run with 800 input splits 300 mappers.
- Copy Job run with 800 input splits, 95 concurrent Mappers.

Problems

- Serious memory usage. HFile block indexes can be quite large, although needn't be loaded in full since HBase 0.92 [3]
- Operational constraints unwieldy. HBase Administration is asynchronous, doesn't return monitoring object.
- Tenuous relationship with non-exposed API. No guarantees that upgrades won't break it.
- Aside from compaction rearranging files, Memstore is not included. We could possibly read WAL and construct our own memstore if compaction problem is solved..

References I

- Schubert Zhang (2009). HFile: A Block-Indexed File Format for Store Sorted Key-Value Pairs http://www.slideshare.net/schubertzhang/hfile-a-blockindexed-file-format-to-store-sorted-keyvalue-pairs
- Matteo Bertozzi (2011). HBase I/O: HFile http://th30z.blogspot.com/2011/02/hbase-io-hfile.html?spref=tw
- Apache Software Foundation (2012). HBase Book: Appendix E. HFile format version 2 http://hbase.apache.org/book/hfilev2.html
- Lars George (2009). HBase Architecture 101 -Storage http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html