Models and Systems for Big Data

KEY-VALUE AND COLUMNAR DATABASES

Nacéra Seghouani

Computer Science Department, CentraleSupélec
Laboratoire de Recherche en Informatique, LRI LaHDAK group
nacera.seghouani@centralesupelec.fr

2019-2020

Key-Value Databases

- A key-value store is a simple database that when presented with a simple string (the key) returns a large BLOB (binary large object) or basic object of data (the value).
- No need to scan the entire dictionary item by item to find what we are looking for.
- Unlike relational databases, there are no relations, no features (attributes) associated with relations, no constraints, no need for joins.
- While in relational database we avoid duplicating data, in key-value (in NoSQL in general) databases it is a common practice.
- The only extra feature supported by key-value databases are buckets, or collections for creating separate namespaces within a database.
- The same keys can be used for more than one namespace to implement something analogous to a relational schema.

Key-Value Database

- ☞ Simple data model. To say the truth, the query language is very primitive.
- ☞ Regardless of the type of an operation, we specify a namespace, and a key to indicate we want to perform an action on a key-value pair. Three main operations: put, get, and delete.
  - ✔ put adds a new key-value pair or updates a value if this key exists.
  - ✔ get returns the value for a given key if it exists.
  - ✔ delete removes a key and its value from if it exists.
- ☞ Other feature which simplifies the data model is its typelessnes. Values in key-value pairs have no type.
  - ✔ It's up to the application to determine what type of data is being used, such as an integer, string, JSON, XML file, or even binary data like image.
  - ✔ useful when the data type changes or we need to support two or more data types for the same attribute (for ex. different sensors)
- ☞ The key in a key-value store is very flexible and can be represented by many formats: number, string, JSON, or even such unusual as binary data (image) or even set or list.

Key-Value Database

- Main characteristics: simplicity, speed and scalability.
- Speed is a consequence of simplicity. Supported with internal design features optimizing performance, key-value databases delivers high-throughput for applications with data-intensive operations.
- Scalability is another most wanted feature all databases wants to have. No relational dependencies write and read requests are independent. Distribution could be easily achieved.
- Many key-value database systems. No standard query language comparable to SQL

Key-Value Database - Example

Invoice number	Customer number
1	10
2	30
3	20
4	30

Table: customer_details

Customer number	Customer name	Customer location
10	Dart Vader	Star Destroyer
20	Luke Skywalker	Naboo
30	C3PO	Tatooine

Table: invoice_details

Invoice number	Invoice item	Item name	Item quantity	Item price
1	1	lightsaber	1	100
1	2	black cloak	2	50
1	3	air filter	10	2
2	1	battery	1	25
3	1	lightsaber	5	75
3	2	belt	1	5
4	1	wires	1	10

Key-Value Database - Example

```
{
"Invoice number" : 1,
"Invoice details" : [
{"Item name" : "lightsaber", "Item quantity" : 1, "Item price" : 100},
{"Item name" : "black cloak", "Item quantity" : 2, "Item price" : 50},
{"Item name" : "air filter", "Item quantity" : 10, "Item price" : 2}
],
"Customer details" : {
"Customer name" : "Dart Vader"
"Customer location" : "Star Destroyer"
}
}
Shop[invoice:1:customerDetails] = ...
Shop[invoice:1:details] = ...
Shop[invoice:2:customerDetails] = ...
Shop[invoice:2:details] = ...
Shop[invoice:3:customerDetails] = ...
Shop[invoice:3:details] = ...
Shop[invoice:4:customerDetails] = ...
Shop[invoice:4:details] = ...
```

You might have noticed parallels between the key-naming convention: Concatenating a relation name with a key to identify a row, and a column name.

Key-Value Database - Example

Dealing with ranges of values is another thing which should be considered. If we need to process invoices by date or date range:

```
Shop[invoice:20171009:1:customerDetails] = ...
Shop[invoice:20171009:1:details] = ...
Shop[invoice:20171010:2:customerDetails] = ...
Shop[invoice:20171010:2:details] = ...
Shop[invoice:20171010:3:customerDetails] = ...
Shop[invoice:20171010:3:details] = ...
```

would be better than

```
Shop[invoice:1:customerDetails] = ...
Shop[invoice:1:details] = ...
Shop[invoice:1:date] = "20171009"
Shop[invoice:2:customerDetails] = ...
Shop[invoice:2:details] = ...
Shop[invoice:2:date] = "20171010"
Shop[invoice:3:customerDetails] = ...
Shop[invoice:3:details] = ...
Shop[invoice:3:date] = "20171010"
```

Key-Value Database

- Working with key-value database requires to carefully select key naming strategy.
- Balancing aggregation boundaries for values to make writes and reads more efficient as well as reduce latency.
- By storing all the information together, the number of disk seeks that must be performed to read all the needed data is reduced.
- When information is updated, the whole structure has to be written to a disc. As structure grows in size, the time required to read and write the data can increase.
- Another drawback is that a such big structure is read even if we need only a small piece of information – this way we waste time for reading and memory for storing it.
- Small values supports cache. Of course the size of cache is limited so we may be able to store say 2 big structures or 10 smaller.

Columnar Database

a columnar database stores data by columns rather than by rows

```
RowId EmpId Lastname Firstname Salary
001 10 Smith Joe 60000
002 12 Jones Mary 80000
003 11 Johnson Cathy 94000
004 22 Jones Bob 55000
```

```
001:10,Smith,Joe,60000;
002:12,Jones,Mary,80000;
003:11,Johnson,Cathy,94000;
004:22,Jones,Bob,55000;
```

To find all records with salaries between 40,000 and 50,000, the RDBMS would have to fully scan the table. The use of database indexes, which store all the values from a set of columns along with rowid pointers back into the original table.

```
001:60000;
003:94000;
002:80000;
004:55000;
```

However, maintaining indexes adds overhead to the system, when data is updated

Columnar Database

A column-oriented database serializes all of the values of a column together, then the values of the next column.

```
10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
60000:001,80000:002,94000:003,55000:004;
```

Any one of the columns more closely matches the structure of an index in a row-based system.
In a row-oriented indexed system, the key is the rowid that is mapped from indexed data.

In the column-oriented system, the primary is the data, which is mapped from rowids.

Columnar Database

- Whether or not a column-oriented system will be more efficient in operation depends heavily on the workload being automated.
- Operations that retrieve all the data for a given object (the entire row) are slower.
- A row-based system can retrieve the row in a single disk read, whereas numerous disk operations to collect data from multiple columns are required from a columnar database.
- Columnar databases boost performance by reducing the amount of data that needs to be read from disk, both by efficiently compressing the columnar data and by reading only the data necessary to answer the query.
- The compression permits columnar operations — like MIN, MAX, SUM, COUNT and AVG — to be performed very rapidly.
- Column storage is most useful for analytical queries and AI architectures because they get just a few attributes from every data entry.