Acknowledgements

- Credits
  - Part of the course material is based on slides provided by the following authors
    - Mohamed Eltabakh, Susan B. Davidson, Kevin Matulef
Introduction

- HDFS focuses on write once, read many workloads
- What if we need to store data and, in addition to full scans for analytics, occasionally update and read random elements?

- Example: Webtable
  - Output of the web crawler: for each web page URL, store the page, the attributes (language, MIME types), ...
  - The whole table is accessed by MapReduce jobs for analytics
  - Random rows are updated by the crawler
  - Random rows are read by the crawler or by other systems

- Solution: HBase

HBase: Overview

- HBase is a distributed column-oriented data store built on top of HDFS
- HBase is an Apache open source project whose goal is to provide storage for the Hadoop Distributed Computing
- Data is logically organized into tables, rows and columns
HBase: Part of Hadoop’s Ecosystem

The Hadoop Ecosystem

HBase is built on top of HDFS

HBase files are internally stored in HDFS

HBase vs. HDFS

- Both are distributed systems that scale to hundreds or thousands of nodes

- HDFS is good for batch processing (scans over big files)
  - Not good for record lookup
  - Not good for incremental addition of small batches
  - Not good for updates

- HBase is designed to efficiently address the above points
  - Fast record lookup
  - Support for record-level insertion
  - Support for updates (not in place)

- HBase updates are done by creating new versions of values
### HBase vs. HDFS (Cont’d)

<table>
<thead>
<tr>
<th></th>
<th>Plain HDFS/MR</th>
<th>HBase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write pattern</td>
<td>Append-only</td>
<td>Random write, bulk incremental</td>
</tr>
<tr>
<td>Read pattern</td>
<td>Full table scan, partition table scan</td>
<td>Random read, small range scan, or table scan</td>
</tr>
<tr>
<td>Hive (SQL)</td>
<td>Very good</td>
<td>4-5x slower</td>
</tr>
<tr>
<td>performance</td>
<td>Do-it-yourself / TSV / SequenceFile / Avro / ?</td>
<td>Sparse column-family data model</td>
</tr>
<tr>
<td>Structured storage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max data size</td>
<td>30+ PB</td>
<td>~1PB</td>
</tr>
</tbody>
</table>

If application has neither random reads or writes ➔ Stick to HDFS

### HBase Data Model

- HBase is based on Google’s Bigtable model
  - Key-Value pairs
  - Rows are ordered by keys

- Table schema only define it's column families
  - Each family consists of any number of columns
  - Each column consists of any number of versions
  - Columns only exist when inserted,
  - Everything except table names are byte[]
  - (Row, Family: Column, Timestamp) ➔ Value

![HBase Data Model Diagram](image-url)
HBase: Keys and Column Families

Each row has a **Key**

Each record is divided into **Column Families**

Each column family consists of one or more **Columns**

A single cell may have different values with different timestamps (not shown here)

### Notes on Data Model

- **HBase schema consists of several Tables**
- **Each table consists of a set of Column Families**
  - Once the Column Families are defined, Columns can be added at any time
    - HBase has Dynamic Columns
  - Hardcoded name convention
    - column_family_name:column_ID
- **Not all the columns are used by all the keys**
  - Table can be very sparse, many cells are empty
HBase Physical Model

- Each column family is stored in a separate file (called HTables)
- Key & Version numbers are replicated with each column family
- Empty cells are not stored

### Table 5.3. ColumnFamily contents

<table>
<thead>
<tr>
<th>Row Key</th>
<th>Time Stamp</th>
<th>ColumnFamily &quot;contents:&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>t6</td>
<td>contents: html = &quot;&lt;html&gt;...&quot;</td>
</tr>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>t5</td>
<td>contents: html = &quot;&lt;html&gt;...&quot;</td>
</tr>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>t3</td>
<td>contents: html = &quot;&lt;html&gt;...&quot;</td>
</tr>
</tbody>
</table>

### Table 5.2. ColumnFamily anchor

<table>
<thead>
<tr>
<th>Row Key</th>
<th>Time Stamp</th>
<th>Column Family anchor</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>9</td>
<td>anchor: cnni.com = &quot;CNN&quot;</td>
</tr>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>8</td>
<td>anchor: my.look.ca = &quot;CNN.com&quot;</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>Row key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutting</td>
<td>info: { 'height': '9ft', 'state': 'CA' }</td>
</tr>
<tr>
<td></td>
<td>roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }</td>
</tr>
<tr>
<td>tipcon</td>
<td>info: { 'height': '5ft7', 'state': 'CA' }</td>
</tr>
<tr>
<td></td>
<td>roles: { 'Hadoop': 'Committer @ts=2010', 'Hadoop': 'PMC @ts=2011', 'Hive': 'Contributor' }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column Family</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row key</td>
<td>Column key</td>
</tr>
<tr>
<td>cutting</td>
<td>info:height</td>
</tr>
<tr>
<td>cutting</td>
<td>info:state</td>
</tr>
<tr>
<td>tipcon</td>
<td>info:height</td>
</tr>
<tr>
<td>tipcon</td>
<td>info:state</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column Family</th>
<th>Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row key</td>
<td>Column key</td>
</tr>
<tr>
<td>cutting</td>
<td>roles:ASF</td>
</tr>
<tr>
<td>cutting</td>
<td>roles:Hadoop</td>
</tr>
<tr>
<td>tipcon</td>
<td>roles:Hadoop</td>
</tr>
<tr>
<td>tipcon</td>
<td>roles:Hadoop</td>
</tr>
<tr>
<td>tipcon</td>
<td>roles:Hive</td>
</tr>
</tbody>
</table>
Column Families

- Different sets of columns may have different properties and access patterns

- Configurable by column family:
  - Compression (none, gzip, ...)
  - Version retention policies

- Column Families stored separately on disk
  - Access one without wasting I/O on the other

HBase Regions

- Each HTable (column family) is partitioned horizontally into regions
  - Regions are counterpart to HDFS blocks

Table 5.3. ColumnFamily contents

<table>
<thead>
<tr>
<th>Row Key</th>
<th>Time Stamp</th>
<th>ColumnFamily &quot;contents:&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>16</td>
<td>contents:html = &quot;&lt;html&gt;...&quot;</td>
</tr>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>15</td>
<td>contents:html = &quot;&lt;html&gt;...&quot;</td>
</tr>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>13</td>
<td>contents:html = &quot;&lt;html&gt;...&quot;</td>
</tr>
</tbody>
</table>

Each will be one region
HBase Components

- **Region**
  - A subset of a table’s rows, like horizontal range partitioning
  - Automatically done

- **RegionServer (many slaves)**
  - Manages data regions
  - Serves data for reads and writes (using a log)

- **Master**
  - Responsible for coordinating the slaves
  - Assigns regions, detects failures
  - Admin functions

Big Picture
Access to data

- Set of APIs that can be used to do basic operations

- Put()
  - Insert a new record (with a new key), or insert a record for an existing key
  - The column ID (given an existing column family) must be also specified
  - Implicit or explicit versioning

- Get()
  - retrieve the value given a key and a column ID

- Scan()
  - retrieve all the values given a column identifier (and a range of keys)

- Delete()
  - Multiple levels
    - Can mark an entire column family as deleted
    - Can make all column families of a given row as deleted

---

Access to data: Get()

Select value from table where key='com.apache.www' AND label='anchor:apache.com'

<table>
<thead>
<tr>
<th>Row key</th>
<th>Time Stamp</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>“com.apache.www”</td>
<td>t12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t10</td>
<td>“anchor:apache.com”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“APACHE”</td>
</tr>
<tr>
<td>“com.cnn.www”</td>
<td>t9</td>
<td>“anchor:cnnsi.com”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“CNN”</td>
</tr>
<tr>
<td></td>
<td>t8</td>
<td>“anchor:my.look.ca”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“CNN.com”</td>
</tr>
<tr>
<td></td>
<td>t6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t3</td>
<td></td>
</tr>
</tbody>
</table>
### Access to data: Scan()

Select value from table where anchor = ‘cnnsi.com’

<table>
<thead>
<tr>
<th>Row key</th>
<th>Time Stamp</th>
<th>Column “anchor:”</th>
</tr>
</thead>
<tbody>
<tr>
<td>“com.apache.www”</td>
<td>t12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t10</td>
<td>“anchor:apache.com”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“APACHE”</td>
</tr>
<tr>
<td>“com.cnnsi.www”</td>
<td>t9</td>
<td>“anchor:cnnsi.com”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“CNN”</td>
</tr>
<tr>
<td></td>
<td>t8</td>
<td>“anchor:my.look.ca”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“CNN.com”</td>
</tr>
<tr>
<td></td>
<td>t6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>t3</td>
<td></td>
</tr>
</tbody>
</table>

### HBase Deployment

**Master node**
- NameNode
- SecondaryNameNode
- HMaster
- JobTracker
- ZooKeeper

**Slave nodes**
- RegionServer
- DataNode
- TaskTracker

The proverbial basket full of eggs

5+ slaves with HBase, HDFS, and MR slave processes
## 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/Authorisation</td>
<td>Work in progress</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>
Overview

- Document oriented, not table/row oriented
- Collection of binary JSON (BSON) documents
- Schemaless
- No relations or transactions native in database
- Scalable and high-performance
- Full index support
- Written in C++
- Servers for all major platforms
- Drivers for all major development environments
- Free and open-source, but also commercial support

BSON

- Binary JSON
- Binary encoded serialization of JSON-like documents
- Like JSON, BSON supports the embedding of documents and arrays within other documents and arrays. BSON also contains extensions that allow representation of data types that are not part of the JSON spec. For example, BSON has a Date type and a BinData type.
- The driver performs translation from the language’s “domain object” data representation to BSON, and back
MondoDB Data Model and Types

- Data Model
  - A MongoDB deployment hosts a number of databases
  - A database holds a set of collections
  - A collection holds a set of documents
  - A document is a set of key-value pairs

- Data types
  - Basic: Null, Boolean, Integer (32- and 64-bit), Floating point, String
  - More complex: Date, Code (JavaScript), Array, Embedded document

Sample Document

```json
{
  _id: ObjectId("4c4ba5c0672c685e5e8aabf3"),
  author: "Kevin",
  date: new Date("02/28/2012"),
  text: "About MongoDB…",
  tags: [ "tech", "databases" ]
  comments: [{author: "jim", comment: "I disagree" },
              {author: "nancy", comment: "Good post"}]
}
```

Array of documents

Always indexed. Can be any BSON data type other than an array
Core MongoDB Operations

- CRUD: create, read, update, and delete
- Insert
  - One at a time: `db.mycollection.insert(mydoc)`
  - Batch insert
- Querying
  - Use `find()`/`findOne()` functions and a query document
  - Ranges, set inclusion, inequalities using $ conditionals
- Delete
  - Documents that match some predicate, e.g. to remove the document just added
    `db.mycollection.remove({"_id": 1})`
  - All documents in a collection
    `db.mycollection.remove()`

Find

- Get the entire collection (called posts)
  `db.posts.find()`

- Get the documents satisfying some constraints
  `db.posts.find({"author": "Kevin"})`

- Specifying Which Keys to Return
  `db.mydoc.find({}, {"author", "tags"})`

```javascript
{
    _id: ObjectId("4c4ba5c0672c685e5e8aabf3"),
    author: "Kevin",
    tags: [ "tech", "databases" ]
}
Ranges, Negation, OR-clauses

- Comparison operators: $lt, $lte, $gt, $gte

  ```
  start = new Date(01/01/2012)
  end = new Date(04/01/2012)
  db.posts.find({"date": {"$gte": start, "$lte": end}})
  ```

- Negation: $ne

  ```
  db.posts.find({"date": {"$ne": end}})
  ```

- Or queries: $in (single key), $or (different keys)

  ```
  db.posts.find({"date": {"$in": [start, end]}})
  db.posts.find({"$or": [{"date": start}, {"name": "John"}]})
  ```

Limits, Skips, Sort, Count

- Limits the number of results to 3

  ```
  db.posts.find().limit(3)
  ```

- Skips the first three results and returns the rest

  ```
  db.posts.find().skip(3)
  ```

- Sorts by author ascending (1) and title descending (-1)

  ```
  db.posts.find().sort({"author":1, "title": -1})
  ```

- Counts the number of documents in the people collection matching the find(...) query

  ```
  db.people.find(...).count()
  ```
Summary of MongoDB

- MongoDB is an example of a document-oriented NoSQL solution
- The query language is limited, and oriented around “collection” (relation) at a time processing
  - Joins are done via a query language
- The power of the solution lies in the distributed, parallel nature of query processing
  - Replication and sharding