Column-oriented databases and HBase

Steve Phelps

2017

Reading

- George [2011] is available as a physical book, as well as an online reference.
- See `https://hbase.apache.org/book.html`.
- For this module, the most important chapter is the Data Model.
- See also [Redmond and Wilson, 2012, ch. 4].

Apache HBase

- HBase is a column-oriented database.
- It is inspired by Google's Bigtable Chang et al. [2006].
- HBase is written in Java (Bigtable was written in C++).
- HBase can be used from Python through a Thrift API.
- It can also be access through the Apache Spark Python API.

Scalability

- The design philosophy is based on highly-distributed, shared-nothing architecture.
- It provides Automatic sharding.
- HBase is part of the Hadoop Ecosystem.

Shared-nothing architecture

- When building scalable systems, we need to parallelize our computations.
- This requires multiple processors (and/or multiple cores).
- How do we distribute the data to each processor?

Scheme	Description
Shared memory (SM)	Multiple processors share a common central memory.
Shared disk (SD)	Multiple processors with private memory share a common collection of disks.
Shared nothing (SN)	Neither memory not peripheral storage is shared among processors.

Stonebraker [1986]

Sharding

- We can provide parallel access to a database by storing different subsets of the data on different RDBMS servers.
- We could use, e.g. a hash function to determine which server a piece of data should reside on.
- Each of these subsets is called a shard.
- With a traditional relational database, sharding is very costly in terms of duplicated resources and the complexity of the configuration.
- HBase provides automatic sharding, with minimal duplication of state.

Regions

- Tables are partitioned vertically into different *regions*.
- Different *region servers* are responsible for one or more regions.
- The HBase Master process performs coordination and load balancing across the region servers.

The Hadoop Distributed File System (HDFS)

- In a production configuration HBase stores its underlying data on a file-system called HDFS.
- HDFS uses a cluster of computers to simulate a single file-system.
- It provides:
  - Resilience against hardware
  - Streaming Data Access
  - Support for large files (e.g. terabytes)
- It is based on the philosophy that moving computation is often cheaper than moving data.

HBase Tables

- Tables are maps of maps.
- We first map from row key to a map of the data for that row.
- Then within each row, column names are mapped to values.
- Tables consist of:
  - *Rows*, which hold the data associated with the row-key.
  - *Column keys*, which are used to index each row.
  - *Row keys*, which are used to index each attribute within a row.
  - *Column families* which group related keys to specify access-control and options.

Multi-dimensional sorted maps

- Row keys are sorted in lexicographic order.
- The number of columns per row is *unbounded*.
- This design implements a persistent, sparse, multi-dimensional sorted map.

Sparse data

- When de-normalising a schema, we introduce many `NULL` values.
- A data-set which contains a significant fraction of `NULL` values is called `sparse` data.
- `NULL` values can be represented by the absence of a mapping.
- Existence is tested using a Bloom filter.
- The bloom filter is cached in memory;
  - This prevents unnecessary disk accesses.

HBase is typeless

- There are *no types*.
- Both keys and their associated values are arbitrary-length arrays of bytes.
- There is no limit on the size of a value.

Lexicographic ordering

- We can define an ordering over arbitrary binary data.
- For example, the following Python code determines whether $x < y$ according to lexicographic ordering:

```
def lexicographic_le(x, y):
    for i in range(len(x)):
        if x[i] == y[i]:
            continue
        else:
            return x[i] < y[i]
    return False

lexicographic_le('steve', 'smith')
Out[77]: False
```

Ordering arrays of bytes

```
x = np.array([15, 9, 5, 16, 4])

y = np.array([15, 16, 1, 18, 1])

lexicographic_le(x, y)
Out[98]: True
```

Joins in HBase

- There is no automatic enforcement of referentially integrity in a column-oriented database.
  - There are no foreign key constraints.
- If we want to retrieve the value associated with a particular key from another table, then that will be a fast operation.
- In general, however, the set-theoretic operations available in SQL can be very expensive.
- Joins are not supported; we "pre-join" the data through de-normalisation.

An example table

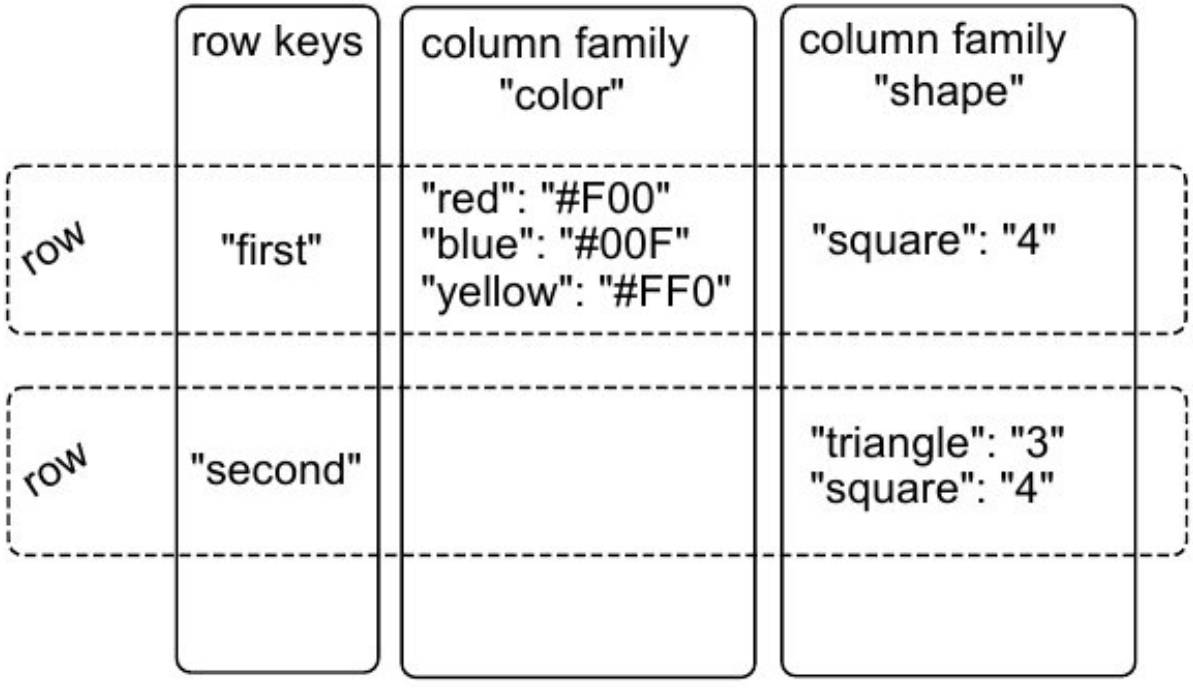

Figure 13—HBase tables consist of rows, keys, column families, columns, and values.

[Redmond and Wilson, 2012, p. 98]

Column qualifiers

- Row keys and column keys can consist of arbitrary binary data.
- Often we will use strings (which HBase will see an array of 8-bit ASCII values).
- Column-keys containing the character ":" specify membership of a column-family:

```
<column family name>:<column qualifer>
```

- The column family name must consist of printable text.
- For example, in the previous example, the `triangle` attribute would be written as:

```
shape:triangle
```

- The fully-qualified key is stored on disk.
- Therefore, the name of a column-family should generally be short, in order to maintain capacity and IO throughput.

The colors and shapes example with Python dicts

```
data = \
    {
        'first':
            {
                'color:red': 0xf00, 'color:blue': 0x00f,
                    'shape:yellow': 0xff0, 'shape:square': 4
            },
        'second':
            {
                'shape:square': 4, 'shape:triangle': 3
            }
    }

data['second']
Out[11]: {'shape:square': 4, 'shape:triangle': 3}

data['second']['shape:triangle']
Out[12]: 3
```

URLs

- BigTable was originally designed to house web data (it originated from Google).
- It is very common to use a URL as a row-key.
- We would typically reverse the URL before using it as a key.
- For example:

`keats.kcl.ac.uk` becomes `uk.ac.kcl.keats`.

Time stamps

- HBase also allows multiple versions of the values associated with a given column for a given row.
- We can optionally use this to index by the time-stamp of the datum.
- For example, we might store several versions of a web page associated with a given URL.
- The time-stamp dimension is optionally.
- When writing data, if we do not specify the time-stamp, then the current system time is used.
- When modifying data, the old value is kept along with the old time-stamp.
- There are therefore three dimensions to the multi-dimensional map.
- For some applications, we can use the time-dimension to store our own data.

Epoch time

- Time-stamps are stored as integers in Unix epoch time.
- Epoch time is number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970
- To get the current epoch time on a Unix system:

```
date +%s
```

Table operations

- There are four core CRUD operations which we can perform on tables:
  1. Get
  2. Put
  3. Delete
  4. Scan
- We execute them by either:
  - Executing a command in the shell.
  - Calling a Java method in the Java API.
  - Calling a Python function in the Thrift API.
- On a big project, it is typically more practical to use the API than the shell.

Get

- Get returns attributes for a specified row-key and column(s).
- It can be used in several ways:
  - Get everything for a row.
  - To get all columns from specific column families.
  - To get specific columns.
  - To get values within a specified time range.
- By default it will retrieve up to three versions of the data
  - We can specify a different number each time we get.

Put

- Put can be used to insert new rows, or update existing rows.
- We always specify a row key.
- Where there is no existing value at a cell, new data is created.
- If there is existing data, it is retained under the old time stamp.

Delete

- Delete a particular cell.

Scan

- Scan is used to retrieve a *range* of rows in one operation.
- In its simplest form it will iterate over all rows in the table.
- We can also specify a minimum and maximum row-key.
- The ordering is given by the lexicographic ordering of the row-key itself.
  - Ascending and descending are supported.
  - Note that we cannot specify an arbitrary ordering of the data;
  - there is no equivalent of SQL's `ORDER BY` clause.

Tall-Narrow Versus Flat-Wide tables

- In a column-oriented database we need to think carefully about the structure of the table in respect of the kinds of queries we want to support.
- We can choose a tall-narrow design as opposed to a flat-wide design.
  - Tall-narrow: very many rows, fewer columns
  - Flat-wide: fewer rows, very many columns
- A tall-narrow design is more readily shard-able.
  - Individual rows cannot be split across regions.
- A flat-wide design can be transformed into a tall-narrow design by storing additional data values in the row-key.
- These attributes can be retrieved by scanning with a *partial-key*.

Partial Key Scans

- Because keys are sorted lexicographically, we can index several attributes simultaneously through concatenation.
- e.g. suppose we use row'keys as with the following format:

`<userId>-<date>-<messageId>-<attachmentId>`

- We can then *scan* the table by specifying a partial-key:

Table 9-1. Possible start keys and their meaning

Command	Description
`<userId>`	Scan over all messages for a given user ID.
`<userId>-<date>`	Scan over all messages on a given date for the given user ID.
`<userId>-<date>-<messageId>`	Scan over all parts of a message for a given user ID and date.
`<userId>-<date>-<messageId>-<attachmentId>`	Scan over all attachments of a message for a given user ID and date.

George [2011]

Underlying data-structures.

- The ordering over keys is provided by Log-Structured Merge Trees O'Neil et al. [1996].
- In contrast to B+ Trees in traditional databases.
- LSMT trees ensure that rows of tables can efficiently be read from disk sequentially.
- The merging operation is scheduled in the background automatically.
- No need to constantly optimise tables to avoid fragmentation of pages.
- *Scanning* a table over a large range is bound only by sequential IO latencies.

CRUD operations from the shell

- We can perform create, read, update and delete (CRUD) operations from the HBase shell.
- To start the HBase shell type a command similar to the following from the Unix shell:

```
hbase shell
```

- To check the status of HBase use the command `status`.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. 2006. URL `http://research.google.com/archive/bigtable.html`.

Lars George. *HBase: The Definitive Guide*. O'Reilly Media, 1st edition, 2011. ISBN 978-1449396107.

Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge Tree. *Acta Informatica*, 33(4): 351–385, 1996. doi: 10.1007/s002360050048. URL `http://link.springer.com/article/10.1007/s002360050048`.

Eric Redmond and Jim R. Wilson. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. The Pragmatic Bookshelf, Dallas, Texas, p1.0 edition, 2012. ISBN 9781934356920.

Michael Stonebraker. The Case for Shared Nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986. ISSN 09547762. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.5370{&}rep=rep1{&}type=pdf.