Using HBase Effectively

Jonathan Hsieh | @jmhsieh | Software Engineer at Cloudera / HBase PMC Member
Himanshu Vashishtha | himanshu@cloudera | Software Engineer at Cloudera
February 2013
Who are we?

Jonathan Hsieh

- **Cloudera:**
  - Software Engineer
  - Apache HBase committer / PMC
  - Apache Flume founder / PMC
  - Apache Sqoop committer / PMC

- **U of Washington:**
  - Research in Distributed Systems

Himanshu Vashishtha

- **Cloudera:**
  - Software Engineer
  - Apache HBase contributor

- **U of Alberta, IIT Varanasi**
What is Apache HBase?

Apache HBase is an open source, distributed, scalable, consistent, low latency, random access non-relational database built on Apache Hadoop.
HBase provides Low-latency Random Access

- **Writes:**
  - 1-3ms, 1k-10k writes/sec per node

- **Reads:**
  - 0-3ms cached, 10-30ms disk
  - 10-40k reads / second / node from cache

- **Cell size:**
  - 0-3MB preferred

- Read, write and insert data anywhere in the table
  - No sequential write limitations
Production Apache HBase Applications

- Inbox
- Storage
- Web
- Search
- Analytics
- Monitoring

More Case Studies at http://www.hbasecon.com/agenda/
Inspiration: Google BigTable (OSDI 2006)

**Bigtable: A Distributed Storage System for Structured Data**

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

*Google, Inc.*

- **Goal:** Low latency, Consistent, random read/write access to massive amounts of **structured data**.
  - It was the data store for Google’s crawler web table, gmail, analytics, earth, blogger, ...
Implementation: Apache HBase (2013)

- Web Application Backend
  - Inboxes
  - Catalogs
  - Search index server
  - Web cache
  - Social Media storage
- Monitoring Real-time Analytics
  - OpenTSDB
- A data storage layer for higher level platforms
Outline

• Enter Apache HBase
• The HBase Data Model
• System Architecture
• Real-World Applications
• Effective Application Schemas
• Conclusions
Enter Apache HBase

Low Latency, Consistent, random read/write big data access
What is Apache HBase?

Apache HBase is an open source, horizontally scalable, consistent, low latency, random access data store built on top of Apache Hadoop.
HBase is Open Source

• **Apache 2.0 License**
• A **Community** project with committers and contributors from diverse organizations
  • Facebook, Cloudera, Salesforce.com, Huawei, TrendMicro, eBay, HortonWorks, Intel, Twitter ...
• Code license means anyone can modify and use the code.
HBase is Horizontally Scalable

- Adding more servers **linearly increases** performance and capacity
  - Storage capacity
  - Input/output operations
- Store and access data on 1-1000’s **commodity servers**

- **Largest cluster**: ~1000 nodes, ~1PB
- **Most clusters**: 10-40 nodes, 100GB-4TB
Commodity Servers (circa 2012)

- 12x 1TB hard disks in a **JBOD** (Just a Bunch Of Disks) configuration
- 2 quad core CPUs, 2+GHz
- 24-96GBs of RAM (96GBs if you’re considering HBase w/ MR)
- 2x 1Gigabit Ethernet
- $5k-10k / machine
What is Apache HBase?

Apache HBase is an open source, horizontally scalable, consistent, low latency, random access big-data store built on top of Apache Hadoop.
What is Apache HBase?

Apache HBase is an open source, horizontally scalable, consistent, low latency, random access big-data store built on top of Apache Hadoop.
HBase is Consistent

- Brewer’s CAP theorem
- **Consistency:**
  - DB-style ACID guarantees on rows
- **Availability:**
  - Favor recovering from faults over returning stale data
- **Partition Tolerance:**
  - If a node goes down, the system continues.
HBase depends on Apache Hadoop

Apache Hadoop is an open source, horizontally scalable system for reliably storing and processing massive amounts of data across many commodity servers.
HBase Dependencies

- **Apache Hadoop HDFS** for data **durability** and **reliability** (Write-Ahead Log)
- **Apache ZooKeeper** for distributed coordination
- **Apache Hadoop MapReduce** support built-in support for running MapReduce jobs
HBase On a Cluster

<table>
<thead>
<tr>
<th>HDFS NameNodes</th>
<th>ZooKeeper Quorum</th>
<th>Slave Boxes (DN + RS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBase Masters</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rack 1
- Name node

Rack 2
- Name node
Do I need an HBase or some other “NoSQL” data store?
Did you try scaling your RDBMS vertically?

- Upgrading to a beefier machines could be quick
  - (upgrade that m1.large to a m2.4xlarge)
- This is a good idea.
- What if this isn’t enough?
Changed your RDBMS schema and queries?

- Remove text search queries (LIKE)
- Remove joins
  - Joins due to Normalization require expensive seeks
- Remove foreign keys and encode your own relations
  - Avoid constraint checks
- Just put all parts of a query in a single table.

- Lots of Full table scans?
  - Good time for Hadoop.
- Time to consider HBase
Need to scale RDBMS reads?

• Using DB replication to make more copies to read from
• Use Memcached

• Assumes 80/20 read to write ratio, this works reasonably well if can tolerate replication lag.

• Unfortunately, eventually you may need more writes.
  • Replication has diminishing returns with more writes.
Need to scale RDBMS writes?

• Let’s shard and federate the DB
  • Loses consistency, order of operations.

• HA introduces operational complexity

• This is definitely a good time to consider HBase
We “optimized the DB” by discarding some fundamental SQL/relational databases features.
Outline

• Enter Apache HBase
• The HBase Data Model
• System Architecture
• Real-World Applications
• Effective Application Schemas
• Conclusions
The HBase Data Model

Rows and columns, gets and puts
Sorted Map Datastore

- It is a big table
- Tables consist of **rows**, each of which has a primary **row key**
- Each row has a set of **columns**
- Rows are stored in **sorted** order
Sorted Map Datastore
(logical view as “records”)

<table>
<thead>
<tr>
<th>Row key</th>
<th>info: height</th>
<th>info:state</th>
<th>roles:hadoop</th>
<th>roles:hbase</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutting</td>
<td>‘9ft’</td>
<td>‘CA’</td>
<td>‘Founder’</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tlipcon</td>
<td>‘5ft7’</td>
<td>‘CA’</td>
<td>‘PMC’ @ts=2011</td>
<td>‘Committer’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘Committer’ @ts=2010</td>
<td></td>
</tr>
</tbody>
</table>
# Sorted Map Datastore

(logical view as “records”)

Implicit PRIMARY KEY in RDBMS terms

<table>
<thead>
<tr>
<th>Row key</th>
<th>info: height</th>
<th>info:state</th>
<th>roles:hadoop</th>
<th>roles:hbase</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutting</td>
<td>’9ft’</td>
<td>’CA’</td>
<td>‘Founder’</td>
<td></td>
</tr>
<tr>
<td>tlipcon</td>
<td>’5ft7’</td>
<td>’CA’</td>
<td>‘PMC’ @ts=2011</td>
<td>‘Committer’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘Committer’ @ts=2010</td>
<td></td>
</tr>
</tbody>
</table>
Sorted Map Datastore
(logical view as “records”)

Implicit PRIMARY KEY in RDBMS terms

Data is all `byte[]` in HBase

<table>
<thead>
<tr>
<th>Row key</th>
<th>info: height</th>
<th>info:state</th>
<th>roles:hadoop</th>
<th>roles:hbase</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutting</td>
<td>‘9ft’</td>
<td>‘CA’</td>
<td>‘Founder’</td>
<td></td>
</tr>
<tr>
<td>tlipcon</td>
<td>‘5ft7’</td>
<td>‘CA’</td>
<td>‘PMC’ @ts=2011</td>
<td>‘Committer’ @ts=2010</td>
</tr>
</tbody>
</table>

| | | | roles:hadoop | roles:hbase |
| | | | | |
| | | | | |

Implicit PRIMARY KEY in RDBMS terms

Data is all `byte[]` in HBase

| | | | roles:hadoop | roles:hbase |
| | | | | |
| | | | | |

Implicit PRIMARY KEY in RDBMS terms

Data is all `byte[]` in HBase

| | | | roles:hadoop | roles:hbase |
| | | | | |
| | | | | |

Implicit PRIMARY KEY in RDBMS terms

Data is all `byte[]` in HBase

| | | | roles:hadoop | roles:hbase |
| | | | | |
| | | | | |

Implicit PRIMARY KEY in RDBMS terms

Data is all `byte[]` in HBase

| | | | roles:hadoop | roles:hbase |
| | | | | |
| | | | | |

Implicit PRIMARY KEY in RDBMS terms

Data is all `byte[]` in HBase

| | | | roles:hadoop | roles:hbase |
| | | | | |
| | | | | |

Implicit PRIMARY KEY in RDBMS terms

Data is all `byte[]` in HBase

| | | | roles:hadoop | roles:hbase |
| | | | | |
| | | | | |

Implicit PRIMARY KEY in RDBMS terms

Data is all `byte[]` in HBase

| | | | roles:hadoop | roles:hbase |
| | | | | |
| | | | | |
## Sorted Map Datastore

*(logical view as “records”)*

<table>
<thead>
<tr>
<th>Row key</th>
<th>info: height</th>
<th>info: state</th>
<th>roles:hadoop</th>
<th>roles:hbase</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutting</td>
<td>‘9ft’</td>
<td>‘CA’</td>
<td>‘Founder’</td>
<td></td>
</tr>
<tr>
<td>tlipcon</td>
<td>‘5ft7’</td>
<td>‘CA’</td>
<td>‘PMC’@ts=2011</td>
<td>‘Committer’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘Committer’@ts=2010</td>
<td></td>
</tr>
</tbody>
</table>

A single cell might have different values at different timestamps.

Implicit PRIMARY KEY in RDBMS terms

Data is all `byte[]` in HBase
Sorted Map Datastore
(logical view as “records”)

<table>
<thead>
<tr>
<th>Row key</th>
<th>info: height</th>
<th>info:state</th>
<th>roles:hadoop</th>
<th>roles:hbase</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutting</td>
<td>‘9ft’</td>
<td>‘CA’</td>
<td>‘Founder’</td>
<td></td>
</tr>
<tr>
<td>tlipcon</td>
<td>‘5ft7’</td>
<td>‘CA’</td>
<td>‘PMC’ @ts=2011</td>
<td>‘Committer’</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘Committer’ @ts=2010</td>
<td></td>
</tr>
</tbody>
</table>

Implicit PRIMARY KEY in RDBMS terms

Data is all byte[] in HBase

Different rows may have different sets of columns (table is sparse)

A single cell might have different values at different timestamps.
### Sorted Map Datastore

*(logical view as “records”)*

<table>
<thead>
<tr>
<th>Row key</th>
<th>info: height</th>
<th>info:state</th>
<th>roles:hadoop</th>
<th>roles:hbase</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutting</td>
<td>'9ft'</td>
<td>'CA'</td>
<td>'Founder'</td>
<td></td>
</tr>
<tr>
<td>tlipcon</td>
<td>'5ft7'</td>
<td>'CA'</td>
<td>'PMC' @ts=2011</td>
<td>'Committer'</td>
</tr>
</tbody>
</table>

A single cell might have different values at different timestamps.

Data is all `byte[]` in HBase.

Implicit PRIMARY KEY in RDBMS terms.

Column format family:qualifier.

Different rows may have different sets of columns (table is *sparse*).
Anatomy of a Row

• Each row has a **primary key**
  • Lexicographically sorted byte[]
  • Timestamp associated for keeping **multiple versions of data** (MVCC for consistency)

• Row is made up of columns.

• Each (row,column) referred to as a **Cell**
  • Contents of a cell are all byte[]’s.
  • Apps must “know” types and handle them.

• Rows are **Strongly consistent**
Access HBase data via an API

- Data operations
  - Get
  - Put
  - Scan
  - CheckAndPut
  - Delete
- DDL operations
  - Create
  - Alter
  - Enable/Disable
- Access via HBase shell, Java API, REST proxy
Example in Hbase Shell

```
hbase(main):024:0> scan 'people'
ROW
  cutting column=info:height, timestamp=1348305684946, value=9ft
  cutting column=info:state, timestamp=1348305750300, value=CA
  cutting column=roles:hadoop, timestamp=2006, value=founder
  jmhsien column=info:height, timestamp=1348305816300, value=5ft4
  jmhsien column=info:state, timestamp=1348305793015, value=CA
  jmhsien column=roles:hbase, timestamp=2017, value=committer
  tipcon column=info:height, timestamp=1348305797072, value=5ft7
  tipcon column=info:state, timestamp=1348305755965, value=CA
  tipcon column=roles:hadoop, timestamp=2011, value= PMC
  tipcon column=roles:hbase, timestamp=2010, value=committer
3 row(s) in 0.0530 seconds
```

2/26/13 Strata Conference 2013
Shell: Create a table, add, and read a row

```bash
[hbase(main):001:0]> create 'table', 'cf1', 'cf2'
0 row(s) in 1.8400 seconds

[hbase(main):002:0]> put 'table', 'row', 'cf1', 'value'
0 row(s) in 0.1160 seconds

[hbase(main):003:0]> scan 'table'
row
  COLUMN+CELL
column=cf1:, timestamp=1348305525318, value=value
1 row(s) in 0.0950 seconds
```
Java API

```java
byte[] row = Bytes.toBytes("row");
byte[] col = Bytes.toBytes("cf1");
byte[] putVal = Bytes.toBytes("your boat");
Configuration config = HBaseConfiguration.create();

HTable table = new HTable(config, "table");
Put p = new Put(row);
p.add(col, putVal)
table.put(p);

Get g = new Get(row);
Result r = table.get(g);
byte[] getValue = r.getValue(col);
assertEquals(putVal, getValue);
```
REST Server

- bin/hbase rest start –p 8070

Browse:
  - http://localhost:8070/ tables
  - http://localhost:8070/people/schema schema
  - http://localhost:8070/people/tlipcon row

- More info here
Simple API. What is the catch?

*With great power comes great responsibility.*
Cost Transparency

• Goal: Predictable latency of random read and write operations.
  • Now you have to understand some of the physical layout of your datastore.

• Efficiencies are based on **Locality** and your **schema**.
• Need to understand some physical concepts:
  • Column Families
  • Sparse Columns
  • Regions
  • Row Key
• Your schema needs to consider these.
Column Families

- **A Column family** is a set of related columns.
- Group sets of columns that have similar access patterns
- Select parameters to tune read performance per column family
## Physical Storage of Columns Families

### info Column Family

<table>
<thead>
<tr>
<th>Row key</th>
<th>Column key</th>
<th>Timestamp</th>
<th>Cell value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutting</td>
<td>info:height</td>
<td>1273516197868</td>
<td>9ft</td>
</tr>
<tr>
<td>cutting</td>
<td>info:state</td>
<td>1043871824184</td>
<td>CA</td>
</tr>
<tr>
<td>tlipcon</td>
<td>info:height</td>
<td>1273878447049</td>
<td>5ft7</td>
</tr>
<tr>
<td>tlipcon</td>
<td>info:state</td>
<td>1273616297446</td>
<td>CA</td>
</tr>
</tbody>
</table>

### roles Column Family

<table>
<thead>
<tr>
<th>Row key</th>
<th>Column key</th>
<th>Timestamp</th>
<th>Cell value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutting</td>
<td>roles:ASF</td>
<td>1273871823022</td>
<td>Director</td>
</tr>
<tr>
<td>cutting</td>
<td>roles:Hadoop</td>
<td>1183746289103</td>
<td>Founder</td>
</tr>
<tr>
<td>tlipcon</td>
<td>roles:Hadoop</td>
<td>1300062064923</td>
<td>PMC</td>
</tr>
<tr>
<td>tlipcon</td>
<td>roles:Hadoop</td>
<td>1293388212294</td>
<td>Committer</td>
</tr>
<tr>
<td>tlipcon</td>
<td>roles:Hive</td>
<td>1273616297446</td>
<td>Contributor</td>
</tr>
</tbody>
</table>

Each column family is contained in its own file.

Sorted on disk by Row key, Col key, descending timestamp.

Milliseconds since unix epoch.
Tuning Column Families

• Good for tuning **read performance**
  • Store related data together for better compression
  • Avoid polluting cache from another
  • Derived data can have different retention policies

• Column family parameters
  • Block Compression (none, gzip, LZO, Snappy)
  • Version retention policies
  • Cache priority
Sparse Columns

• Sparseness provides **schema flexibility**
  
  • Add columns later, no need to transform entire schema
  
  • If you find yourself adding columns to your db, HBase is a good model.
Sparse Columns

• Sparseness can improve performance
  • Null columns don’t take space. You don’t need to read what is not there.
  • If you have a traditional db table with lots of nulls, your data will probably fit well!
Horizontal Scaling - Regions

- Tables are divided into sets of rows called **regions**
- **Scale read and write capacity** by spreading across many regions.
Regions: Tradeoffs

- Easier to scale cluster capacity
  - Auto sharding and load balancing capability
- Greater throughput and storage capacity
  - Horizontal scalability of writes and reads and storage
- Enough consistency for many applications
  - Per row ACID guarantees

- No built-in atomic multi-row operations
- No built-in consistent secondary indices
- No built-in global time ordering
**SQL + HBase**

- No built-in SQL query language and query optimizer
- There is work on integration Apache Hive (SQL-like query language)
  - Currently not the optimal, x5 slower than normal Hive+HDFS
- Apache Sqoop and HBase Integration
  - Copy RDMS tables from database to Hbase
  - Copy HBase Tables into RDMS
  - (there is some impedance mismatch)
    - 2 hbase tables as to a primary/secondary tables in rdbms
- Impala integration: Work is on progress.
## HBase vs RDBMS

<table>
<thead>
<tr>
<th></th>
<th>RDBMS</th>
<th>HBase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data layout</strong></td>
<td>Row-oriented</td>
<td>Column-family-oriented</td>
</tr>
<tr>
<td><strong>Transactions</strong></td>
<td>Multi-row ACID</td>
<td>Single row only</td>
</tr>
<tr>
<td><strong>Query language</strong></td>
<td>SQL</td>
<td>get/put(scan)/etc *</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>Authentication/Authorization</td>
<td>Authentication / Column qualifier level Authorization</td>
</tr>
<tr>
<td><strong>Indexes</strong></td>
<td>On arbitrary columns</td>
<td>Row-key only*</td>
</tr>
<tr>
<td><strong>Max data size</strong></td>
<td>TBs</td>
<td>~1PB</td>
</tr>
<tr>
<td><strong>Read/write throughput limits</strong></td>
<td>1000s queries/second</td>
<td>Millions of “queries”/second</td>
</tr>
</tbody>
</table>
Outline

• Enter Apache HBase
• The HBase Data Model
• System Architecture
• Real-World Applications
• Effective Application Schemas
• Conclusions
System Architecture

Clusters, Nodes, and Dependencies

HBase Internals: MemStore-flush, Compaction
A Typical Look...

• 5-4000 commodity servers
  (8-core, 48 GB RAM, 12-24 TB, 1 gig-E)

• 2-level network architecture
  • 20-40 nodes per rack
HBase Processes

- HDFS
  - Name node
  - Data node
- ZooKeeper
  - Quorum Peer
- HBase
  - Master
  - Region Server
HDFS Nodes (physically)

HDFS NameNodes | ZooKeeper Quorum | Slave Boxes (DN)

Rack 1

Name node

Rack 2

Name node
HDFS Name Node

- Stores file system metadata on disk and in memory
  - Directory structures, permissions
  - Modifications stored as an edit log
- Fault tolerant and Highly Available
HDFS Data nodes

- HDFS splits the files into 128MB (or 256MB) blocks.
- **Data nodes** store and serve these Blocks:
  - By default, pipeline writes to 3 different machines.
  - By default, local machine, machines on other racks.
  - Locality helps significantly on subsequent reads and computation scheduling.
HDFS Nodes (physically)

<table>
<thead>
<tr>
<th>Rack 1</th>
<th>Rack 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name node</td>
<td>Name node</td>
</tr>
</tbody>
</table>

- **HDFS NameNodes**
- **ZooKeeper Quorum**
- **Slave Boxes (DN)**
HBase + HDFS Nodes (physically)

<table>
<thead>
<tr>
<th>HDFS NameNodes</th>
<th>ZooKeeper</th>
<th>Slave Boxes (DN + RS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBase Masters</td>
<td>Quorum</td>
<td></td>
</tr>
</tbody>
</table>

Rack 1

- Name node
- Name node
- HBase
- HDFS

Rack 2

- Name node
- Name node
- HBase
- HDFS

2/26/13 Strata Conference 2013
HMaster and ZooKeeper

• HMaster
  • Controls which Regions are served by which Region Servers.
  • Assigns regions to new region servers when they arrive or fail.
  • Standby master becomes the active master if original master goes down.
  • Transitions are coordinated by ZooKeeper

• Apache ZooKeeper
  • Highly Available System for coordination.
  • Generally 3 or 5 machines (always an odd number)
  • Uses consensus to guarantee common shared state.
  • Writes are considered expensive
Region Server

• Tables are chopped up into regions
• A region is only served by a single “region server” at a time.
• Region Server can serve multiple regions
  • Automatic load balancing if region server goes down.
• Co-locate region servers with data nodes.
  • Takes advantage of HDFS file locality
HBase + HDFS Nodes (physically)

<table>
<thead>
<tr>
<th>HDFS NameNodes</th>
<th>ZooKeeper Quorum</th>
<th>Slave Boxes (DN + RS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBase Masters</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rack 1
- Name node

Rack 2
- Name node
<table>
<thead>
<tr>
<th>HBase + HDFS: No SPOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS NameNodes</td>
</tr>
<tr>
<td>HBase Masters</td>
</tr>
<tr>
<td>ZooKeeper Quorum</td>
</tr>
<tr>
<td>Slave Boxes (DN + RS)</td>
</tr>
</tbody>
</table>

### Rack 1
- Name node
- HBase

### Rack 2
- Name node
- HBase
HBase Write Path: MemStore-flush/Compaction/Split

• Region level background processes
• Region?
  • a subset ([start-rowkey, end-rowkey]) of a table
  • Host data (all Column Families) for that range
• A Column Family
  • has a Memstore (in-memory, sorted map); contains last modified rows, and
  • HFile: on disk data file; created after flushing a Memstore
HBase Write Path

Region Server

HLog

HRegion

Store

MemStore

Put  Put  Del

HFile

Store

MemStore

Put  Del

HFile
HBase Write Path: Flush

Flush all MemStores in region to store files

All column families are flushed on flush call

Put Put Put Put Del ...
HBase Write Path: Flush

Region Server

HLog
Put Put Put Put Del ...

HRegion

Store
MemStore

Put Del HFile

Store
MemStore

Put Del HFile
There are too many HFiles and a Read can be inefficient. Let's combine them.
HBase Write Path: Compaction
HBase Write Path: Region Splitting

- After Compaction, a region size may change
- In case it becomes too large*, it splits into two child regions.
HBase Write Path: Region Splitting

This region has way more data than other regions. Load is now imbalanced!
Let’s Split this region into two and the share the load.
HBase Write Path: Region Splitting

RegionServer

HRegion A HRegion B

RegionServer

HRegion
HBase Write Path: Region Balancing

Still too much load on this RS
HBase Write Path: Region Balancing
Outline

• Enter Apache HBase
• The HBase Data Model
• System Architecture
• Real-World Applications
• Effective Application Schemas
• Conclusions
Real World Applications

How and where this infrastructure is being used
HBase Application Architecture

Web Application

Cluster boundary

Thrift/REST Gateway
HBase Client

Analysis:
MapReduce
HBase Client

Bulk Import:
MapReduce
HBase Client

Bulk Import:
MapReduce
HBase Client

Analysis:
MapReduce
HBase Client

HBase Application Architecture
**Example Apache HBase Applications**

- **Web Application Backends**
  - Inboxes: *Facebook Messages*, Tumblr
  - Catalogs: Intuit Mint Merchant DB, Gap Inc. Clothing Database, OLCL (world library catalog)
  - URL Shortener: StumbleUpon [http://su.pr](http://su.pr)
  - Search Index: *eBay Cassini*, PhotoBucket, YapMap

- **Massive datastore for Analysis**
  - *Mozilla Socorro* (crash report DB), Yahoo! Web Crawl Cache
  - Mignify / Internet Memory project

- **Monitoring Real-time Analytics**
  - *OpenTSDB*, Sproxil, Sematext

HBase Web Application

Cluster boundary

Web Application

Thrift/REST Gateway

HBase Client

... (bulk import: MapReduce)

HBase Client

HDFS

Analysis: MapReduce

HBase Client

2/26/13 Strata Conference 2013
RDBMS: Data-centric schema design

• Entity relational model.
  • Design schema in “Normalized form”
  • Figure out your queries
  • DBA to sets primary secondary keys once query is known

• Issues:
  • Join latency and cost can be difficult to predict
  • Difficult/Expensive to change schema / add columns
HBase: Query-centric schema design

• Know your queries then design your schema

• Column-family oriented
  • Create these by knowing fields needed by queries
  • Its better to have a fewer than many

• App developers optimize the queries, not DBAs

• If you’ve done the relational DB query optimizations, you are mostly there already!
Url Shortener Service

- Lookup hash, track click, and forward to full url
- Enter new long url, generate store to users’ mapping to short url
- Look up all of a users shortened urls and display
- Track historical click counts over time
Url Shortener schema

- All queries have at least one join
- Constraints when adding new urls, and short urls.
- How do we delete users?
Url Shortener schema

- All queries have at least one join
- Constraints when adding new urls, and short urls.
- How do we delete users?
Url Shortener HBase schema

- All single queries
- Use compression settings on content column families.
- Using rowkey to group all of a user’s shortened urls
- Consistency not guaranteed between tables.

1. Get url from url hash
2. Put to update click metrics

Figure 1-3. The Hush schema in HBase
Facebook Messages (as of 12/10)

- 15Bn/month message email, 1k = 14TB
- 120Bn/month, 100 bytes = 11TB
HBase Analysis Application Architecture

- Web Application
- Thrift/REST Gateway
- HBase Client
- HBase
- HDFS

Cluster boundary

Bulk Import: MapReduce
HBase Client

Analysis: MapReduce
HBase Client

2/26/13 Strata Conference 2013
**Example: Web Tables**

- **Goal**: Manage web crawls and its data by keeping snapshots of the web.
  - Google used BigTable for Web table example
  - Yahoo uses HBase for Web crawl cache
Hadoop MapReduce and HBase

- Use to scalably **process** data into HBase

- Create new data sets from raw data.
  - ETL into DBs/HBase

- High throughput batch processing
  - You would not serve live traffic from MR query or directly from HDFS.

- Users just write a “map” function and a “reduce” function.
MapReduce Processes

• Job Tracker
  • Schedules work and resource usage throughout the cluster
  • Makes sure work gets done
    • Controls retry, speculative execution, etc.

• Task Trackers
  • These slaves do the “map” and “reduce” work
  • Co-located with data nodes
Processing with Map Reduce

map | sort | shuffle

input | map | output

input | map

input | map

input | map

input | map

merge | reducer

reduce | output
Processing with Map Reduce With HBase

- One mapper per region / Bulk load output || individual puts
HBase + HDFS + MR Nodes (physically)

- HDFS NameNodes
- HBase Masters
- Hadoop MR Job Tracker
- ZooKeeper Quorum
- Slave Boxes (DN + RS + TT)

Rack 1
- Name node

Rack 2
- Name node
Data Loading Patterns

• Random Writes
  • Uses Put API
  • Simple
  • Low latency
  • Less throughput (more HBase overhead)

• Use Case:
  • Real-time Web apps
  • Real-time serving of data

• Example:
  • Inbox, url shortener

• Bulk import
  • Use MapReduce to generate HBase native files, atomically add metadata.
  • High Latency
  • High Throughput

• Use Case:
  • Large scale analysis
  • Generating derived data
  • ETL

• Examples:
  • Delayed Secondary indexes
  • Building search index
  • Exploring HBase Schemas
Schema Design Exploration

• Applications optimized by designing the structure of data in HBase
  • MR, HDFS and HBase complement each other.

• Exploration Steps:
  • Save raw data to HDFS/HBase.
  • MR for data transformation and ETL-like jobs from raw data.
  • Use bulk import from MR to HBase.
  • Serve data from HBase
## HBase vs just HDFS

<table>
<thead>
<tr>
<th></th>
<th>Plain HDFS/MR</th>
<th>HBase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstractions</strong></td>
<td>Files + bytes</td>
<td>Tables + Rows</td>
</tr>
<tr>
<td><strong>Write pattern</strong></td>
<td>Append-only</td>
<td>Random write, bulk incremental</td>
</tr>
<tr>
<td><strong>Read pattern</strong></td>
<td>Full file scan, partition table scan (hive)</td>
<td>Random read, small range scan, or table scan</td>
</tr>
<tr>
<td><strong>Structured storage</strong></td>
<td>Do-it-yourself / TSV / SequenceFile / Avro / ?</td>
<td>Sparse column-family data model</td>
</tr>
<tr>
<td><strong>Max data size</strong></td>
<td>30+ PB</td>
<td>~1PB</td>
</tr>
</tbody>
</table>

If you have neither random write nor random read, stick to HDFS!
Outline

- Enter Apache HBase
- The HBase Data Model
- System Architecture
- Real-World Applications
- Effective Application Schemas
- Conclusions
Effective Applications Schemas

Trade-offs, Schema Design, and Row key design
Designing Effective Application Schemas

- Characterize your application
- Understand HBase’s strengths
- Understanding Row key selection
- Experiment, Measure, and Repeat
Characterizing your workload

- Write heavy?
- Read:
  - Scan heavy?
  - Random read heavy?
- Update heavy?
- Sequential vs Random Accesses?
- Distribution of rowkey reads? (Uniform or Zipf?)
HBase’s performance Strengths

• Scaling out the number of writes handled
• Scaling out the number of reads handled
• Read workloads with short scans

• Techniques:
  • Pick the more efficient data arrangement
  • Minimize # of seeks (file accessed) per operation
  • Effectively balance work across nodes
  • Smart Row Key selection
How should I arrange my data?

- Isomorphic data represnetations!

### Short Fat Table using column qualifiers

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>jon</td>
<td>aaaa</td>
<td>bbbb</td>
<td>cccc</td>
<td>dddd</td>
</tr>
<tr>
<td>him</td>
<td>eeee</td>
<td>ffff</td>
<td>gggg</td>
<td>hhhhh</td>
</tr>
</tbody>
</table>

### Short Fat Table using column families

<table>
<thead>
<tr>
<th>Rowkey</th>
<th>col1:</th>
<th>col2:</th>
<th>col3:</th>
<th>col4:</th>
</tr>
</thead>
<tbody>
<tr>
<td>jon</td>
<td>aaaa</td>
<td>bbbb</td>
<td>cccc</td>
<td>dddd</td>
</tr>
<tr>
<td>him</td>
<td>eeee</td>
<td>ffff</td>
<td>gggg</td>
<td>hhhhh</td>
</tr>
</tbody>
</table>

### Tall skinny with compound rowkey

<table>
<thead>
<tr>
<th>rowkey</th>
<th>d:</th>
</tr>
</thead>
<tbody>
<tr>
<td>jon-col1</td>
<td>aaaa</td>
</tr>
<tr>
<td>jon-col2</td>
<td>bbbb</td>
</tr>
<tr>
<td>jon-col3</td>
<td>cccc</td>
</tr>
<tr>
<td>jon-col4</td>
<td>dddd</td>
</tr>
<tr>
<td>him-col1</td>
<td>eeee</td>
</tr>
<tr>
<td>him-col2</td>
<td>ffff</td>
</tr>
<tr>
<td>him-col3</td>
<td>gggg</td>
</tr>
<tr>
<td>him-col4</td>
<td>hhhh</td>
</tr>
</tbody>
</table>
HBase Read path

HBase client

Send get to regionserver

Region Server

Get

Server

HRegion

HStore

scanner

KV Scanner

Heap

Each CF is a new set of files and requires seeks

Each HFile (flush) may require seek
HBase Read path

Construct result from scan results

Blooms filters and scan filters can skip seeks or reduce seeks
HBase Read path

HBase client

Region Server

Result

Server

HRegion

HStore scanner

KV Scanner Heap

HStore scanner

KV Scanner Heap

Filtered results (a la DB push down predicate) send less data over the wire.

Return result.
Performance characteristics of writes

- Linear cost increase for adding more columns
  - At 10,100 bytes per value, overhead dominates.
  - At 1000 bytes per value, IO dominates
- Cheaper to have fewer columns and big values.
  - Write 1MM rows with 1000 bytes per row
  - Write 1MM rows with 100 bytes per row
HBase Schema Rules of Thumb

• More Writes:
  • Tall skinny tables preferred
  • Consolidate data into single cols if possible
  • Use Compression (Snappy/GZ/LZO)

• More Reads:
  • Use fewer column families
  • Use bloom filters
  • Use filters

<table>
<thead>
<tr>
<th>rowkey</th>
<th>d:</th>
</tr>
</thead>
<tbody>
<tr>
<td>jon-col1</td>
<td>aaaa</td>
</tr>
<tr>
<td>jon-col2</td>
<td>bbbb</td>
</tr>
<tr>
<td>jon-col3</td>
<td>cccc</td>
</tr>
<tr>
<td>jon-col4</td>
<td>dddd</td>
</tr>
<tr>
<td>him-col1</td>
<td>eeee</td>
</tr>
<tr>
<td>him-col2</td>
<td>ffff</td>
</tr>
<tr>
<td>him-col3</td>
<td>gggg</td>
</tr>
<tr>
<td>him-col4</td>
<td>hhhh</td>
</tr>
</tbody>
</table>
Proper load balancing

- Ideally 10-20 regions per region server
- Pre-split your regions
  - Creating a table by default creates one region that splits.
  - If you know your key distribution presplit so that writes and reads can be load balanced.
Proper load balance and Row Key Design

- Row Key design for schemas are critical
  - Make sure key distributes to spread write load.
  - Take advantage of lexicographic sort order.
Row key design techniques

• Numeric Keys and lexicographic sort
  • Store numbers big-endian.
  • Pad ASCII numbers with 0’s.

• Use **reversal** to have most significant traits first.
  • Reverse URL.
  • Reverse timestamp to get most recent first.
    • (MAX_LONG - ts) so “time” gets monotonically smaller.

• Use **composite keys** to make key distribute nicely and work well with sub-scans
  • Ex: User-ReverseTimeTimestamp
  • Do not use current timestamp as first part of row key!
Row Key exercise: Web Table

- Crawler continuously updating link and pages
- Want to track individual pages over time
- Want to group related pages from same site
- Want to calculate PageRank (links and backlinks)
- Want to do build a search index
- Want to do ad-hoc analytical queries on page content
### Google web table schema

#### Composite key:
- Reversed URLs '@' (MAXINT – millis for unix ts)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>com.cloudera.archive@2935290495</td>
<td>&lt;html&gt;....</td>
<td>CDH4 Homepage</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Com.cloudera.archive@2966912895</td>
<td>&lt;html&gt;....</td>
<td>CDH3 Home Page</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.cloudera.blog@2935280495</td>
<td>&lt;html&gt;....</td>
<td>Downloads</td>
<td>Tweet me!</td>
<td></td>
<td></td>
</tr>
<tr>
<td>com.cloudera.www@2935280495</td>
<td>&lt;html&gt;....</td>
<td>Home</td>
<td>@cloudera #strataconf</td>
<td>Patents</td>
<td></td>
</tr>
<tr>
<td>gov.uspto.www@2935280495</td>
<td>&lt;html&gt;....</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.apache.hbase@2935280495</td>
<td>&lt;html&gt;....</td>
<td>Open Source</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Uses of sparse column names in link col fam**

- Vendors
- Patents
- Open Source
- HBase
Key Take-aways

• Denormalized schema localized data for single lookups.

• Rowkey is critical for lookups and subset scans.

• Make sure when writing row keys are distributed.

• Use Bulk loads and Map Reduce to re-organize or change you schema (during down time).

• Multiple clusters for different workloads if you can afford it.
Performance benchmarks suites

- Hbase’s Performance Evaluation
  - Test some basic hbase workloads (random/sequential reads/writes)
    hbase org.apache.hadoop.hbase.PerformanceEvaluation

- Yahoo!’s Cloud Serving Benchmark (YCSB)
  - http://research.yahoo.com/Web_Information_Management/YCSB
  - Different query and write distributions: Uniform and Zipf
  - Records latencies and throughput.
    java -classpath `hbase classpath` com.yahoo.ycsb.Client -db com.yahoo.ycsb.db.HBaseClient
Outline

- Enter Apache HBase
- The HBase Data Model
- System Architecture
- Real-World Applications
- Effective Application Schemas
- Conclusions
Conclusions
Key takeaways

• Apache HBase is not a RDBMS! There are other scalable databases.
• Query-centric schema design, not data-centric schema design.
• In production at 100’s of TB scale at several large enterprises.
• If you are restructuring your SQL DB to scale it, you may be a candidate for HBase.
• HBase complements and depends upon Hadoop.
• New features focused for enterprise needs.
HBase vs other “NoSQL”

- Favors **Strong Consistency** over **Availability** (but availability is good in practice!)
- **Great Hadoop integration** (very efficient bulk loads, MapReduce analysis)
- **Ordered range partitions** (not hash)
- **Automatically shards/scales** (just turn on more servers, really proven at petabyte scale)
- **Sparse column storage** (not key-value)
## HBase vs just HDFS

<table>
<thead>
<tr>
<th></th>
<th>Plain HDFS/MR</th>
<th>HBase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Write pattern</strong></td>
<td>Append-only</td>
<td>Random write, bulk incremental</td>
</tr>
<tr>
<td><strong>Read pattern</strong></td>
<td>Full table scan, partition table scan</td>
<td>Random read, small range scan, or table scan</td>
</tr>
<tr>
<td><strong>Structured storage</strong></td>
<td>Do-it-yourself / TSV / SequenceFile / Avro / ?</td>
<td>Sparse column-family data model</td>
</tr>
<tr>
<td><strong>Max data size</strong></td>
<td>30+ PB</td>
<td>~1PB</td>
</tr>
</tbody>
</table>

If you have neither random write nor random read, stick to HDFS!
Deployment and Operations

• HBase interacts with many systems.
  • This is powerful –
    • Each component excels at a particular function
    • Live and in production at top 100 websites
  • This can be challenging–
    • Many knobs to tune
    • Isolated instances to make managing easier

• Monitor monitor monitor
More resources?

• Download Hbase (and Hadoop)!
  • CDH - Cloudera’s Distribution including Apache Hadoop
    http://cloudera.com/
  • http://hbase.apache.org/
• Try it out! (Locally, VM, or EC2)
Install HBase on your laptop

- Download and untar
  
  ```
  wget http://apache.osuosl.org/hbase/hbase-0.94.5/hbase-0.94.5.tar.gz
  tar xvfz hbase-0.94.5.tar.gz
  cd hbase-0.94.5
  bin/start-hbase.sh
  ```

- Verify:
  
  ```
  bin/hbase shell
  Browse http://localhost:60010
  ```