Building applications with HBase

Amandeep Khurana | Solutions Architect
Matteo Bertozzi | Software Engineer

October 2012
About AK

• Solutions Architect, Cloudera Inc
• Amazon Web Services
• Interested in large scale distributed systems
• Co-author, HBase In Action
• Twitter: amansk
I’m a wanna-be (author)

www.hbaseinaction.com
About MB

• Solutions Engineer, Cloudera Inc
• HBase
• Interested in file-systems and data storage
• Twitter: th30z
About the tutorial

• Install HBase
• Basic interactions with the shell
• Use of HBase Java API
• Data model
• HBase and Hadoop integration
Get HBase

• Get it here -> http://hbase.apache.org/
• Downloads -> Choose a mirror -> 0.94.1
• Download the file hbase-0.94.1.tar.gz
What’s Big Data?

... except another marketing term
Data management
Data has changed
“Big Data”
Hadoop - Big Data poster child

File System Mount
- FUSE-DFS

UI Framework
- HUE

Machine Learning
- APACHE MAHOUT

Data Integration
- APACHE FLUME, APACHE SQOOP

Fast Read/Write Access
- APACHE HBASE

Languages / Compilers
- APACHE PIG, APACHE HIVE, APACHE MAHOUT

Workflow Scheduling
- APACHE OOZIE

Metadata
- APACHE HIVE

FUSE-DFS

HDFS, MAPREDUCE

HUE

APACHE MAHOUT

APACHE HBASE

APACHE ZOOKEEPER
What is HBase?

- Scalable, distributed data store
- Sorted map of maps
- Open source avatar of Google’s Bigtable
- Sparse
- Multi dimensional
- Tightly integrated with Hadoop
- Not a RDBMS
So, what’s the big deal?

- Give up system complexity and sophisticated features for ease of scale and manageability
- Simple system != easy to build applications on top
- Goal: predictable read and write performance at scale
  - Effective API usage
  - Optimal schema design
  - Understand internals and leverage at scale
Use cases

• Canonical use case: storing crawl data and indices for search

Indexing the Internet
1 Crawlers constantly scour the Internet for new pages. Those pages are stored as individual records in Bigtable.
2 A MapReduce job runs over the entire table, generating search indexes for the Web Search application.

Searching the Internet
3 The user initiates a Web Search request.
4 The Web Search application queries the Search Indexes and retrieves matching documents directly from Bigtable.
5 Search results are presented to the user.
Use cases (continued)

- Capturing incremental data
- Content serving
- Information exchange
Ok, enough of high level stuff

- Big data is a buzzword but there is more to it
  - New products, applications, types of data
- Hadoop ecosystem is the poster child of big data architectures
  - Consists of several components
- HBase is a part of the ecosystem
  - Scalable database
  - Lots of production deployments
HBase - Getting Started

Installation and basic interaction
Installing HBase

- Downloads -> Choose a mirror -> 0.94.1
- Download the file `hbase-0.94.1.tar.gz`
- Untar it
  - `mkdir ~/strata`
  - `cp hbase-0.94.1.tar.gz ~/strata`
  - `cd ~/strata`
  - `tar xvf hbase-0.94.1.tar.gz`
Basics

- Modes of operation
  - Standalone
  - Pseudo-distributed
  - Fully distributed
- Start up in standalone mode
  - cd hbase-0.94.1
  - bin/start-hbase.sh
  - Logs are in ./logs/
- Check out [http://localhost:60010](http://localhost:60010) in your browser
Shell

• Basic interaction tool
• Written in JRuby
• Uses the Java client library
• Good place to start poking around
Shell (continued)

- bin/hbase shell
- Create table
  - create ‘mytable’, ‘cf1’
- List tables
  - list
- Describe table
  - describe ‘mytable’
Shell (continued)

- Put a row
  - put ‘mytable’, ‘row1’, ‘cf1:cq1’, ‘val1’
- Get a row
  - get ‘mytable’, ‘row1’
- Put more
- Get a row
  - get ‘mytable’, ‘row1’
- Scan table
  - scan ‘mytable’
Java API

... can’t really build an application just with the shell
Important terms

- Table
  - Consists of rows and columns
- Row
  - Has a bunch of columns.
  - Identified by a rowkey (primary key)
- Column Qualifier
  - Dynamic column name
- Column Family
  - Column groups - logical and physical (Similar access pattern)
- Cell
  - The actual element that contains the data for a row-column intersection
- Version
  - Every cell has multiple versions.
Introducing TwitBase

- Twitter clone built on HBase
- Designed to scale
- It’s an example, not a production app
- Start with users and twits for now
- Users
  - Have profiles
  - Post twits
- Twits
  - Have text posted by users
- Get sample code at: https://github.com/hbaseinaction/twitbase
Connecting to HBase

• Using default confs:
  ```java
  HTableInterface usersTable = new HTable("users");
  ```

• Passing custom conf:
  ```java
  Configuration myConf = HBaseConfiguration.create();
  myConf.set("parameter_name", "parameter_value");
  HTableInterface usersTable = new HTable(myConf, "users");
  ```

• Connection pooling
  ```java
  HTablePool pool = new HTablePool();
  HTableInterface usersTable = pool.getTable("users");
  // work with the table
  usersTable.close();
  ```
Inserting data

- **Put** call on HTable

```java
Put p = new Put(Bytes.toBytes("TheRealMT"));
p.add(Bytes.toBytes("info"),
        Bytes.toBytes("name"),
        Bytes.toBytes("Mark Twain"));
p.add(Bytes.toBytes("info"),
        Bytes.toBytes("email"),
        Bytes.toBytes("samuel@clemens.org"));
p.add(Bytes.toBytes("info"),
        Bytes.toBytes("password"),
        Bytes.toBytes("Langhorne"));

usersTable.put(p);
```
Data coordinates

- Row is addressed using rowkey
- Cell is addressed using
  [rowkey + family + qualifier]
Updating data

• Update == Write

```java
Put p = new Put(Bytes.toBytes("TheRealMT"));
p.add(Bytes.toBytes("info"),
    Bytes.toBytes("password"),
    Bytes.toBytes("abc123"));
usersTable.put(p);
```
Write path

Write goes to the write-ahead log (WAL) as well as the in memory write buffer called the Memstore. Client's don't interact directly with the underlying HFiles during writes.

Memstore gets flushed to disk to form a new HFile when filled up with writes.
Reading data

• Get the entire row
  
  Get g = new Get(Bytes.toBytes("TheRealMT"));
  Result r = usersTable.get(g);

• Get selected columns
  
  Get g = new Get(Bytes.toBytes("TheRealMT"));
  g.addColumn(Bytes.toBytes("info"),
           Bytes.toBytes("password"));
  Result r = usersTable.get(g);

• Extract the value out of the Result object
  
  byte[] b = r.getValue(Bytes.toBytes("info"),
                        Bytes.toBytes("email"));
  String email = Bytes.toString(b);
Reading data (continued)

- **Scan**
  ```java
  Scan s = new Scan(startRow, stopRow);
  ResultsScanner rs = twits.getScanner(s);
  for(Result r : rs) {
    // iterate over scanner results
  }
  ```

- Single threaded iteration over defined set of rows. Could be the entire table too.
What happens when you read data?
Deleting data

• Another simple API call

```java
Delete d = new Delete(Bytes.toBytes("TheRealMT"));
usersTable.delete(d);
```

• Delete entire row

```java
Delete d = new Delete(Bytes.toBytes("TheRealMT"));
usersTable.delete(d);
```

• Delete selected columns

```java
Delete d = new Delete(Bytes.toBytes("TheRealMT"));
d.deleteColumns(Bytes.toBytes("info"),
                Bytes.toBytes("email"));
usersTable.delete(d);
```
Delete internals

- Tombstone markers
- Deletion only happens during major compactions
- Compactions
  - Minor
  - Major
Two or more HFiles get combined to form a single HFile during a compaction.
Hands-on: Java client to HBase

• Create a table from the HBase shell
  • create ‘users’, ‘info’

• Write a program to do the following
  • Add 10 users to the ‘users’ table with userid as rowkey and user’s name in the column ‘info:name’. userid could just be user + serial number. eg: user1, user2, user3
  • Get a user from the ‘users’ table and display the result on the screen
  • Scan the entire ‘users’ table and display the result on the screen
DAO

- Like any other application
- Makes data access management easy
  - Cleaner code
  - Table info at a single place
- Optimizations like connection pooling etc
- Easier management of database configs
Admin

HBaseAdmin
• Administrative API to the cluster.
• Provides an interface to manage HBase database tables metadata and general administrative functions.

Table Operations
• createTable()
• deleteTable()
• addColumn()
• modifyColumn()
• deleteColumn()
• listTables()
Data Model

It’s not a relational database system
HBase is ...

- Column family oriented database
  - Column family oriented
  - Tables consisting of rows and columns
- Persisted Map
  - Sparse
  - Multi dimensional
  - Sorted
  - Indexed by rowkey, column and timestamp
- Key Value store
  - [rowkey, col family, col qualifier, timestamp] -> cell value
HBase is not ...

- A relational database
  - No SQL query language
  - No joins
  - No secondary indexing
  - No transactions
Tabular representation

The table is lexicographically sorted on the rowkeys.

<table>
<thead>
<tr>
<th>Rowkey</th>
<th>Column Family - Info</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrandpaD</td>
<td>Mark Twain</td>
<td>abc123</td>
</tr>
<tr>
<td>HMS_Surprise</td>
<td>Patrick O'Brien</td>
<td>abc123</td>
</tr>
<tr>
<td>SirDoyle</td>
<td>Fyodor Dostoyevsky</td>
<td>abc123</td>
</tr>
<tr>
<td>TheRealMT</td>
<td>Sir Arthur Conan Doyle</td>
<td><a href="mailto:art@TheQueensMen.co.uk">art@TheQueensMen.co.uk</a></td>
</tr>
</tbody>
</table>

The coordinates used to identify data in an HBase table are: (1) rowkey, (2) column family, (3) column qualifier, (4) version.

Cells

Each cell has multiple versions, typically represented by the timestamp of when they were inserted into the table (ts2>ts1).

Cells

| ts1=1329088321289 | ts2=1329088818321 |

Cells
Key-Value store

A single `KeyValue` instance
Key-Value store

1. Start with coordinates of full precision

2. Drop version and you're left with a map of version to values

3. Omit qualifier and you have a map of qualifiers to the previous maps

4. Finally, drop the column family and you have a row, a map of maps
Sorted map of maps

```
{  
  "Rowkey": "TheRealMT",  
  "Column family": {  
    "info": {  
      "email": {  
        1329088321289: "samuel@clemens.org",  
      },  
      "name": {  
        1329088321289: "Mark Twain",  
      },  
      "password": {  
        1329088818321: "abc123",  
        1329088321289: "Langhorne"  
      }  
    }  
  }  
}
HFiles and physical data model

- HFiles are
  - Immutable
  - Sorted on rowkey + qualifier + timestamp
  - In the context of a column family per region

```
"TheRealMT", "info", "email", 1329088321289, "samuel@clemens.org"
"TheRealMT", "info", "name", 1329088321289, "Mark Twain"
"TheRealMT", "info", "password", 132908818321, "abc123",
"TheRealMT", "info", "password", 1329088321289, "Langhorne"
```

HFile for the info column family in the users table
HBase in distributed mode

... what really goes on under the hood
## Distributed mode

- Table is split into Regions

### Table T1

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001</td>
<td>John</td>
<td>415-111-1234</td>
</tr>
<tr>
<td>00002</td>
<td>Paul</td>
<td>408-432-9922</td>
</tr>
<tr>
<td>00003</td>
<td>Ron</td>
<td>415-993-2124</td>
</tr>
<tr>
<td>00004</td>
<td>Bob</td>
<td>818-243-9988</td>
</tr>
<tr>
<td>00005</td>
<td>Carly</td>
<td>206-221-9123</td>
</tr>
<tr>
<td>00006</td>
<td>Scott</td>
<td>818-231-2566</td>
</tr>
<tr>
<td>00007</td>
<td>Simon</td>
<td>425-112-9877</td>
</tr>
<tr>
<td>00008</td>
<td>Lucas</td>
<td>415-992-4432</td>
</tr>
<tr>
<td>00009</td>
<td>Steve</td>
<td>530-288-9832</td>
</tr>
<tr>
<td>00010</td>
<td>Kelly</td>
<td>916-992-1234</td>
</tr>
<tr>
<td>00011</td>
<td>Betty</td>
<td>650-241-1192</td>
</tr>
<tr>
<td>00012</td>
<td>Anne</td>
<td>206-294-1298</td>
</tr>
</tbody>
</table>

### Table T1 split into 3 regions - R1, R2 and R3

#### R1

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001</td>
<td>John</td>
<td>415-111-1234</td>
</tr>
<tr>
<td>00002</td>
<td>Paul</td>
<td>408-432-9922</td>
</tr>
<tr>
<td>00003</td>
<td>Ron</td>
<td>415-993-2124</td>
</tr>
</tbody>
</table>

#### R2

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>00005</td>
<td>Carly</td>
<td>206-221-9123</td>
</tr>
<tr>
<td>00006</td>
<td>Scott</td>
<td>818-231-2566</td>
</tr>
</tbody>
</table>

#### R3

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>00009</td>
<td>Steve</td>
<td>530-288-9832</td>
</tr>
<tr>
<td>00010</td>
<td>Kelly</td>
<td>916-992-1234</td>
</tr>
<tr>
<td>00012</td>
<td>Anne</td>
<td>206-294-1298</td>
</tr>
</tbody>
</table>
Distributed mode (continued)

- Regions are served by RegionServers
- RegionServers are Java processes, typically collocated with the DataNodes
Finding your region

- Two special tables: -ROOT- and .META.

-ROOT- table, consisting of only 1 region hosted on region server RS1

.META. table, consisting of 3 regions, hosted on RS1, RS2 and RS3

User tables T1 and T2, consisting of 4 and 3 regions respectively, distributed amongst RS1, RS2 and RS3
Regions distributed across the cluster

RegionServer 1 (RS1), hosting regions R1 of the user table T1 and M2 of .META.

RegionServer 2 (RS2), hosting regions R2, R3 of the user table and M1 of .META.

RegionServer 3 (RS3), hosting just -ROOT-
What the client does

-ROOT-

ZooKeeper

Row per META region

.META.

Row per table region

MyTable

TheRealMT
HBase and Hadoop

... they play well together
HBase and Hadoop

- HBase has tight integration with Hadoop
- Two separate concerns
  - Access via MapReduce
  - Storage on HDFS
Parallel processing
MapReduce
HBase and MapReduce

- HBase is tightly integrated with the Hadoop stack
  - MapReduce
  - HDFS
- Source for MapReduce jobs
  - Read from an HBase Table with a Map-Reduce Job (Export, Row Count, ...)
- Sink for MapReduce jobs
  - Writing to an HBase Table from a Map-Reduce Job
- Lookups during MapReduce jobs
  - Join between hbase and/or another source
HBase as MR source

• One mapper per region
• **Mapper**
  ```java
  protected void map(ImmutableBytesWritable rowkey, Result result,
                     Context context) {
    // mapper logic goes here
  }
  ```

• **Job setup**
  ```java
  Scan scan = new Scan();
  scan.addColumn(Bytes.toBytes("twits"),
                 Bytes.toBytes("twit"));
  TableMapReduceUtil.initTableMapperJob("twits", scan,
                                          Map.class, ImmutableBytesWritable.class, Result.class,
                                          job);
  ```
HBase as MR sink

- Write to regions via MR job

Reduce tasks writing to HBase regions. Reduce tasks don’t necessarily write to a region on the same physical host. They will write to whichever region contains the key range that they are writing into. This could potentially mean that all reduce tasks talk to all regions on the cluster.
HBase as MR sink (continued)

• Reducer
  protected void reduce(ImmutableBytesWritable rowkey,
    Iterable<Put> values, Context context) {
    // reduce logic goes here
  }

• Job setup
  TableMapReduceUtil.initTableReducerJob("users",
    Reducer.class, job);
HBase as a shared resource

- Lookups from MR jobs
- Map-side joins
- Simple Get/Scan calls

[Diagram of HBase and its components, including Region Servers and Datanodes, with arrows indicating data flow and interaction with HBase.]
Integration with HDFS

- HDFS is a filesystem
- Underlying storage fabric for HBase
- HBase tightly integrated with HDFS and leverages the semantics
  - LSM trees fit well into a write once filesystem
- Availability (fault tolerance) and reliability (durability) at scale
Availability and fault tolerance

Disaster strikes on the host which was serving region R1

Another server picks up the region and starts serving it from the persisted HFile from HDFS. The lost copy of the HFile will get replicated to another Datanode.
Durability

- WAL keeps writes intact even if RS fails
- HDFS is reliable and doesn’t lose data if individual nodes fail
Summary

• Big data = new products, applications
• Hadoop ecosystem - poster child
• HBase serves many use cases
• Allows to solve aspects of big data problems
• Simple Java API
• Tight Hadoop integration
• Low latency access to large amounts of data
• Large scale computation using MR
Schema design

... it’s a database after-all
But isn’t HBase schema-less?

- Number of tables
- Rowkey design
- Number of column families per table. What goes into what column family
- Column qualifier names
- What goes into the cells
- Number of versions
Rowkeys

• Rowkey design is the single most important aspect of HBase table designs
• The only way to address rows in HBase
TwitBase relationships

- Users follow users
- Relationships need to be persisted for usage later on
- Model tables for the expected access patterns
- Read pattern
  - Who does A follow?
  - Who follows A?
  - Does A follow B?
- Write pattern
  - A follows B
  - A unfollows B
Start simple

• Adjacency list

<table>
<thead>
<tr>
<th>Col Qualifier</th>
<th>Cell value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TheFakeMT</td>
<td>1: TheRealMT</td>
</tr>
<tr>
<td></td>
<td>2: MTFanBoy</td>
</tr>
<tr>
<td>TheRealMT</td>
<td>1: HRogers</td>
</tr>
<tr>
<td></td>
<td>2: Olivia</td>
</tr>
<tr>
<td></td>
<td>3: Olivia</td>
</tr>
<tr>
<td></td>
<td>4: HRogers</td>
</tr>
</tbody>
</table>
Optimizing the adjacency list

- We need a count
- Where does a new followed user go?

<table>
<thead>
<tr>
<th>follows</th>
</tr>
</thead>
<tbody>
<tr>
<td>TheFakeMT 1: TheRealMT 2: MTFanBoy 3: Olivia 4: HRogers count: 4</td>
</tr>
<tr>
<td>TheRealMT 1: HRogers 2: Olivia count: 2</td>
</tr>
</tbody>
</table>
Adding a new user

Client code:
Step 1: Get current count
Step 2: Update count
Step 3: Add new entry
Step 4: Write the new data to HBase

Row that needs to be updated

<table>
<thead>
<tr>
<th>follows</th>
</tr>
</thead>
<tbody>
<tr>
<td>TheFakeMT</td>
</tr>
<tr>
<td>1: TheRealMT</td>
</tr>
<tr>
<td>2: MTFanBoy</td>
</tr>
<tr>
<td>3: Olivia</td>
</tr>
<tr>
<td>4: HRogers</td>
</tr>
<tr>
<td>count: 4</td>
</tr>
<tr>
<td>TheRealMT</td>
</tr>
<tr>
<td>1: HRogers</td>
</tr>
<tr>
<td>2: Olivia</td>
</tr>
<tr>
<td>count: 2</td>
</tr>
</tbody>
</table>

1. TheFakeMT : follows: {count -> 4}
   - increment count

2. TheFakeMT : follows: {count -> 5}
   - add new entry

3. TheFakeMT : follows: {5 -> MTFanBoy2, count -> 5}
   - write the new data to HBase
Transactions == not good

- HBase doesn’t have native support (think scale)
- Don’t want to complicate client side logic
- Only solution -> simplify schema

<table>
<thead>
<tr>
<th></th>
<th>TheFakeMT</th>
<th>TheRealMT:1</th>
<th>MTFanBoy:1</th>
<th>Olivia:1</th>
<th>HRogers:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>TheFakeMT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TheRealMT</td>
<td>HRogers:1</td>
<td></td>
<td></td>
<td>Olivia:1</td>
<td></td>
</tr>
</tbody>
</table>
Revisit the questions

- Read pattern
  - Who all does A follow?
  - **Who all follows A?**
  - Does A follow B?
- Write pattern
  - A follows B
  - A unfollows B
Denormalization

- Second table for reverse relationship
- Otherwise scan across entire table and affect read performance
More optimizations

- Convert into tall-narrow table
- Leverage rowkey indexing better
- Gets -> short Scans

Keeping the column family and column qualifier names short reduces the data transferred over the network back to the client. The KeyValue objects become smaller.

The + in the row key refers to concatenating the two values. You could delimitate using any character you like.

eg: A-B or A,B
Tall-narrow table example

• Denormalization is the way to go

<table>
<thead>
<tr>
<th></th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>TheFakeMT+TheRealMT</td>
<td>Mark Twain:1</td>
</tr>
<tr>
<td>TheFakeMT+MTFanBoy</td>
<td>Amandeep Khurana:1</td>
</tr>
<tr>
<td>TheFakeMT+Olivia</td>
<td>Olivia Clemens:1</td>
</tr>
<tr>
<td>TheFakeMT+HRogers</td>
<td>Henry Rogers:1</td>
</tr>
<tr>
<td>TheRealMT+Olivia</td>
<td>Olivia Clemens:1</td>
</tr>
<tr>
<td>TheRealMT+HRogers</td>
<td>Henry Rogers:1</td>
</tr>
</tbody>
</table>

Putting the user name in the column qualifier saves you from looking up the users table for the name of the user given an id. You can simply list out names or ids while looking at relationships just from this table. The downside of this is that you need to update the name in all the cells if the user updates their name in their profile. This is classic Denormalization.
Uniform rowkey length

- MD5 the userids -> 16 bytes + 16 bytes rowkeys
- Better distribution of load

Using MD5 of the user ids gives you fixed lengths instead of variable length user ids. You don't need concatenation logic anymore.
Uniform rowkey length (continued)

<table>
<thead>
<tr>
<th>MD5(TheFakeMT) MD5(TheRealMT)</th>
<th>TheRealMT:Mark Twain</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5(TheFakeMT) MD5(MTFanBoy)</td>
<td>MTFanBoy:Amandeep Khurana</td>
</tr>
<tr>
<td>MD5(TheFakeMT) MD5(Olivia)</td>
<td>Olivia:Olivia Clemens</td>
</tr>
<tr>
<td>MD5(TheFakeMT) MD5(HRogers)</td>
<td>HRogers:Henry Rogers</td>
</tr>
<tr>
<td>MD5(TheRealMT) MD5(Olivia)</td>
<td>Olivia:Olivia Clemens</td>
</tr>
<tr>
<td>MD5(TheRealMT) MD5(HRogers)</td>
<td>HRogers:Henry Rogers</td>
</tr>
</tbody>
</table>
Tall v/s Wide tables storage footprint

Logical representation of an HBase table. We’ll look at what it means to Get() row r5 from this table.

<table>
<thead>
<tr>
<th></th>
<th>CF1</th>
<th>CF2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>c1:v1</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>c1:v2</td>
<td>c3:v6</td>
</tr>
<tr>
<td>r3</td>
<td>c2:v3</td>
<td>c5:v6</td>
</tr>
<tr>
<td>r4</td>
<td>c2:v4</td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td>c1:v1</td>
<td>c3:v5</td>
</tr>
</tbody>
</table>

Actual physical storage of the table

HFile for CF1
- r1:CF1:c1:t1:v1
- r2:CF1:c1:t2:v2
- r2:CF1:c3:t3:v6
- r3:CF1:c2:t1:v3
- r4:CF1:c2:t1:v4
- r5:CF1:c1:t2:v1
- r5:CF1:c3:t3:v5

HFile for CF2
- r1:CF2:c1:t1:v9
- r1:CF2:c6:t4:v2
- r3:CF2:c5:t4:v6
- r5:CF2:c7:t3:v8

Result object returned for a Get() on row r5
- r5:CF1:c1:t2:v1
- r5:CF1:c3:t3:v5
- r5:CF2:c7:t3:v8

Structure of a KeyValue object

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row Key</td>
<td>Col Fam</td>
</tr>
</tbody>
</table>

KeyValue objects
Rowkey design

• Single most important aspect of designing tables
• Depends on expected access patterns
• HFiles are sorted on Key part of KeyValue objects

HFile for the info column family in the users table

"TheRealMT", "info", "email", 1329088321289, "samuel@clemens.org"
"TheRealMT", "info", "name", 1329088321289, "Mark Twain"
"TheRealMT", "info", "password", 1329088818321, "abc123",
"TheRealMT", "info", "password", 1329088321289, "Langhorne"
Write optimized

- Distribute writes across the cluster
  - Issue most pronounced with time series data

- Hashing
  
  hash("TheRealMT") -> random byte[]

- Salting
  
  ```java
  int salt = new Integer(new Long(timestamp).hashCode()).shortValue() % <number of region servers>;
  byte[] rowkey = Bytes.add(Bytes.toBytes(salt) + Bytes.toBytes("|") + Bytes.toBytes(timestamp));
  ```
• Data to be accessed together should be stored together
  • eg: twit streams - last 10 twits by the users I follow

| Olivia1  | 1Olivia  |
| Olivia2  | 1TheRealMT |
| Olivia5  | 2Olivia  |
| Olivia7  | 2TheFakeMT |
| Olivia9  | 2TheRealMT |
| TheFakeMT2 | 3TheFakeMT |
| TheFakeMT3 | 4TheFakeMT |
| TheFakeMT4 | 5Olivia |
| TheFakeMT5 | 5TheFakeMT |
| TheFakeMT6 | 5TheRealMT |
| TheRealMT1 | 6TheFakeMT |
| TheRealMT2 | 7Olivia |
| TheRealMT5 | 8TheRealMT |
| TheRealMT8 | 9Olivia |
Relational to Non-relational

- Relational concepts
  - Entities
  - Attributes
  - Relationships
- Entities
  - Table is a table. Not much going on there
  - Users table contains... users. Those are entities
    - Good place to start. Denormalization bends that rule a bit
Relational to Non relational (continued)

• Attributes
  • Identifying
    • Primary keys. Compound keys
    • Maps to rowkeys
  • Non-identifying
    • Other columns
    • Maps to column qualifiers and cells
• Relationships
Nested Entities

- Column Qualifiers can contain data instead of just a column name
Schema design summary

- Schema can make or break the performance you get
- Rowkey is the single most important thing
  - Use tricks like hashing and salting
- Denormalize to your advantage
  - There are no joins
- Isolate access patterns
  - Separate CFs or even separate tables
- Shorter names -> lower storage footprint
- Column qualifiers can be used to store data and not just column names
  - Big difference as compared to RDBMS
Deployments and Operations

... for some real stuff, beyond your laptop
Components

- HDFS (DataNodes)
  - Storage
- ZooKeeper
  - Membership management
- RegionServers
  - Serve the regions
- HBase Masters
  - Janitorial work
Hardware

- DataNodes and RegionServers collocated
  - Plenty of RAM and Disk
  - 8-12 cores
  - 48 GB RAM
  - 12 x 1 TB disks
Hardware (continued)

- HBase Master and ZooKeeper collocated
  - Don’t necessarily need to be though
- Keep odd number of ZKs
  - ~3 is good enough for upto 50-70 node cluster typically
- Give ZK independent spindle for log writing
  - I/O latency sensitive
- 4-6 cores
- 24 GB RAM
Types of deployments

• Standalone for dev purpose
• Prototype
  • < 10 nodes
  • No real SLAs
  • 1 ZK + Master, rest are RS
• Small production cluster
  • 10-20 nodes
  • 1 ZK + Master, maybe NNHA, rest are RS
• Medium production cluster
  • 20-50 nodes
  • 3 ZK + Master, NNHA, rest are RS
• Large production cluster
  • 50+ nodes
  • 3 or 5 ZK + Master, NNHA, rest are RS
• Hardware choices remain the same mostly
Deploying software and configuration

- Automated software bits deployment highly recommended
  - Chef
  - Puppet
  - Cloudera Manager
- Centralized configuration management highly recommended
  - Consistency
  - Ease of management
Configurations

- `$HBASE_HOME/conf`
  - `hbase-env.sh`
    - Environment configurations
  - `hbase-site.xml`
    - HBase system configurations
- Linux configurations
  - Number of open files
  - Swappiness
- HDFS configurations
  - `$HADOOP_HOME/hdfs-site.xml`
Monitoring

- Metrics context from Hadoop
  - Ganglia
  - File context
- JMX
  - Cacti
- Several metrics available
  - Write path specific
  - Read path specific
Monitoring (continued)
Performance tuning

- Performance testing
  - Use real (or synthetic) workloads
  - YCSB
- Tuning
  - Write optimized
    - Memstore
  - Random-read optimized
    - Block cache
    - Smaller block size
  - Sequential-read optimized
    - Block cache
    - Higher block size
- GC tuning