UNIVERSITY OF MARYLAND

UMD DATA605 - Big Data Systems

Apache HBase

Instructor: Dr. GP Saggese - gsaggese@umd.edu

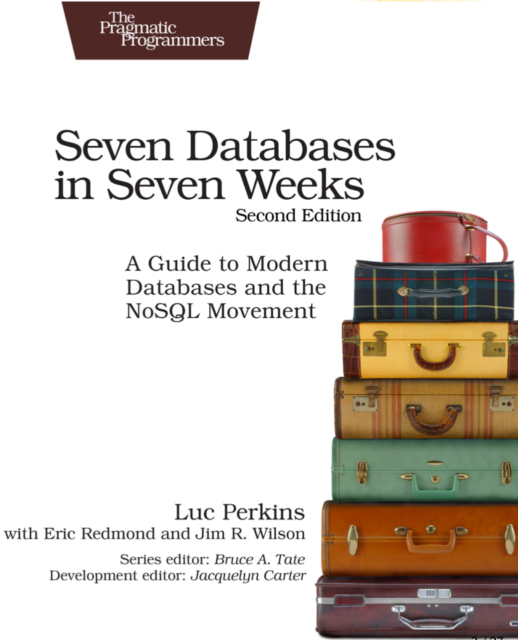
SCIENCE ACADEMY

(Apache) HBase

UMD DATA605 - Big Data Systems

- Issues with Relational DBs
- NoSQL Taxonomy
- **(Apache) HBase**

Resources



The Pragmatic Programmers
Seven Databases in Seven Weeks
Second Edition
A Guide to Modern Databases and the NoSQL Movement
Luc Perkins
with Eric Redmond and Jim R. Wilson
Series editor: Bruce A. Tate
Development editor: Jacquelyn Carter

(Apache) HBase

- HBase = **H**adoop Data**Base**
  - Support very large tables on clusters of commodity hardware
  - Column oriented DB
  - Part of Apache Hadoop ecosystem
  - Use Hadoop filesystem (HDFS)
    - HDFS modeled after Google File System (GFS)
    - HBase based on Google BigTable
    - Google BigTable runs on GFS, HBase runs on HDFS
  - Used at Google, Airbnb, eBay
- **When to use HBase**
  - For large DBs (e.g., at least many 100 GBs or TBs)
  - When having at least 5 nodes in production
- **Applications**
  - Large-scale online analytics
  - Heavy-duty logging
  - Search systems (e.g., Internet search)
  - Facebook Messages (based on Cassandra)
  - Twitter metrics monitoring

HBase: Features

- Data versioning
  - Store versions of data
- Data compression
  - Compress and decompress on-the-fly
  - Makes the system much more complicated
  - Difficult to do random access
- Garbage collection (for expired data)
- In-memory tables
- Atomicity, but only at row level
  - Relational DBs have flexible atomicity **begin ... end transaction**
- Strong consistency guarantees
- Fault tolerant (for machines and network)
  - Write-ahead logging
    - Write data to an in-memory log before it's written to disk
  - Distributed configuration
    - Nodes can rely on each other rather than on a centralized source

From HDFS to HBase

- **Different types of workloads for DB backends**
  - **OLTP** (On-Line Transactional Processing)
    - Read and write individual data items in a large table
    - E.g., update inventory and price as orders come in
  - **OLAP** (On-Line Analytical Processing)
    - Read large amount of data and process it
    - E.g., analyze item purchases over time
- `text Hadoop FileSystem (HDFS) supports OLAP workloads`
  - Provide a filesystem consisting of arbitrarily large files
  - Data should be read sequentially, end-to-end
  - Rarely updated
- `text HBase supports OLTP interactions`
  - Built on top of HDFS
  - Use additional storage and memory to organize the tables
  - Write tables back to HDFS as needed

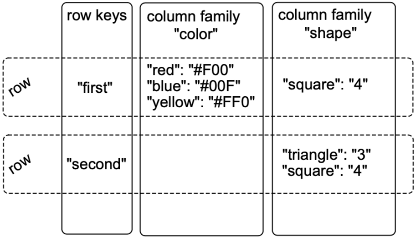
HBase Data Model

- **Warning**: HBase uses names similar to relational DB concepts, but with different meanings
- A **database** consists of multiple tables
- Each **table** consists of multiple rows, sorted by row key
- Each row contains a *row key* and one or more column families
- Each **column family**
  - Can contain multiple columns (family:column)
  - Is defined when the table is created
- A **cell**
  - Is uniquely identified by (table, row, family:column)
  - Contains metadata (e.g., timestamp) and an uninterpreted array of bytes (blob)
- **Versioning**
  - New values don't overwrite the old ones
  - `put()` and `get()` allow to specify a timestamp (otherwise uses current time)

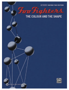
```
\# HBase Database: from table name to Table.
Database = Dict[str, Table]

\# HBase Table.
table: Table = {
  # Row key
  'row1': {
    # (column family:column) → value
    'cf1:col1': 'value1',
    'cf1:col2': 'value2',
    'cf2:col1': 'value3'
  },
  'row2': {
    ... # More row data
  }
}
database = {'table1': table}

\# Querying data.
(value, metadata) = \
table['row1']['cf1:col1']
```

Example 1: Colors and Shape

- Table with:
  - 2 column families
    - "color" and "shape"
  - 2 rows
    - "first" and "second"
- The row "first" has:
  - 3 columns in the column family "color"
    - "red", "blue", "yellow"
  - 1 column in the column family "shape"
    - shape = 4
- The row "second" has:
  - no columns in "color"
  - 2 columns in the column family "shape"
- Data is accessed using a row key and column (family:qualifier)

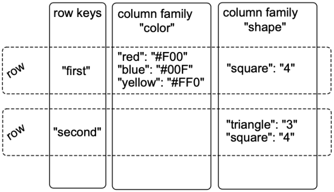



```
table = {
  'first': {
    # (column family, column) → value
    'color': {'red': '#F00',
              'blue': '#00F',
              'yellow': '#FF0'}
    'shape': {'square': '4'}
  },
  'second': {
    'shape': {'triangle': '3',
              'square': '4'}
  }
}
```



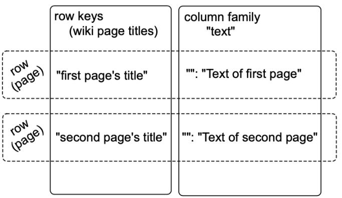
Why all this convoluted stuff?

- **A row in HBase is almost like a mini-database**
  - A cell has many different values associated with it
  - Data is stored in a sparse format
- **Rows in HBase are "deeper" than in relational DBs**
  - In relational DBs rows contain a lot of column values (fixed array with types)
  - In HBase rows contain something like a two-level nested dictionary and metadata (e.g., timestamp)
- **Applications**
  - Store versioned web-site data
  - Store a wiki



Example 2: Storing a Wiki

- **Wiki (e.g., Wikipedia)**
  - Contains pages
  - Each page has a title, an article text varying over time
- **HBase data model**
  - Table name → `wikipedia`
  - Row → entire wiki page
  - Row keys → wiki identifier (e.g., title or URL)
  - Column family → `text`
  - Column → '' (empty)
  - Cell value → article text




```
wikipedia_table = {
  # wiki id.
  'Home': {
    # Column family:column $\to$ value
    ':text': 'Welcome to the wiki!',
  },
  'Welcome page': {
    ... # More row data
  }
}
Database = Dict[str, Table]
database: Database = {'wikipedia':
wiki_table}
(article, metadata) = \
wiki_table['Home']['text']
```

Example 2: Storing a Wiki

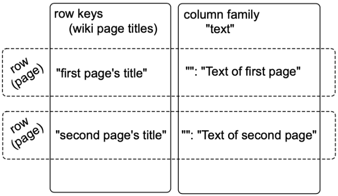
- **Add data**
  - Columns don't need to be predefined when creating a table
  - The column is defined as text

```
> put 'wikipedia', 'Home', 'text',
'Welcome!'
```

  - **Query data**
  - Specify the table name, the row key, and optionally a list of columns

```
> get 'wikipedia', 'Home', 'text'
text: timestamp=1295774833226,
value=Welcome!
```

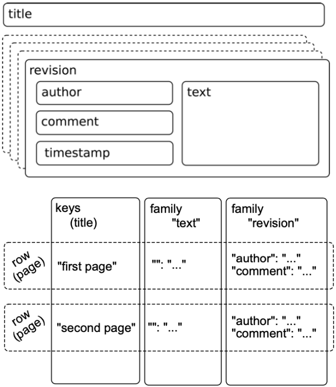
  - HBase returns the timestamp (ms since the epoch 01-01-1970 UTC)




```
wikipedia_table = {
  # wiki id.
  'Home': {
    # Column family, column → value
    'text': 'Welcome to the wiki!',
  },
  'Welcome page': {
    ... # More row data
  }
}
Database = Dict[str, Table]
database: Database = {'wikipedia':
wiki_table}
(queried_value, metadata) = \
wiki_table['Home']['text']
```

Example 2: Improved Wiki

- **Improved wiki using versioning**
- A page
  - Is uniquely identified by its title
  - Can have multiple revisions
- A revision
  - Is made by an author
  - Contains optionally a commit comment
  - Is identified by its timestamp
  - Contains text
- **HBase data model**
- Add a family column "revision" with multiple columns
  - E.g., author, comment, ...
- Timestamp is automatic and binds article text and metadata
- The title is not part of the revision
  - It's fixed and identified uniquely the page (like a primary key)
  - If you want to change the title you need to re-write all the row



Data in Tabular Form

	Name		Home		Office	
Key	First	Last	Phone	Email	Phone	Email
101	Florian	Krepsbach	555-1212	florian@wobegon.org	666-1212	fk@phc.com
102	Marilyn	Tollerud	555-1213		666-1213	
103	Pastor	Inqvist			555-1214	inqvist@wel.org

- Fundamental operations
  - CREATE table, families
  - PUT table, rowid, family:column, value
  - PUT table, rowid, whole-row
  - GET table, rowid
  - SCAN table (*WITH filters*)
  - DROP table

Data in Tabular Form

| Name | | |Home | |Office | | Social |

|———|———|———|———|———|—————|———|————| ——— | | Key | First | | Last | Phone | Email | Phone | Email | FacebookID | |———|———|———|———|———|—————|———|————| ——— | | 101 | Florian | Garfield| Krepsbach | 555-1212 | florian@wobegon.org | 666-1212 | fk@phc.com | - | | 102 | Marilyn | - | Tollerud | 555-1213 | | 666-1213 | | - | | 103 | Pastor | - | Inqvist | | | 555-1214 | inqvist@wel.org | - | :::columns ::::{.column width=20%}

:::: ::::{.column width=20%}

```
New columns can be
added at runtime
```

:::: ::::{.column width=50%}

:::: ::::{.column width=20%}

```
Column families cannot
be added at runtime
```

:::: :::

```
Table People(Name, Home, Office)
{
    101: {
        Timestamp: T403;
        Name: {First="Florian", Middle="Garfield", Last="Krepsbach"},
        Home: {Phone="555-1212", Email="florian@wobegon.org"},
        Office: {Phone="666-1212", Email="fk@phc.com"}
    },
    102: {
        Timestamp: T593;
        Name: {First="Marilyn", Last="Tollerud"},
        Home: {Phone="555-1213"},
        Office: {Phone="666-1213"}
    },
    ...
```

Nested Data Representation

	Name		Home		Office	
Key	First	Last	Phone	Email	Phone	Email
101	Florian	Krepsbach	555-1212	florian@wobegon.org	666-1212	fk@phc.com
102	Marilyn	Tollerud	555-1213		666-1213	
103	Pastor	Inqvist			555-1214	inqvist@wel.org

```
GET People:101
    {
        Timestamp: T403;
        Name: {First="Florian", Last="Krepsbach"},
        Home: {Phone="555-1212", Email="florian@wobegon.org"},
        Office: {Phone="666-1212", Email="fk@phc.com"}
    }

GET People:101:Name
    {First="Florian", Last="Krepsbach"}

GET People:101:Name:First
    "Florian"
```

Column Family vs Column

- **Adding a column**
  - Is cheap
  - Can be done at run-time
- **Adding a column family**
  - Can't be done at run-time
  - Need a copy operation of the table (expensive)
  - This tells you something about how the data is stored
    - Easy to add is a map
    - Hard to add is some sort of static array
    - E.g., MongoDB document vs Relational DB column
- **Why differentiating column families vs columns?**
  - Why not storing all the row data in a single column family?
  - Each column family can be configured independently, e.g.,
    - Compression
    - Performance tuning
    - Stored together in files
  - Everything is designed to accommodate a special kind of data
    - E.g., timestamped web data for search engine

Consistency Model

- **Atomicity**
  - Entire rows are updated atomically or not at all
  - Independently of how many columns are affected
- **Consistency**
  - A GET is guaranteed to return a complete row that existed at some point in the table's history
    - Weak / eventual consistency
    - Check the timestamp to be sure!
  - A SCAN
    - Must include all data written prior to the scan
    - May include updates since it started
- **Isolation**
  - Concurrent vs sequential semantics
  - Not guaranteed outside a single row
  - The atom of information is a row
- **Durability**
  - All successful writes have been made durable on disk

Checking for Row or Column Existence

- HBase supports Bloom filters to check whether a row or column exists
  - It's like a cache for key in keys, instead of keys[key]
  - E.g., instead of querying one can keep track of what's present
- **Hashset complexity**
  - Space needed to store data is unbounded
  - No false positives
  - O(1) in average / amortized (because of reallocations, re-balancing)
- **Bloom filter implementation**
  - Bloom filter is like a probabilistic hash set
  - Array of bits initially all equal to 0
  - When a new blob of data is presented, turning the blob into a hash, and then use hash to set some bits to 1
  - To test if we have seen a blob, compute the hash, check the bits
    - If all bits are 0s, then for sure we didn't see it
    - If all bits are 1s, then it's likely but not sure you have seen that blob (false positive)
- **Bloom filter complexity**
  - Use a constant amount of space
  - Has false positives (no false negatives)
  - O(1)

SCIENCE ACADEMY

Write-Ahead Log (WAL)

- Write-Ahead Log is a general technique used by DBs
  - Provide atomicity and durability
  - Protect against node failures
  - Equivalent to journaling in file system
- HBase and Postgres uses WAL
- **WAL mechanics**
- For performance reasons, the updated state of tables are:
  - Not written to disk immediately
  - Buffered in memory
  - Written to disk as checkpoints periodically
- **Problem**
  - If the server crashes during this limbo period, the state is lost
- **Solution**
  - Use append-only disk-resident data structure
  - Log of operations performed since last table checkpoint are appended to the WAL (it's like storing deltas)
  - When tables are stored to disk, the WAL is cleared
  - If the server crashes during the limbo period, use WAL to recover the state that was not written yet
- When running a big import job, disable the WAL to improve performance
  - Trade off disaster recovery protection for speed

SCIENCE ACADEMY

Storing variable-length data in DBs

SQL Table

```
People(ID: Integer, FirstName: CHAR[20], LastName: CHAR[20], Phone: CHAR[8])
UPDATE People SET Phone="555-3434" WHERE ID=403;
```

ID	FirstName	LastName	Phone
101	Florian	Krepsbach	555-3434
102	Marilyn	Tollerud	555-1213
103	Pastor	Ingvist	555-1214

- Each row is exactly $4 + 20 + 20 + 8 = 52$ bytes long
- To move to the next row: fseek(file,+52)
- To get to Row 401 fseek(file, 401*52);
- Overwrite the data in place

HBase Table

```
People(ID, Name, Home, Office)
PUT People, 403, Home:Phone, 555-3434
```

```
{
  101: {
    Timestamp: T403;
    Name: {First="Florian", Middle="Garfield", Last="Krepsbach"},
    Home: {Phone="555-1212", Email="florian@wobegon.org"},
    Office: {Phone="666-1212", Email="fk@phc.com"}
  },
...
}
```

Need to use pointers

SCIENCE ACADEMY

HBase Implementation

- **How to store the web on disk?**
- **HBase is backed by HDFS**
  - Store each table (e.g., Wikipedia) in one file
  - "One file" means one gigantic file stored in HDFS
    - HDFS splits/replicate file into blocks on different servers
- Here is the idea in several steps:
  - **Idea 1: Put an entire table in one file**
    - Need to overwrite the file every time there is a change in any cell
    - Too slow
  - `text Idea 2: One file + WAL`
    - Better, but doesn't scale to large data
  - `text Idea 3: One file per column family + WAL`
    - Getting better!
  - `text Idea 4: Partition table into regions by key`
    - Region = a chunk of rows [a, b)
    - Regions never overlap

Idea 1: Put the Table in a Single File

- How do we do the following operations?
  - CREATE, DELETE (easy / fast)
  - SCAN (easy / fast)
  - GET, PUT (difficult / slow)

Table People(Name, Home, Office) { 101: { Timestamp: T403; Name: {First="Florian", Middle="Garfield", Last="Krepsbach"}, Home: {Phone="555-1212", Email="florian@wobegon.org"}, Office: {Phone="666-1212", Email="fk@phc.com"} }, 102: { Timestamp: T593; Name: {First="Marilyn", Last="Tollerud"}, Home: {Phone="555-1213"}, Office: {Phone="666-1213"} }, ... }

File "People"

SCIENCE ACADEMY

Idea 2: One file + WAL

Table People(Name, Home, Office)

PUT 101:Office:Phone = "555-3434" PUT 102:Home:Email = mt@yahoo.com

. . . .

WAL for Table People

- Changes are applied only to the log file
- The resulting record is cached in memory
- Reads must consult both memory and disk

Memory Cache for Table People

101

102

GET People:101

GET People:103

PUT People:101:Office:Phone = "555-3434"

SCIENCE ACADEMY

```
101: {
    Timestamp: T403;
```

Idea 2 Requires Periodic Table Update

101: {Timestamp: T403;Name: {First="Florian", Middle="Garfield", Last="Krepsbach"},Home: {Phone="555-1212", Email="florian@wobegon.org"},Office: {Phone="666-1212", Email="fk@phc.com"}}, 102: {Timestamp: T593;Name: { First="Marilyn", Last="Tollerud"},Home: { Phone="555-1213" },Office: { Phone="666-1213" }}, . . .

Table for People on Disk (Old)

PUT 101:Office:Phone = "555-3434" PUT 102:Home:Email = mt@yahoo.com . . .

WAL for Table People:

101: {Timestamp: T403;Name: {First="Florian", Middle="Garfield", Last="Krepsbach"},Home: {Phone="555-1212", Email="florian@wobegon.org"},Office: {Phone="**555-3434**", Email="fk@phc.com"}},102: {Timestamp: T593;Name: { First="Marilyn", Last="Tollerud"},Home: { Phone="555-1213", **Email="my@yahoo.com"** }, . . .

Table for People on Disk (New)

Idea 3: Partition by Column Family

Data for Column Family Name

Tables for People on Disk (Old)

PUT 101:Office:Phone = "555-3434" PUT 102:Home:Email = mt@yahoo.com

. . .

WAL for Table People

Tables for People on Disk (New)

- Write out a new copy of the table, with all of the changes applied
- Delete the log and memory cache
- Start over

Data for Column Family Home

Data for Column Family Office

Data for Column Family Home (Changed)

Data for Column Family Office (Changed)

Data for Column Family Name

Idea 4: Split Into Regions

Region 1: Keys 100-200

Region 2: Keys 100-200

Region 3: Keys 100-200

Region 4: Keys 100-200

Region Server

Region Master

Region Server

Region Server

Region Server

Transaction Log

Memory Cache

Table

Table

Final HBase Data Layout