

AN IN-DEPTH LOOK AT THE HBASE ARCHITECTURE



August 07, 2015 | BY Carol McDonald

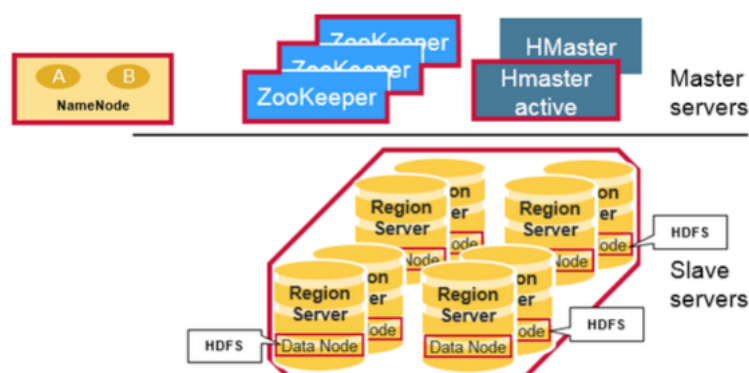
In this blog post, I'll give you an in-depth look at the HBase architecture and its main benefits over NoSQL data store solutions. Be sure and read the first blog post in this series, titled "HBase and MapR-DB: Designed for Distribution, Scale, and Speed." (/blog/hbase-and-mapr-db-designed-distribution-scale-and-speed#.VcKFnlVhBc)

HBASE ARCHITECTURAL COMPONENTS

Physically, HBase is composed of three types of servers in a master slave type of architecture. Region servers serve data for reads and writes. When accessing data, clients communicate with HBase RegionServers directly. Region assignment, DDL (create, delete tables) operations are handled by the HBase Master process. Zookeeper, which is part of HDFS, maintains a live cluster state.

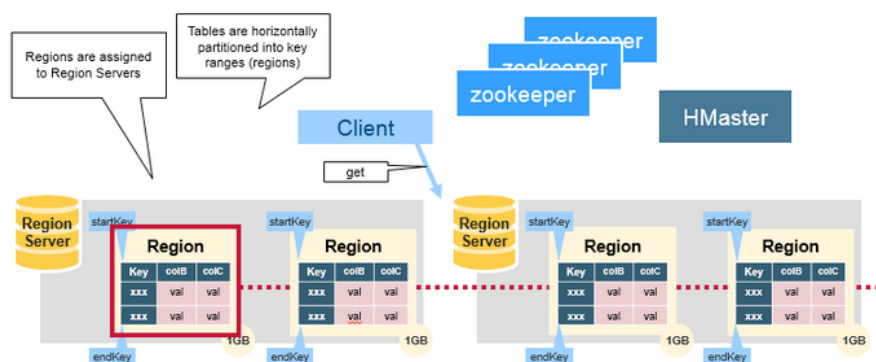
The Hadoop DataNode stores the data that the Region Server is managing. All HBase data is stored in HDFS files. Region Servers are collocated with the HDFS DataNodes, which enable data locality (putting the data close to where it is needed) for the data served by the RegionServers. HBase data is local when it is written, but when a region is moved, it is not local until compaction.

The NameNode maintains metadata information for all the physical data blocks that comprise the files.



REGIONS

HBase Tables are divided horizontally by row key range into “Regions.” A region contains all rows in the table between the region’s start key and end key. Regions are assigned to the nodes in the cluster, called “Region Servers,” and these serve data for reads and writes. A region server can serve about 1,000 regions.

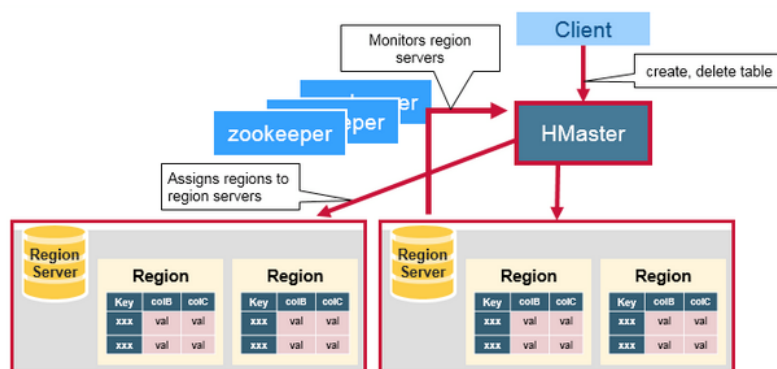


HBASE HMASTER

Region assignment, DDL (create, delete tables) operations are handled by the HBase Master.

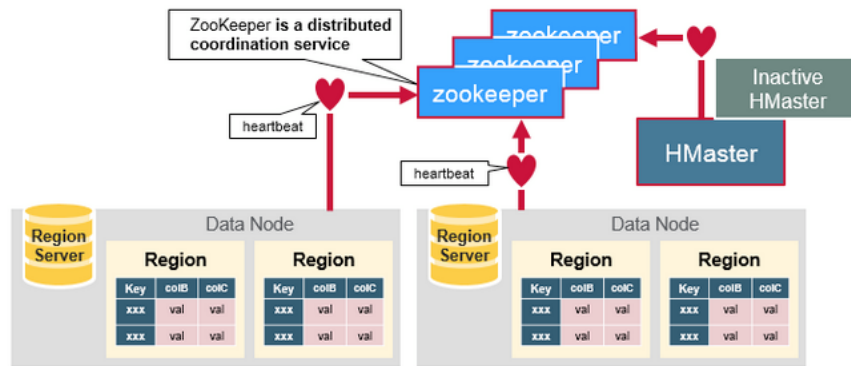
A master is responsible for:

- Coordinating the region servers
 - Assigning regions on startup , re-assigning regions for recovery or load balancing
 - Monitoring all RegionServer instances in the cluster (listens for notifications from zookeeper)
- Admin functions
 - Interface for creating, deleting, updating tables



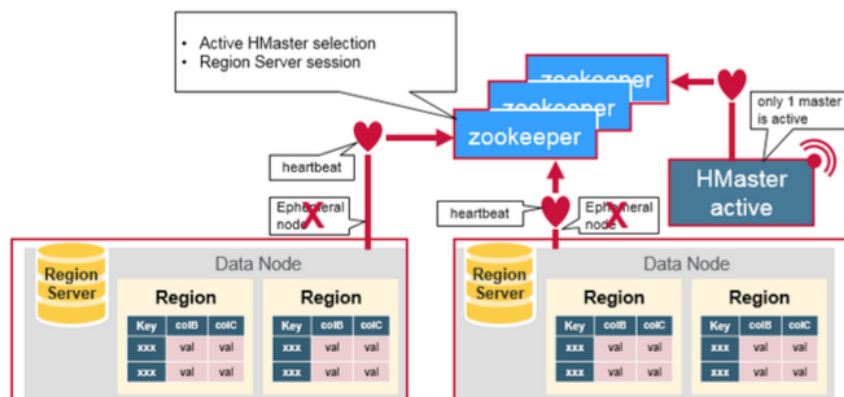
ZOOKEEPER: THE COORDINATOR

HBase uses ZooKeeper as a distributed coordination service to maintain server state in the cluster. Zookeeper maintains which servers are alive and available, and provides server failure notification. Zookeeper uses consensus to guarantee common shared state. Note that there should be three or five machines for consensus.



HOW THE COMPONENTS WORK TOGETHER

ZooKeeper is used to coordinate shared state information for members of distributed systems. Region servers and the active HMaster connect with a session to ZooKeeper. The ZooKeeper maintains ephemeral nodes for active sessions via heartbeats.



Each Region Server creates an ephemeral node. The HMaster monitors these nodes to discover available region servers, and it also monitors these nodes for server failures. HMasters vie to create an ephemeral node. Zookeeper determines the first one and uses it to make sure that only one master is active. The active HMaster sends heartbeats to Zookeeper, and the inactive HMaster listens for notifications of the active HMaster failure.

If a region server or the active HMaster fails to send a heartbeat, the session is expired and the corresponding ephemeral node is deleted. Listeners for updates will be notified of the deleted nodes. The active HMaster listens for region servers, and will recover region servers on failure. The Inactive HMaster listens for active HMaster failure, and if an active HMaster fails, the inactive HMaster becomes active.

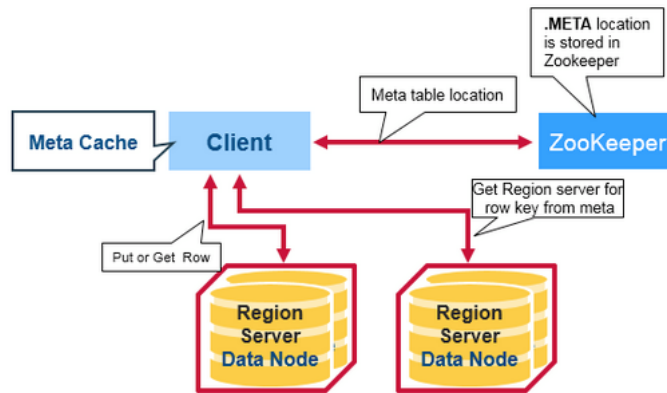
HBASE FIRST READ OR WRITE

There is a special HBase Catalog table called the META table, which holds the location of the regions in the cluster. ZooKeeper stores the location of the META table.

This is what happens the first time a client reads or writes to HBase:

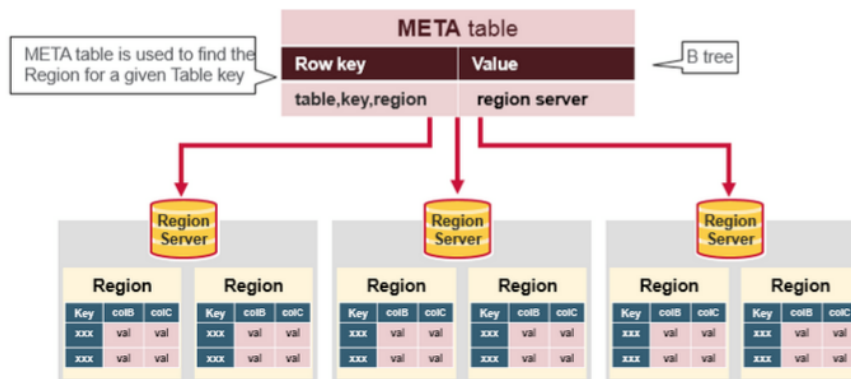
1. The client gets the Region server that hosts the META table from ZooKeeper.
2. The client will query the .META. server to get the region server corresponding to the row key it wants to access. The client caches this information along with the META table location.
3. It will get the Row from the corresponding Region Server.

For future reads, the client uses the cache to retrieve the META location and previously read row keys. Over time, it does not need to query the META table, unless there is a miss because a region has moved; then it will re-query and update the cache.



HBASE META TABLE

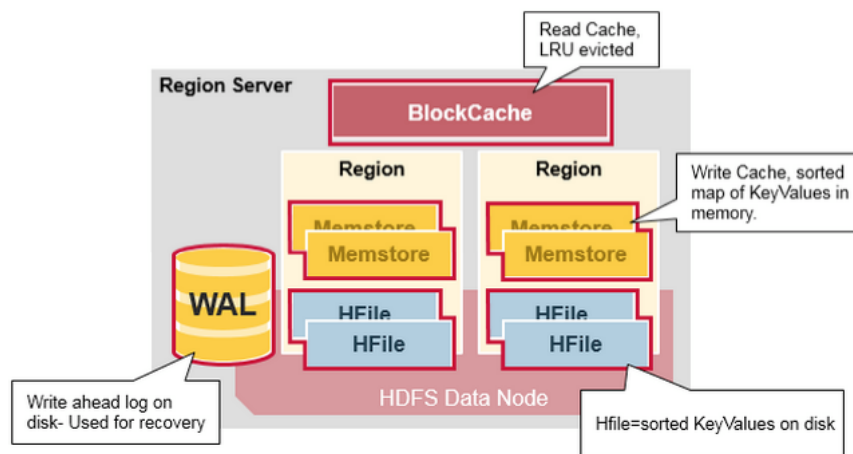
- This META table is an HBase table that keeps a list of all regions in the system.
- The .META. table is like a b tree.
- The .META. table structure is as follows:
 - Key: region start key, region id
 - Values: RegionServer



REGION SERVER COMPONENTS

A Region Server runs on an HDFS data node and has the following components:

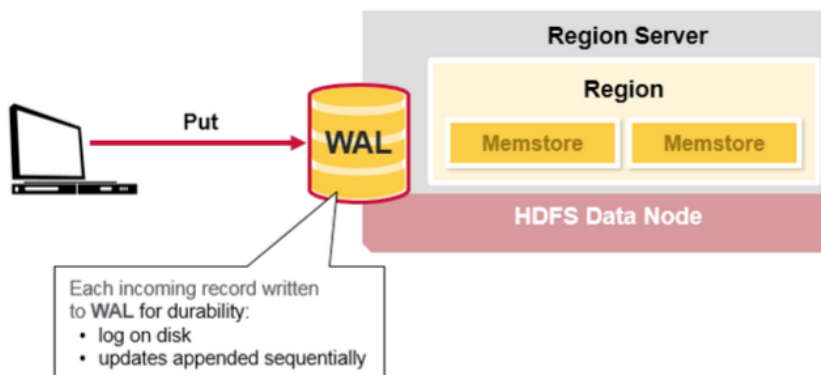
- WAL: Write Ahead Log is a file on the distributed file system. The WAL is used to store new data that hasn't yet been persisted to permanent storage; it is used for recovery in the case of failure.
- BlockCache: is the read cache. It stores frequently read data in memory. Least Recently Used data is evicted when full.
- MemStore: is the write cache. It stores new data which has not yet been written to disk. It is sorted before writing to disk. There is one MemStore per column family per region.
- Hfiles store the rows as sorted KeyValues on disk.



HBASE WRITE STEPS (1)

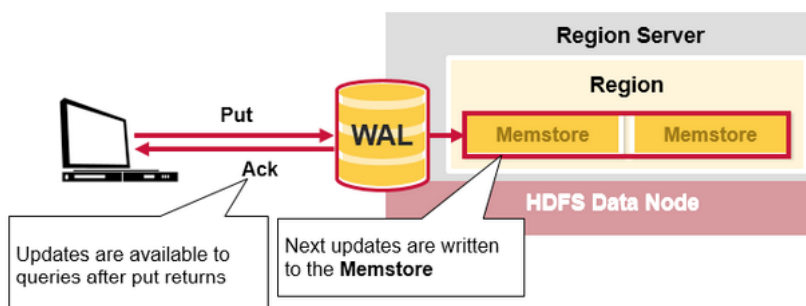
When the client issues a Put request, the first step is to write the data to the write-ahead log, the WAL:

- Edits are appended to the end of the WAL file that is stored on disk.
- The WAL is used to recover not-yet-persisted data in case a server crashes.



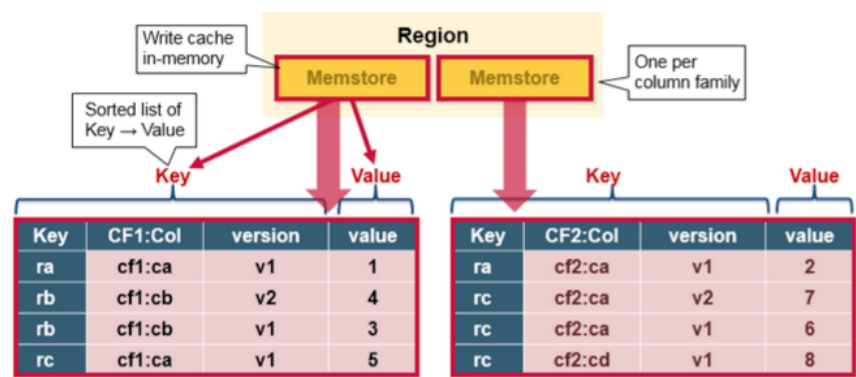
HBASE WRITE STEPS (2)

Once the data is written to the WAL, it is placed in the MemStore. Then, the put request acknowledgement returns to the client.



HBASE MEMSTORE

The MemStore stores updates in memory as sorted KeyValues, the same as it would be stored in an HFile. There is one MemStore per column family. The updates are sorted per column family.

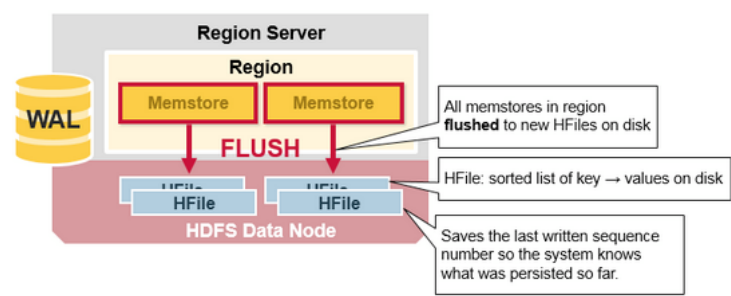


HBASE REGION FLUSH

When the MemStore accumulates enough data, the entire sorted set is written to a new HFile in HDFS. HBase uses multiple HFiles per column family, which contain the actual cells, or Key-Value instances. These files are created over time as Key-Value edits sorted in the MemStores are flushed as files to disk.

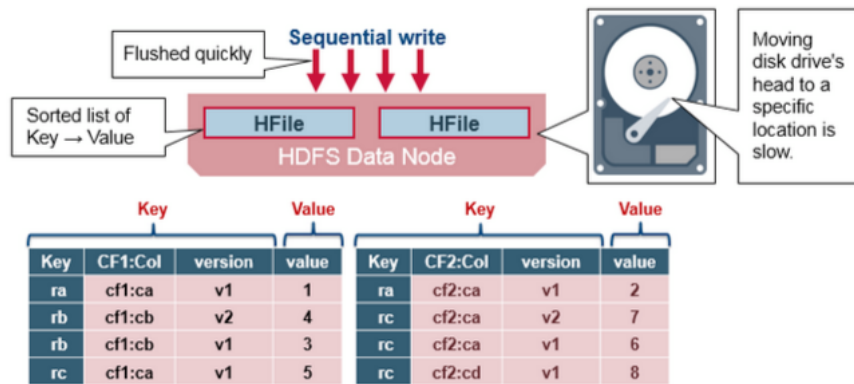
Note that this is one reason why there is a limit to the number of column families in HBase. There is one MemStore per CF; when one is full, they all flush. It also saves the last written sequence number so the system knows what was persisted so far.

The highest sequence number is stored as a meta field in each HFile, to reflect where persisting has ended and where to continue. On region startup, the sequence number is read, and the highest is used as the sequence number for new edits.



HBASE HFILE

Data is stored in an HFile which contains sorted key/values. When the MemStore accumulates enough data, the entire sorted Key-Value set is written to a new HFile in HDFS. This is a sequential write. It is very fast, as it avoids moving the disk drive head.

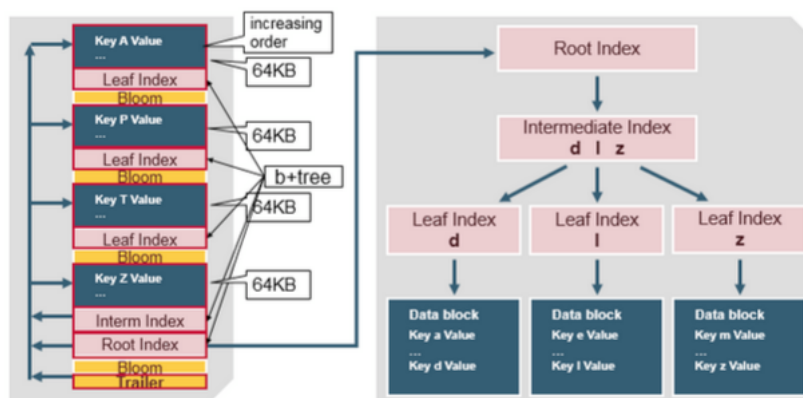


HBASE HFILE STRUCTURE

An HFile contains a multi-layered index which allows HBase to seek to the data without having to read the whole file. The multi-level index is like a b+tree:

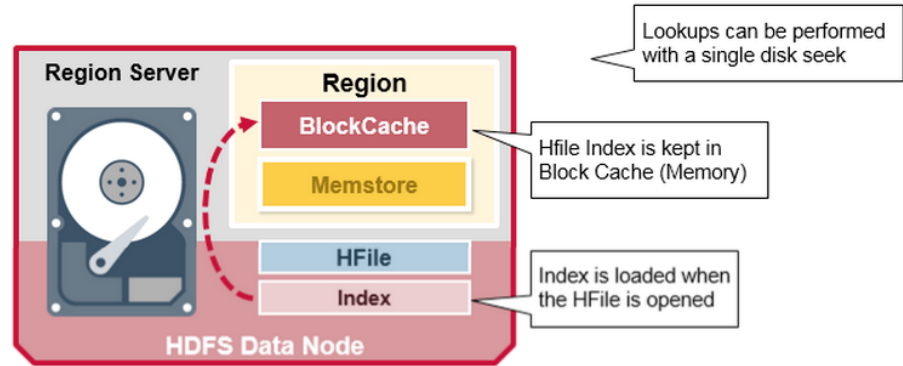
- Key value pairs are stored in increasing order
- Indexes point by row key to the key value data in 64KB "blocks"
- Each block has its own leaf-index
- The last key of each block is put in the intermediate index
- The root index points to the intermediate index

The trailer points to the meta blocks, and is written at the end of persisting the data to the file. The trailer also has information like bloom filters and time range info. Bloom filters help to skip files that do not contain a certain row key. The time range info is useful for skipping the file if it is not in the time range the read is looking for.



HFILE INDEX

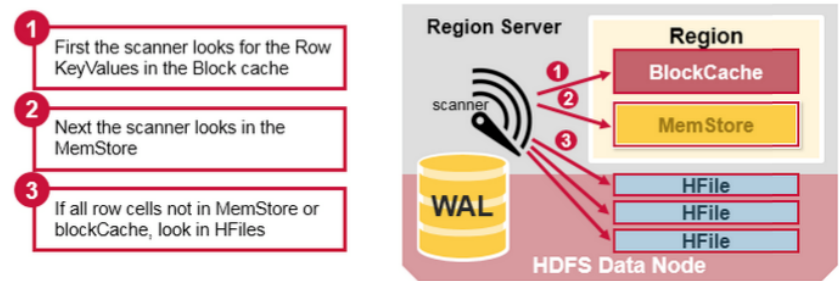
The index, which we just discussed, is loaded when the HFile is opened and kept in memory. This allows lookups to be performed with a single disk seek.



HBASE READ MERGE

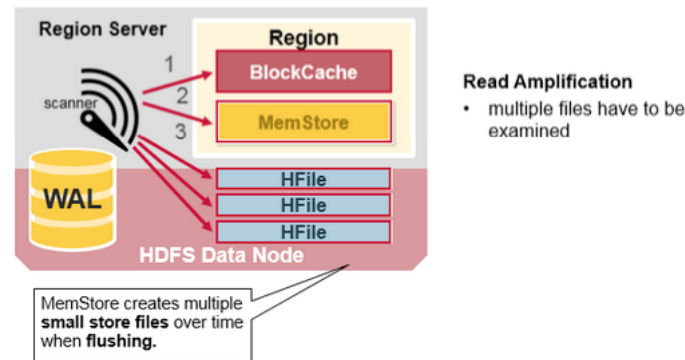
We have seen that the KeyValue cells corresponding to one row can be in multiple places, row cells already persisted are in Hfiles, recently updated cells are in the MemStore, and recently read cells are in the Block cache. So when you read a row, how does the system get the corresponding cells to return? A Read merges Key Values from the block cache, MemStore, and HFiles in the following steps:

- 1. First, the scanner looks for the Row cells in the Block cache - the read cache. Recently Read Key Values are cached here, and Least Recently Used are evicted when memory is needed.
- 2. Next, the scanner looks in the MemStore, the write cache in memory containing the most recent writes.
- 3. If the scanner does not find all of the row cells in the MemStore and Block Cache, then HBase will use the Block Cache indexes and bloom filters to load HFiles into memory, which may contain the target row cells.



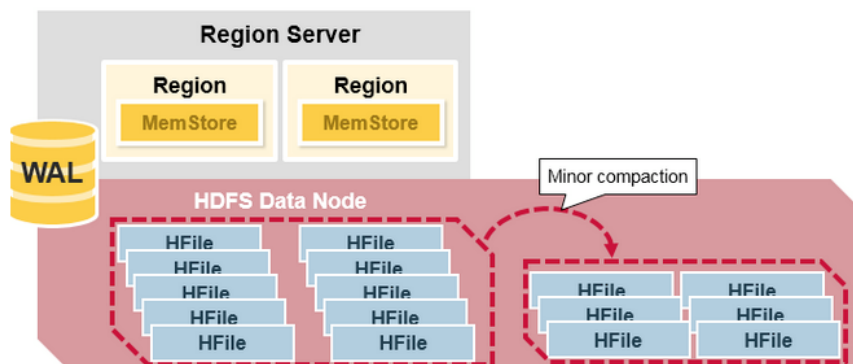
HBASE READ MERGE

As discussed earlier, there may be many HFiles per MemStore, which means for a read, multiple files may have to be examined, which can affect the performance. This is called read amplification.



HBASE MINOR COMPACTION

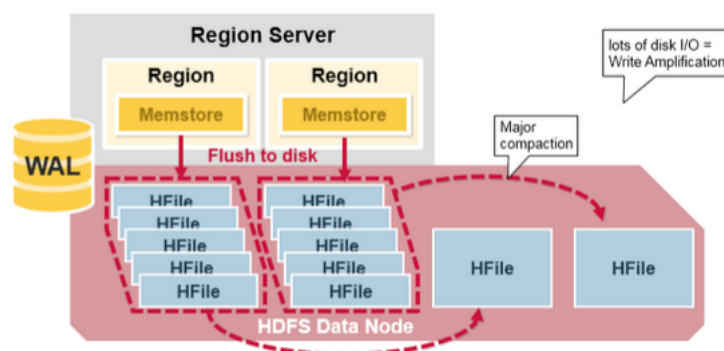
HBase will automatically pick some smaller HFiles and rewrite them into fewer bigger HFiles. This process is called minor compaction. Minor compaction reduces the number of storage files by rewriting smaller files into fewer but larger ones, performing a merge sort.



HBASE MAJOR COMPACTION

Major compaction merges and rewrites all the HFiles in a region to one HFile per column family, and in the process, drops deleted or expired cells. This improves read performance; however, since major compaction rewrites all of the files, lots of disk I/O and network traffic might occur during the process. This is called write amplification.

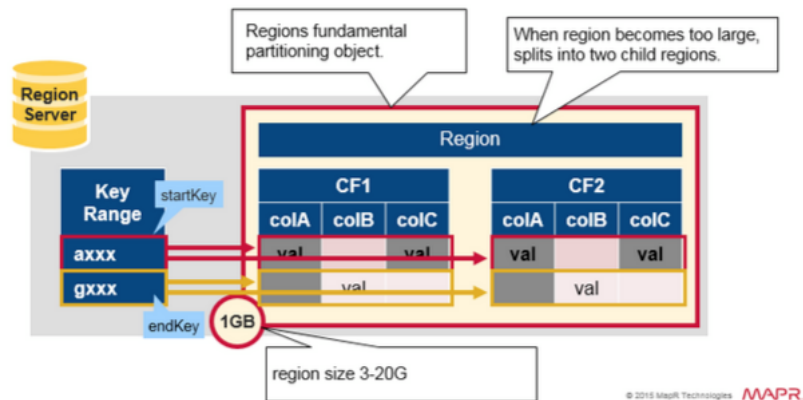
Major compactions can be scheduled to run automatically. Due to write amplification, major compactions are usually scheduled for weekends or evenings. Note that MapR-DB has made improvements and does not need to do compactions. A major compaction also makes any data files that were remote, due to server failure or load balancing, local to the region server.



REGION = CONTIGUOUS KEYS

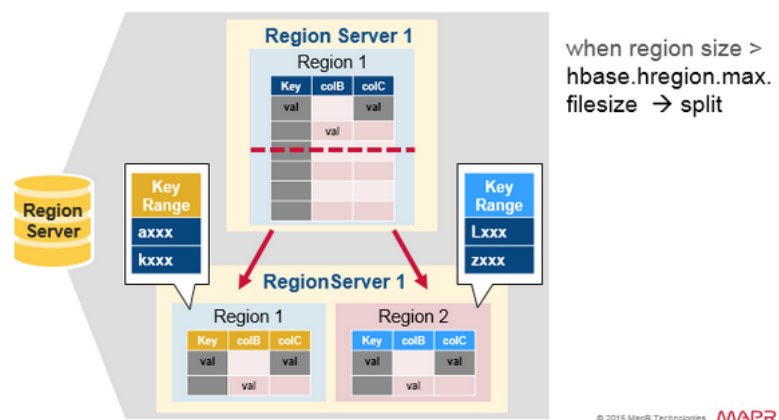
Let's do a quick review of regions:

- A table can be divided horizontally into one or more regions. A region contains a contiguous, sorted range of rows between a start key and an end key
- Each region is 1GB in size (default)
- A region of a table is served to the client by a RegionServer
- A region server can serve about 1,000 regions (which may belong to the same table or different tables)



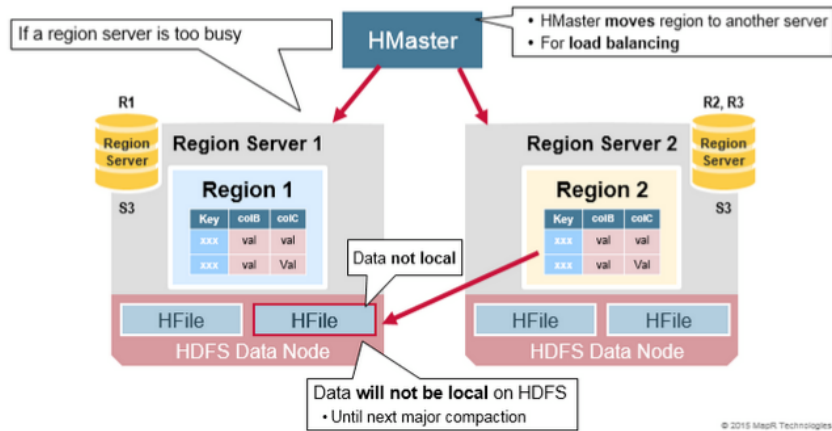
REGION SPLIT

Initially there is one region per table. When a region grows too large, it splits into two child regions. Both child regions, representing one-half of the original region, are opened in parallel on the same Region server, and then the split is reported to the HMaster. For load balancing reasons, the HMaster may schedule for new regions to be moved off to other servers.



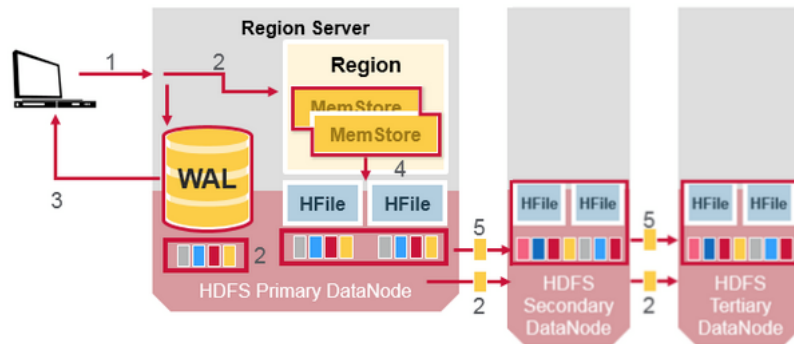
READ LOAD BALANCING

Splitting happens initially on the same region server, but for load balancing reasons, the HMaster may schedule for new regions to be moved off to other servers. This results in the new Region server serving data from a remote HDFS node until a major compaction moves the data files to the Regions server's local node. HBase data is local when it is written, but when a region is moved (for load balancing or recovery), it is not local until major compaction.



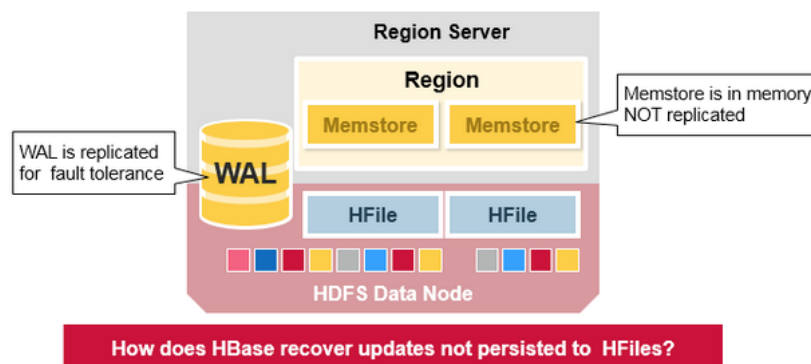
HDFS DATA REPLICATION

All writes and Reads are to/from the primary node. HDFS replicates the WAL and HFile blocks. HFile block replication happens automatically. HBase relies on HDFS to provide the data safety as it stores its files. When data is written in HDFS, one copy is written locally, and then it is replicated to a secondary node, and a third copy is written to a tertiary node.



HDFS DATA REPLICATION (2)

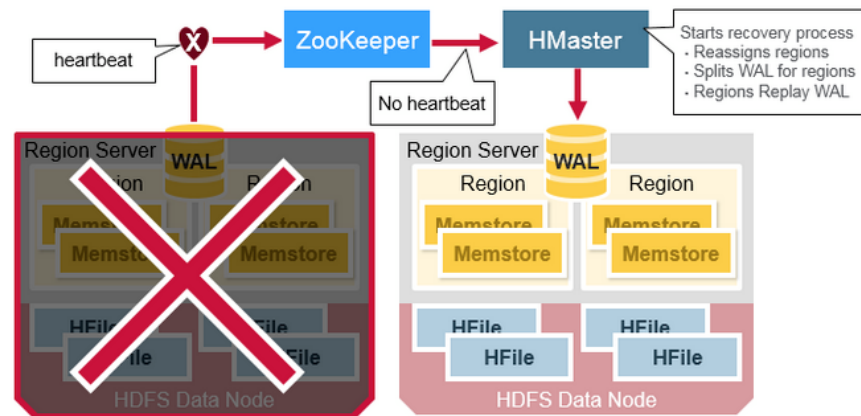
The WAL file and the Hfiles are persisted on disk and replicated, so how does HBase recover the MemStore updates not persisted to HFiles? See the next section for the answer.



HBASE CRASH RECOVERY

When a RegionServer fails, Crashed Regions are unavailable until detection and recovery steps have happened. Zookeeper will determine Node failure when it loses region server heart beats. The HMaster will then be notified that the Region Server has failed.

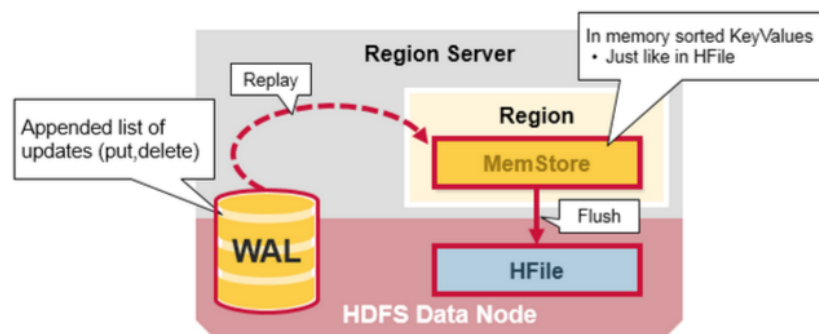
When the HMaster detects that a region server has crashed, the HMaster reassigns the regions from the crashed server to active Region servers. In order to recover the crashed region server's memstore edits that were not flushed to disk. The HMaster splits the WAL belonging to the crashed region server into separate files and stores these file in the new region servers' data nodes. Each Region Server then replays the WAL from the respective split WAL, to rebuild the memstore for that region.



DATA RECOVERY

WAL files contain a list of edits, with one edit representing a single put or delete. Edits are written chronologically, so, for persistence, additions are appended to the end of the WAL file that is stored on disk.

What happens if there is a failure when the data is still in memory and not persisted to an HFile? The WAL is replayed. Replaying a WAL is done by reading the WAL, adding and sorting the contained edits to the current MemStore. At the end, the MemStore is flush to write changes to an HFile.



APACHE HBASE ARCHITECTURE BENEFITS

HBase provides the following benefits:

- **Strong consistency model**
 - When a write returns, all readers will see same value
- **Scales automatically**
 - Regions split when data grows too large

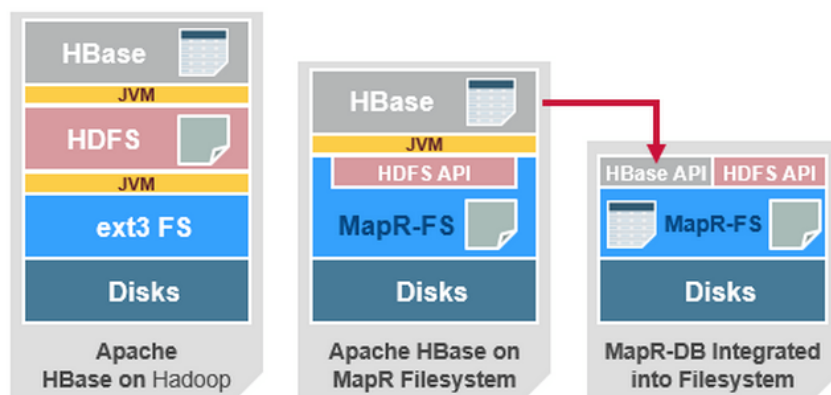
- Uses HDFS to spread and replicate data
- **Built-in recovery**
 - Using Write Ahead Log (similar to journaling on file system)
- **Integrated with Hadoop**
 - MapReduce on HBase is straightforward

APACHE HBASE HAS PROBLEMS TOO...

- **Business continuity reliability:**
 - WAL replay slow
 - Slow complex crash recovery
 - Major Compaction I/O storms

MAPR-DB WITH MAPR-FS DOES NOT HAVE THESE PROBLEMS

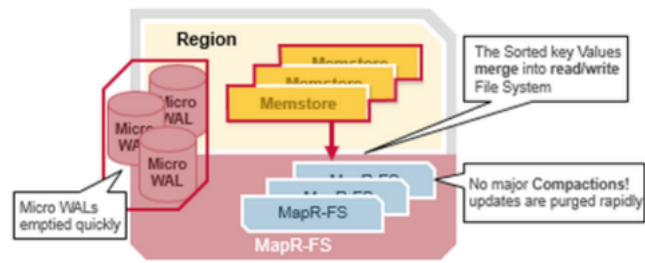
The diagram below compares the application stacks for Apache HBase on top of HDFS on the left, Apache HBase on top of MapR's read/write file system MapR-FS in the middle, and MapR-DB and MapR-FS in a Unified Storage Layer on the right.



MapR-DB exposes the same HBase API and the Data model for MapR-DB is the same as for Apache HBase. However the MapR-DB implementation integrates table storage into the MapR file system, eliminating all JVM layers and interacting directly with disks for both file and table storage.

MapR-DB offers many benefits over HBase, while maintaining the virtues of the HBase API and the idea of data being sorted according to primary key. MapR-DB provides operational benefits such as no compaction delays and automated region splits that do not impact the performance of the database. The tables in MapR-DB can also be isolated to certain machines in a cluster by utilizing the topology feature of MapR. The final differentiator is that MapR-DB is just plain fast, due primarily to the fact that it is tightly integrated into the MapR file system itself, rather than being layered on top of a distributed file system that is layered on top of a conventional file system.

KEY DIFFERENCES BETWEEN MAPR-DB AND APACHE HBASE



- Tables part of the MapR Read/Write File system
 - Guaranteed data locality
- Smarter load balancing
 - Uses container Replicas
- Smarter fail over
 - Uses container replicas
- Multiple small WALs
 - Faster recovery
- Memstore Flushes Merged into Read/Write File System
 - No compaction !

You can take this free On Demand training to learn more about MapR-FS and MapR-DB (/training/hadoop-demand-training/hde-110)

In this blog post, you learned more about the HBase architecture and its main benefits over NoSQL data store solutions. If you have any questions about HBase, please ask them in the comments section below.

WANT TO LEARN MORE?

- MapR provides Complete HBase Certification Curriculum as part of Free Hadoop On-
- Demand Training (/services/mapr-academy/big-data-hadoop-online-training)
 - Installing HBase on MapR (<http://doc.mapr.com/display/MapR/HBase>)
Getting Started with HBase on MapR
 - (<http://doc.mapr.com/display/MapR/Working+with+HBase>)
Release notes for HBase on MapR
 - (<http://doc.mapr.com/display/components/HBase+Release+Notes>)
Apache HBase docs (<https://issues.apache.org/jira/browse/HBASE/?report=com.atlassian.jira.jira-projects-plugin:roadmap-panel&selectedTab=com.atlassian.jira.jira-projects-plugin:summary-panel>)
HBase: The Definitive Guide, by Lars George
 - (<http://shop.oreilly.com/product/0636920014348.do>)

search

CATEGORIES

| | |
|--|--|
| All (/blog/) | Apache Drill (/en/blog/apache-drill) |
| Apache Hadoop (/en/blog/apache-hadoop) | Machine Learning (/en/blog/machine-learning) |
| No SQL (/en/blog/nosql) | Apache Spark (/en/blog/apache-spark) |
| Partners (/en/blog/partners) | Cloud Computing (/en/blog/cloud-computing) |
| Apache Hive (/en/blog/apache-hive) | MapReduce (/en/blog/mapreduce) |
| MapR Platform (/en/blog/mapr-platform) | Open Source Software (/en/blog/open-source-software) |
| Apache Mesos (/en/blog/apache-mesos) | Use Cases (/en/blog/use-cases) |

- Whiteboard Walkthrough Videos
(/en/blog/whiteboard-walkthrough-videos)
- Streaming (/en/blog/streaming)
- Enterprise Data Hub (/en/blog/enterprise-data-hub)
- Apache Myriad (/en/blog/apache-myriad)


First name

Email


SUBSCRIBE NOW

39 Comments mapr.com sachinjain024

Recommend 9 Share Sort by Best




Join the discussion...



jason zhang · 3 months ago

Absolute Awesome. Answered many of my question


^ | v · Reply · Share



Umesh Patil · 5 months ago

A nice Article. But i got a problem with my hbase when i m trying to read or write a large number of row in hbase its stuck in single node cluster any solution for that. m i missing something please help.


^ | v · Reply · Share



Carol McDonald → Umesh Patil · 5 months ago

maybe you need to redesign your row key so that the table splits, or pre-split your table? It's not clear from your question.


^ | v · Reply · Share



Umesh Patil → Carol McDonald · 5 months ago

when i m trying to read or write a large number of row in hbase its giving an exception as zookeeper session expired in my log file and in console it gave an exception as Memory leaked too many files.

^ | v · Reply · Share




sachinjain024 · 6 months ago

Very informative article. I must say!! One question.

> scanner does not find all of the row cells in the MemStore and Block Cache,

How does scanner know if all cells have been read with data present in MemStore and Block cache ?


^ | v · Edit · Reply · Share



Carol McDonald → sachinjain024 · 6 months ago

For even more detail , I recommend reading HBase: The Definitive Guide Random Access to Your Planet-Size Data By Lars George Publisher: O'Reilly Media


^ | v · Reply · Share



Carol McDonald → sachinjain024 · 6 months ago

it depends on the scan, or get. If you specify to get a specific rowkey and column then it knows to stop when it reads them . If you specify a broader scan then it will check indexes and bloom filters in memory , and may have to load more indexes from files .

^ | v · Reply · Share



Sarnath Kannan · 6 months ago

Undoubtedly, the best writeup in HBase i had seen so far. I was expecting a bit on scan performance. If every Put is going through Wal which is in HDFS, I believe write performance of

HBase really sucks... Is that so? Any pointers would help.

^ | v · Reply · Share ›



Carol McDonald → Sarnath Kannan · 6 months ago

thanks, writes are just appended to the WAL in HDFS, so writes are fast. Writes are not the problem with HBase, unless they happen during a compaction. Compactions, and java garbage collection can cause poor performance. MapR-DB DOES NOT have this problem. Also the WAL is large in HBase which means a long recovery to replay updates. MapR-DB also DOES NOT have this problem. To learn more take the HBase and MapR-DB training at learn.mapr.com

^ | v · Reply · Share ›



Sarnath Kannan → Carol McDonald · 6 months ago

Thanks for your time and comments. I still don't get how writing to WAL will be faster because it is in HDFS and writes to disk. If every single PUT goes to harddisk, how can it be faster? If we return before persisting to WAL, then it affects durability. Any comments?

I am aware of how MapR DB manages disks by itself and in the sense avoids compaction overheads etc. Good to know.

^ | v · Reply · Share ›



Carol McDonald → Sarnath Kannan · 6 months ago

Here is a lot of detail about HBase WAL writes, but like I said before, MapR-DB is different <http://www.larsgeorge.com/2...>

^ | v · Reply · Share ›



Carol McDonald → Sarnath Kannan · 6 months ago

appends are sequential writes to disk. sequential reads and writes are fast, what is slow is moving the disk head around for random updates , like updating indexes on disk which RDBMS do .

^ | v · Reply · Share ›



yassine · 6 months ago

a very good article, it helps a lot

^ | v · Reply · Share ›



Carol McDonald → yassine · 6 months ago

Merci!

^ | v · Reply · Share ›



akila · 7 months ago

In the section "HBase Read Merge", you mention that HBase attempts to satisfy read operations first through blockcache and then thru the memstore. Is it not the other way round? Memstore first then blockcache?

^ | v · Reply · Share ›



Carol McDonald → akila · 7 months ago

No first the Blockcache (read cache) then the memstore (write cache).

^ | v · Reply · Share ›



Alwyn Pereira · 8 months ago

This article has been of tremendous help! Thanks Carol.

Apache HBase has one WAL per Region server. By implication, write operations to regions (for one or more tables) hosted by the Region server are serialized?

For MapR, does having one WAL per hosted Region in turn result in higher parallel writes across Regions for the same table hosted in the Region server?

^ | v · Reply · Share ›



Carol McDonald → Alwyn Pereira · 7 months ago

Yes writes go to the primary region server. MapR-DB has multiple mini-WALs per region which makes the recovery much faster. Also the memstores flush is merged into a read-write file system , which means no compaction

^ | v · Reply · Share ›



Kanagha · 8 months ago

Excellent article! Clearly explains how data is stored and access pattern.

^ | v · Reply · Share ›



TechUser2011 · 9 months ago

Thanks for the great article. I have some questions:

1. In the section "Apache HBase Architecture Benefits," you state: "Strong consistency model: When a write returns, all readers will see the same value." How is this strong consistency provided? Could you please point out where in the article this is explained?
2. In the section "HBase Region Flush," you wrote: "Note that this is one reason why there is a limit to the number of column families in HBase." What is the "this" referring to in that sentence? Also, could you please expand on what are the other reasons for a limit on the number of column families?

^ | v · Reply · Share ·



Carol McDonald → TechUser2011 · 9 months ago

Strong Consistency is when all readers will see the same value as opposed to eventual consistency where readers may see different values. With HBase all writes and reads are to/from the primary server, and reads are not allowed until a write returns. This is different than cassandra where writes and reads can be from any of the replicas. You can read more about HBase Locking and Multiversion Concurrency Control here <https://blogs.apache.org/hb...>

There is one MemStore per CF; when one is full, they all flush to a new file. Compactions cause lots of I/O with HBase and having lots of column families creates more files to merge. MapR-DB does not do compactions so it does not have this problem.

^ | v · Reply · Share ·



KS Mahadevan · 10 months ago

Detailed explanation of the Region Servers architecture and thier functionalities. Thanks lot.

^ | v · Reply · Share ·



Zeeshan Ahmed · a year ago

One of the best articles on HBASE. Spot on and to the point. Thanks a lot!

^ | v · Reply · Share ·



Pratyay Pandey · a year ago

What is the merge strategy used during major compactions when HFiles span multiple column families? I guess it won't be a simple merge sort on rowkey since column family too needs to be in the sorted order so that data for the same family is sequential on the disk?

^ | v · Reply · Share ·



Ranjith Kumar · a year ago

Nicely and clear cut explanation of Hbase.

^ | v · Reply · Share ·



Sandeep Vaid · a year ago

Very Nice explanation of HBASE Architecture without going into too many implementation details.

^ | v · Reply · Share ·



Pratyay Pandey · a year ago

IO in Hbase happens at Hfile block level which is 64KB by default, this involves a disk seek and sequential copying of data to block cache. How does Minor and Major compactions reduce read latencies when the operation itself is atomic at block level?

^ | v · Reply · Share ·



Carol McDonald → Pratyay Pandey · a year ago

Each memstore flush writes a new HFile to disk which means for a read multiple HFiles may have to be examined to find the cells corresponding to the query. Look at the drawing, a Read is actually a merge between the memstore, read cache, and HFiles. Minor compaction reduces the number of storage files by rewriting smaller files into fewer but larger ones. Major compaction merges and rewrites all the HFiles in a region to one HFile per column family. Fewer HFiles reduces read latency

^ | v · Reply · Share ·



Pratyay Pandey → Pratyay Pandey · a year ago

Alright I guess the compactions make sure that the data associated with a particular row key is available within the same block, thereby, reducing unnecessary seeks when data is scattered across multiple blocks.

^ | v · Reply · Share ·



Igor Dincic · a year ago

This is by far the best explanation of HBase architecture that I've found online. It covers all the important architectural aspects, without digressing into irrelevant and tedious implementation details.

Many thanks, Carol!

^ | v · Reply · Share ·

**Carol McDonald** → Igor Dincic · a year ago

Thanks Igor!

^ | v · Reply · Share ›

**Abhi** · a year ago

Does Hbase need YARN?

^ | v · Reply · Share ›

**Carol McDonald** → Abhi · a year ago

HBase does not have to have YARN. YARN is for resource-management, it is needed by MapReduce and other data processing applications.

^ | v · Reply · Share ›

**Muhasin Cm** → Carol McDonald · a year ago

so hbase runs map -reduce programs itself?, it does not need yarn installed with map-reduce module.

^ | v · Reply · Share ›

**Pratyay Pandey** → Muhasin Cm · a year ago

HBase provides random access over HDFS it has got nothing to do with MR, for MRv1 the job gets submitted to JobTracker which assigns tasks for each node depending on the availability of data on the datanodes (the information it gets through namenode). Each task runs as an independent JVM process. These tasks while scanning the data make use of HBase capabilities.

^ | v · Reply · Share ›

**Chaitany Kulkarni** · 2 years ago

Sharding, de/normalized schema, projective pushdown in query execution plan optimization, working with foreign data, able to store data in JSON, Key-Value and relational structure along with Transactions, MVCC, etc. are some of the good features of PostgreSQL. It lacks full MPP support and clustering solution.

But you can always create and model your relational database as per HBase data model. You can create a separate table for each column family and share the same KEY across all such tables. When you join using LEFT JOIN all such tables in a query, Postgres will select data from only required tables without touching to other tables.

As per my opinion most of the the software developers never utilized full potential of RDBMS. A solution like PostgreSQL can scale very well. But DBA must be aware about how tune it perfectly. even you can put some data in HDD, some in SSD and some in memory in same instance of a PostgreSQL using table spaces. The problem is that no one is thinking of all possible combinations of the features.

As per my understanding HBase is good at scaling due to clustering and HDFS. HBase will even have his own limitations like poor performance when joining tables. Solution architects have to choose different components very carefully keeping in mind the future. Many thanks for both HBase Articles. They give really in-depth information.

^ | v · Reply · Share ›

**Carol McDonald** → Chaitany Kulkarni · a year ago

Thanks, yes HBase is good at scaling due to automatic sharding across a cluster. The HBase write steps make it very fast at writes, much faster than a relational database could be .

^ | v · Reply · Share ›

**Raja K Thaw** → Carol McDonald · a year ago

Nice to see WAL streams parallelization in Hbase , by partitioning incoming edits. But how that is achieved I can't see in a document.

^ | v · Reply · Share ›

**Carol McDonald** → Raja K Thaw · a year ago

This is not streams parallelization.

HBase has a WAL per Region Server. Incoming edits are first written to memory then appended to the WAL , which is a file on disk. The WAL is used for recovery. MapR-DB has Multiple small WALs per Regions, smaller WALs allows a faster recovery