Column family stores

Introduction to NoSQL: Lecture 4

Piotr Fulmański

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
University of Lodz

NoSQL Theory and examples

by Piotr Fulmański

Piotr Fulmański, 2021

PIOTR FULMAŃSKI

NoSQL
Theory and examples

SIMPLE INTRODUCTION SERIES

General overview of column family stores

- Basic ideas and features
- Working with HBase

The origins

From Big Data through Big Table to Hadoop

The story began when PageRank was developed in Google. Predicted large amount of data to be processed, PageRank required the use of non-standard solutions.

- Google decided to **extensively use massively parallelizing and distributing processing** across very large numbers of low budget servers.
- Instead of dedicated storage servers built and operated by other companies, they use storage based on disks directly attached to the same servers which perform data processing.
- Redefine unit of computation. No more individual servers. Instead the Google Modular Data Center were used. The Modular Data Center comprises shipping containers that house about a thousand custom-designed Intel based servers running Linux. Each module includes an independent power supply and air conditioning. Thus, data center capacity is increased not by adding new servers individually but by adding new modules with 1,000 servers each.

The origins

From Big Data through Big Table to Hadoop

There were three major software layers to serve as the foundation for the Google platform.

- GoogleFileSystem (GFS) is a distributed cluster filesystem.
- MapReduce is a distributed processing framework.
- BigTable is a database system.

The origins

From Big Data through Big Table to Hadoop

In this architecture data which are going to be computed are divided into smaller but still large blocks and distributed across nodes. Then packaged code is transferred into nodes to process the data in parallel. **This approach takes advantage of data locality, where nodes manipulate the data they have access to.** This allows the dataset to be processed faster and more efficiently than it would be in a more conventional supercomputer architecture that relies on a parallel file system where computation and data are distributed via high-speed networking.

The origins

From Big Data through Big Table to Hadoop

Based on Google's ideas, and having positively verified working examples (Google itself), an open source Apache project being an equivalent of Google stack, has been started. This project is known under the *Hadoop* name.

The origins

From Big Data through Big Table to Hadoop

We can say that no any other project before has had as great influence on data processing as Hadoop. The main reasons for this were (and still are):

• Accessibility. Hadoop was free–every one can download, install and work with it. Everyone can do this because we don't need superserver to host this software stack – we can install it even on a low budget laptop.

• One week of self training is enough to start using this platform thanks to good level of abstraction:

- GFS abstracts the storage contained in the nodes.
- MapReduce abstracts the processing power contained within these nodes.
- BigTable allows easy storage of almost everything.

• Perfect scalability. We can start very small company almost without any costs. Over time, as we become more and more recognizable, we can offer "more" without needs to architectural changes – we simply add another nodes to get more power. This allows many start-ups to offer good solutions, based on a proven architectural conception.

Hadoop

The base Apache Hadoop framework is composed of the following modules:

• Hadoop Common contains libraries and utilities needed by other Hadoop modules.

• Hadoop Distributed File System (HDFS) is a distributed file system that stores data on commodity machines.

• Hadoop YARN is a platform responsible for managing computing resources in clusters and using them for scheduling users' applications.

• Hadoop MapReduce is an implementation of the MapReduce programming model for large-scale data processing.

Hadoop

All the modules in Hadoop are designed with a fundamental assumption that

hardware failures are common situation and should be automatically handled by the framework.

HBase

Although HBase is a schema free database it does enforce some kind of structure on the data. This structure is very basic and is based on well known terms: *columns*, *rows*, *tables* and *keys*.

However, HBase tables vary significantly from the relational tables with which we are familiar. To get a sense how the HBase data model works, we have to familiarize ourself with two concepts: ***aggregation oriented model*** and ***sparse data***.

HBase

Aggregation oriented model

HBase

Sparse data

HBase

Column family

A general idea behind column families data model is to **support frequent access to data used together**.

Column families for a single row may or may not be near each other when stored on disk, but **columns within a one column family are kept together**.

This is a fundamental element supporting (or supported) by aggregations.

And this is also one of the fundamental assumption made by Google stack where **nodes manipulate the data they have access to**, taking a full advantage of data locality.

Remember:

BigTable was created to support that model of computation.

HBase

Column family

HBase is a schema free database, so we can store

HBase

Column family

HBase is a schema free database, so we can store

- what we want,

HBase

Column family

HBase is a schema free database, so we can store

- what we want,
- when we want (we don't have to specify values for all columns as well as we don't have to specify types of our data)

HBase

Column family

HBase is a schema free database, so we can store

- what we want,
- when we want (we don't have to specify values for all columns as well as we don't have to specify types of our data)
- and how we want (no need to create tables and columns ahead)

HBase

Column family

HBase is a schema free database, so we can store

- what we want,
- when we want (we don't have to specify values for all columns as well as we don't have to specify types of our data)
- and how we want (no need to create tables and columns ahead)

but one thing should be decided before we start our database: column families.

HBase

Column family

The BigTable database, in terms of structure, is completely different from relational databases.

It is a ***fixed (persistent) multidimensional ordered map***.

Column family stores use row and column identifiers as general purposes keys for data lookup in this map.

More precisely, map is indexed by four keys/names:

- *row keys*,
- *column families*,
- *column keys*,
- and *time stamps* (the latter defines different versions of the same data).

All **values** in this map appear as unmanaged character tables (their interpretation, such as data type, is an application task).

HBase

Column family

```
{
  "character0001": {          ⟵ row key
    "info": {                 ⟵ column family
      "livingPlace": {        ⟵ column name
        t2: "Star Destroyer"  ⟵ version
        t1: "Tatooine"
      },
      "name": {
        t2: "Darth Vader"
        t1: "Anakin Skywalker"
      }
    },

    "history": {
      ...
    }
  },

  "character0002": {
    "info": {
      "livingPlace" : {
        t1: "unknown"
      },
      "name": {
        t1: "Boba Fett"
      }
    },
    "history": {
      ...
    }
  }
}
```

HBase

Rows

The basic unit of storing information is the row identified by a key.

The row key is a string of arbitrary characters from a typical length of 10 to 100 bytes (the maximum length is 64KiB).

Saving and reading rows is an atomic operation regardless of the number of columns we read or write to.

BigTable stores rows structured lexicographically according to theirs keys.

Rows in the table are dynamically partitioned (divided into smaller subsets). Each of these subset (row ranges), called a *tablet*, determines a distribution unit in a cluster. Wise use of the tablet(s) helps to balance the database load. Readings from small row ranges are more efficient, because they typically require communication between fewer machines, in the best case we use data from one machine.

HBase

Columns

The rows consists of columns.

Each row may have a different number of columns storing different data types (including compound types).

The column keys are grouped into sets called column families. They are the basic unit when accessing data.

Data stored in a given family is usually of the same type.

Column families must be created before starting data placement. Once created, you cannot use a column key that does not belong to any family.

The number of different column families in the table should not exceed 100 and should be rarely changed at work (their change requires a database stop and restart).

The number of columns is not limited. They can be added and removed while the database is running.

HBase

Versions

There may be different versions of the same cell in the BigTable database identified by **timestamp**.

Different versions of cells are stored in the descending order of the date.

There are two options to not store too many versions and automatically remove older data:

- keep the last *n* versions,
- or for example versions from some time period (e.g. the last week).

HBase

Ways of data access

We may use one set of four keys (row key, column family name, column name, time stamp) to create compound key and have this way an access to one specific data version.

Key: `["character0001", "info", "name", t1]`

Result:

HBase

Ways of data access

We may use one set of four keys (row key, column family name, column name, time stamp) to create compound key and have this way an access to one specific data version.

Key: `["character0001", "info", "name", t1]`

Result:

```
{
  "character0001": {          ⟵ row key
    "info": {                 ⟵ column family
      "livingPlace": {        ⟵ column name
        t2: "Star Destroyer"  ⟵ version
        t1: "Tatooine"
      },
      "name": {
        t2: "Darth Vader"
        t1: "Anakin Skywalker"
      }
    },

    "history": {
      ...
    }
  },

  "character0002": {
    "info": {
      "livingPlace" : {
        t1: "unknown"
      },
```

HBase

Ways of data access

We may use one set of four keys (row key, column family name, column name, time stamp) to create compound key and have this way an access to one specific data version.

Key: `["character0001", "info", "name", t1]`

Result:

```
"Anakin Skywalker"
```

```
{
  "character0001": {          ⟵ row key
    "info": {                 ⟵ column family
      "livingPlace": {        ⟵ column name
        t2: "Star Destroyer"  ⟵ version
        t1: "Tatooine"
      },
      "name": {
        t2: "Darth Vader"
        t1: "Anakin Skywalker"
      }
    },

    "history": {
      ...
    }
  },

  "character0002": {
    "info": {
      "livingPlace" : {
        t1: "unknown"
      },
```

HBase

Ways of data access

Using a set of three keys (row key, column family name, column name) we get all versions of a given cell.

Key: `["character0001", "info", "name"]`

Result:

HBase

Ways of data access

Using a set of three keys (row key, column family name, column name) we get all versions of a given cell.

Key: `["character0001", "info", "name"]`

Result:

```
{
  "character0001": {          ⟵ row key
    "info": {                 ⟵ column family
      "livingPlace": {        ⟵ column name
        t2: "Star Destroyer"  ⟵ version
        t1: "Tatooine"
      },
      "name": {
        t2: "Darth Vader"
        t1: "Anakin Skywalker"
      }
    },

    "history": {
      ...
    }
  },

  "character0002": {
    "info": {
      "livingPlace" : {
        t1: "unknown"
      },
```

HBase

Ways of data access

Using a set of three keys (row key, column family name, column name) we get all versions of a given cell.

Key: `["character0001", "info", "name"]`

Result:

```
{
  t2: "Darth Vader",
  t1: "Anakin Skywalker"
}
```

```
{
  "character0001": {          ⟵ row key
    "info": {                 ⟵ column family
      "livingPlace": {        ⟵ column name
        t2: "Star Destroyer"  ⟵ version
        t1: "Tatooine"
      },
      "name": {
        t2: "Darth Vader"
        t1: "Anakin Skywalker"
      }
    },

    "history": {
      ...
    }
  },

  "character0002": {
    "info": {
      "livingPlace" : {
        t1: "unknown"
      },
```

HBase

Ways of data access

Using a set of two keys (row key, column family name), we specify all cells in a given column family.

Key: `["character0001", "info"]`

Result:

HBase

Ways of data access

Using a set of two keys (row key, column family name), we specify all cells in a given column family.

Key: `["character0001", "info"]`

Result:

```
{
  "character0001": {          ⟵ row key
    "info": {                 ⟵ column family
      "livingPlace": {        ⟵ column name
        t2: "Star Destroyer"  ⟵ version
        t1: "Tatooine"
      },
      "name": {
        t2: "Darth Vader"
        t1: "Anakin Skywalker"
      }
    },

    "history": {
      ...
    }
  },

  "character0002": {
    "info": {
      "livingPlace" : {
        t1: "unknown"
      },
```

HBase

Ways of data access

Using a set of two keys (row key, column family name), we specify all cells in a given column family.

Key: `["character0001", "info"]`

Result:

```
{
  "livingPlace": {
    t2: "Star Destroyer"
    t1: "Tatooine"
  },

  "name": {
    t2: "Darth Vader"
    t1: "Anakin Skywalker"
  }
}
```

```
{
  "character0001": {          ⟵ row key
    "info": {                 ⟵ column family
      "livingPlace": {        ⟵ column name
        t2: "Star Destroyer"  ⟵ version
        t1: "Tatooine"
      },
      "name": {
        t2: "Darth Vader"
        t1: "Anakin Skywalker"
      }
    },

    "history": {
      ...
    }
  },

  "character0002": {
    "info": {
      "livingPlace" : {
        t1: "unknown"
      },
```

HBase

Ways of data access

Using a row key we specify all columns from all column families.

Key: `["character0001"]`

Result:

HBase

Ways of data access

Using a row key we specify all columns from all column families.

Key: `["character0001"]`

Result:

HBase

Ways of data access

Using a row key we specify all columns from all column families.

Key: `["character0001"]`

Result:

```
{
  "info": {
    "livingPlace": {
      t2: "Star Destroyer"
      t1: "Tatooine"
    },

    "name": {
      t2: "Darth Vader"
      t1: "Anakin Skywalker"
    }
  },

  history: {
    ...
  }
}
```

```
{
  "character0001": {          ⟵ row key
    "info": {                 ⟵ column family
      "livingPlace": {        ⟵ column name
        t2: "Star Destroyer"  ⟵ version
        t1: "Tatooine"
      },
      "name": {
        t2: "Darth Vader"
        t1: "Anakin Skywalker"
      }
    },

    "history": {
      ...
    }
  },

  "character0002": {
    "info": {
      "livingPlace" : {
        t1: "unknown"
      },
```

Summary

- Column families stores were created to support specific processing modle which was a modified divide and conquer approach known now as MapReduce paradigm.

- Data processing based on MapReduce takes advantage of data locality, where nodes manipulate the data they have access to. Limiting needs for remote data access through slow channels results in faster processing. Therefore the crucial requirement is to keep local data together. Local in this case means all data needed to perform computation.

- All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common situation and should be automatically handled by the framework.

Bibliography

- [Ful] Piotr Fulmański, *NoSQL. Theory and examples*, Piotr Fulmański, 2021