Key-value stores

Introduction to NoSQL: Lecture 5

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 key-value stores

- Basic ideas and features
- Working with Riak

Basic ideas

Basic ideas

- From *array* to *dictionary*.

Basic ideas

- From *array* to *dictionary*.
- A key-value store is a simple database that when presented with a simple string (the key) returns an arbitrary large BLOB (value).

Key-value store vs. relational databases

- Simplicity is a key word associated with key-value databases where everything is simple:
    - there are no tables,
    - so there are no features associated with tables, such as columns and constraints on columns;
    - if there are no tables, there is no need for joins;
    - in consequence foreign keys do not exists and so key-value databases do not support a rich query language such as SQL.
- Contrary to relational database where **meaningless** keys are used, the keys in key-value databases are **meaningful**.
- While in relational database we avoid **duplicating data, in key-value (in NoSQL in general) databases it is a common practice**.

Key-value store vs. relational databases

- The only way to look up values is by key.
- Range queries are not supported out of the box.
- Queries from one key-value database may not be portable to the other.

Essential features of key-value databases

- **Simplicity** In key-value databases, we work with a very simple data model which resembles dictionary. The syntax for manipulating data is simple. There are three operations performed on a key-value store: `put`, `get`, and `delete`.
- **Speed** There is no need for complicated query resolving logic. Every query directly specify the key and always it is only one key.
- **Scalability** Working with key-value databases we have no relational dependencies and all write and read requests are independent and this seems to be a perfect state for scaling.

Key is the key

- In key-value databases, generally speaking, there is no method to scan or search values so the right key naming strategy is crucial.
- While working with relational databases, counters or sequences are very often used to generate keys. Working with numbers is the easiest way to ensure that every new call for a new key returns a value (number in this case) which is unique and unused so far.
- Because of the way **relational databases** work, it makes sense (sometimes it is considered as a good practice) to have such a **meaningless** keys in this case.
- **In key-value databases the rules are different.** If there are no tables, there are no rows and columns so the question arise: how to "join", combine or somehow collect all information related to a given object? Use right aggregation and key names – is the answer.

Key is the key

First attempt

customer table

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

```
CustomerDetails[10] = 'Dart Vader'
```

Key is the key

Add one more namespace

customer table

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

```
CustomerDetails[10] = 'Dart Vader'
CustomerLocation[10] = 'Star Destroyer'
```

Key is the key

What about aggregates?
All under a one key

invoice details table

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

```
InvoiceDetails[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}
  ]
}
```

Key is the key

What about aggregates?
All under a one key

```
InvoiceDetails[1] =
{
  "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"
  }
}
```

invoice details table

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

So, maybe use bigger aggregates?

Key is the key

What about aggregates?
All under a one key

```
InvoiceDetails[1] =
{
  "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"
  }
}
```

invoice details table

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

So, maybe use bigger aggregates?

Be careful!

Key is the key

Rules

Avoid to use many namespaces.
Remember: key is the key

```
Shop[Customer:10:name] = 'Dart Vader'
Shop[Customer:10:location] = 'Star Destroyer'
```

Key is the key

Rules

Avoid to use many namespaces.
Remember: key is the key

```
Shop[Customer:10:name] = 'Dart Vader'
Shop[Customer:10:location] = 'Star Destroyer'
```

`Shop` - only one namespace

Key is the key

Rules

invoice table

invoice number	customer number	customer name	customer location
1	10	Dart Vader	Star Destroyer
2	30	C3PO	Tatooine
3	20	Luke Skywalker	Naboo
4	30	C3PO	Tatooine

Do not follow relational pattern.

Never ever copy relational data model.

Following relational pattern for C3PO we may have

```
Shop[customer:30:invoice:2] = ...
Shop[customer:30:invoice:4] = ...
```

which

Key is the key

Rules

invoice table

invoice number	customer number	customer name	customer location
1	10	Dart Vader	Star Destroyer
2	30	C3PO	Tatooine
3	20	Luke Skywalker	Naboo
4	30	C3PO	Tatooine

Do not follow relational pattern.

Never ever copy relational data model.

Following relational pattern for C3PO we may have

```
Shop[customer:30:invoice:2] = ...
Shop[customer:30:invoice:4] = ...
```

which **is useless.**

Key is the key

Rules

invoice table

invoice number	customer number	customer name	customer location
1	10	Dart Vader	Star Destroyer
2	30	C3PO	Tatooine
3	20	Luke Skywalker	Naboo
4	30	C3PO	Tatooine

Do not follow relational pattern.

Never ever copy relational data model.

Following relational pattern for C3PO we may have

```
Shop[customer:30:invoice:2] = ...
Shop[customer:30:invoice:4] = ...
```

which **is useless.**

Better:

```
Shop[customer:30:invoice:1] = ...
Shop[customer:30:invoice:2] = ...
```

Key is the key

Rules

Mind aggregation you expect to use

If we suppose that we will use the data most often for processing orders

```
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] = ...
```

Key is the key
Rules

Mind range queries you expect to use

Is it good?

```
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"
Shop[invoice:4:customerDetails] = ...
Shop[invoice:4:details] = ...
Shop[invoice:4:date] = "20171013"
```

Key is the key
Rules

Mind range queries you expect to use

Is it good?

```
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"
Shop[invoice:4:customerDetails] = ...
Shop[invoice:4:details] = ...
Shop[invoice:4:date] = "20171013"
```

Key is the key
Rules

Mind range queries you expect to use

This is much better

```
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] = ...
Shop[invoice:20171013:4:customerDetails] = ...
Shop[invoice:20171013:4:details] = ...
```

Key is the key

Rules

Mind range queries you expect to use

This is the best

```
Shop[invoice:20171009:1:customerDetails] = ...
Shop[invoice:20171009:1:details] = ...
Shop[invoice:20171010:1:customerDetails] = ...
Shop[invoice:20171010:1:details] = ...
Shop[invoice:20171010:2:customerDetails] = ...
Shop[invoice:20171010:2:details] = ...
Shop[invoice:20171013:1:customerDetails] = ...
Shop[invoice:20171013:1:details] = ...
```

Values

Rules

We have to balance aggregation boundaries for values to make writes and reads more efficient as well as reduce latency.

Bellow there are some strategies. If they are good or bad depends on you.

- Values which are big aggregates.
- Keep together values commonly used.
- Small values supports cache.

Summary

- No tables, so there are no features associated with tables, such as columns types or constraints on columns.
- There is no tables so there is no need for joins. In consequence foreign keys do not exists.
- Do not support a rich query language such as SQL. Saying the truth, query language is very primitive and limited to simple select, insert and delete equivalent commands.
- Contrary to relational databases where meaningless keys are used, the keys in key-value databases are meaningful and play crucial role.
- Although key-value databases don’t have any structure we have to very carefully balance aggregation boundaries for values to make writes and reads more efficient as well as reduce latency.

Bibliography

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