I404B NoSQL

Session 2

Key-Value Model: Riak, Memcached, Redis

Sébastien Combéfis

Fall 2019

Objectives

- The key-value model
  - Principle and characteristics of key-value storage
  - Use case and non-use cases
  - Data repartition models
- Examples of key-value databases
  - Riak
  - Memcached
  - Redis

Key-Value Model

Key-Value (1)

- Key-value databases similar to hashtables

  Stores key-value pairs, identifiable by their key

- Similar to a relational table with two columns

  Used when searching on primary key

- Very good performance thanks to indexing on the key

Id	**Name**
16133	Yannis
16067	Théo
16050	Yassine
15089	Maxime

Key-Value (2)

- The simplest NoSQL storage space

  Regarding the API to use it

- Mainly three operations on the store

  Retrieve/set a value for a key, delete a key

Data Type

- The stored value is a blob type *(Binary Large OBject)*

  It is up to the application to manage the values and their format

- Sometimes limits on the size of stored values

  For performance reasons

- Sometimes domain constraints on aggregates

  Redis supports lists, sets and hashes

Basic API

- Three basic operations supported by all engines
  - `get(k)` retrieves the v value associated to the k key
  - `put(k, v)` adds the (k, v) pair in the store
  - `delete(k)` deletes the pair associated to the k key
- The engine can propose specific operations

  Redis proposes the union of sets, for example

Use Case

- Storing session information for a website

  Unique identifier convenient for a key-value database

- Profiles and preferences of a given user

  User is characterised by a unique username

- Shopping carts on an e-commerce website

  Storing the current shopping cart of a user

Non-Use Case

- Links to establish between data related to different keys

  Following the links between data is not easy

- Backup of several keys and failure of some backups

  Not possible to restore operations already realised

- Not possible to make requests on the values

  Except for some specific engines

Distribution Model

Distribution Model

- Several possible models to operate a cluster

  End of scale up (larger server) for scale out (more servers)

- The aggregate information unit can be easily distributed

  Fine granulometry of information

- Several reasons to use a cluster
  - Ability to manage larger amounts of data
  - Provide a larger read/write traffic
  - Resist to network slowdowns or failures

Unique Server

- No distribution in the simplest version

  Execution on a single machine that manages reads/writes

- Solution very simple to implement and operate
  - Easy to manage for operators
  - Easy to reason for application developers

- Suitable for graph-oriented databases

  Where operations to perform are often aggregations

Sharding (1)

- Store should be busy with several users

  When they are accessing different parts of the data

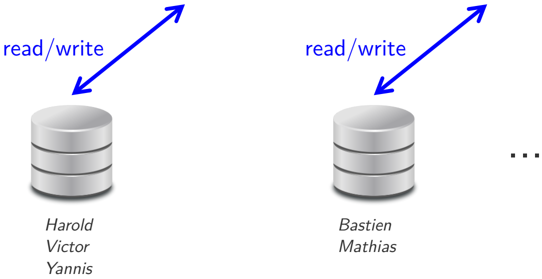
- Sharding places data on several servers

  Horizontal scalability with with deployment of several nodes

- Load balancing between the different servers

  If the users are requesting different data

Sharding (2)


read/write
read/write
Harold
Victor
Yannis
Bastien
Mathias
...

Load Balancing

- Ideally, the load is well distributed between clients

  With 5 nodes, each node manages 20% of the load

- Data accessed together must be place on the same node
  - Using aggregate as the distribution unit
  - Using the geographical location of data
  - Collecting aggregates by common access probability

- Possibility to have automatic sharding

  The engine manages the sharding and data rebalancing

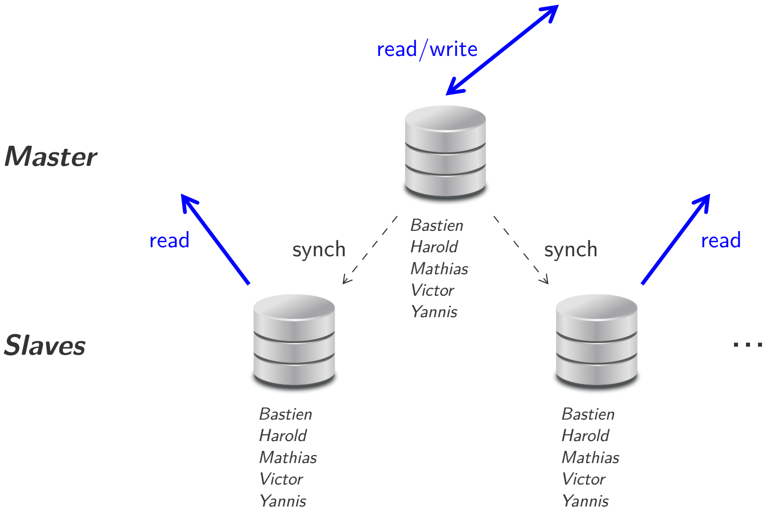
Master-Slave Replication (1)

- Data replicated on several nodes

  Suitable when more reads than writes

- Two kinds of nodes in the system
  - A master node responsible for data and update
  - Several slave nodes that are replicates of the master
- Two properties for this kind of replication
  - Read resilience allows reads if the master fails
  - Values read by users may differ by inconsistency

Master-Slave Replication (2)



Data Scattering

- Routing requests based on the type

  Read sent to the slaves and writes to the master

- Slaves synchronisation by replication process
  - Modifications on the master are communicated to the slaves
  - Election of a slave as the master if it fails

- Two modes of choice of the master
  - Manual choice by configuration
  - Automatic choice by dynamic election

Peer-to-Peer Replication (1)

- Data replicated on several nodes that are all equal

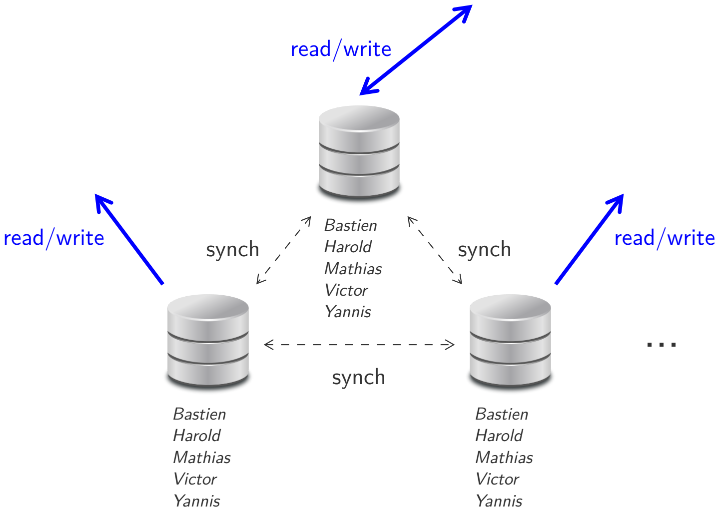
  Brings scalability for write operations

- Synchronising all the nodes at each write

  Concurrent and permanent write conflicts, not like with read

- Several properties for this kind of replication
  - Complete read and write resilience
  - Values read by different users different by inconsistency

Peer-to-Peer Replication (2)



Sharding vs. Replication

- Sharding distributes the load, no resilience

  Different data on different nodes

- Replication offers resilience, heavy synchronisation

  Same data places on different nodes

Strategy	**Scaling**	**Resilience**	**Inconsistency**
Sharding	Write	–	–
M/S Replication	Read	Read	Yes
P2P Replication	Read/Write	Read/Write	Yes

Combining Sharding and Replication

- Master-slave replication and sharding
  - Possibility to have several masters, but only one by data
  - Node with a single role or mixed roles
- Peer-to-peer replications and sharding
  - Data sharded on hundreds of nodes
  - Data is replicated on N nodes (replication factor)


Riak

Riak

- Created and developed by the Basho company

  Company founded in 2008 and develops Riak and other solutions

- Active company and last version in may 2019

  Riak is developed in Erlang and the last version is Riak 2.9.0

- Decentralised NoSQL engine based on Amazon Dynamo

  Scales by adding new machines to the cluster

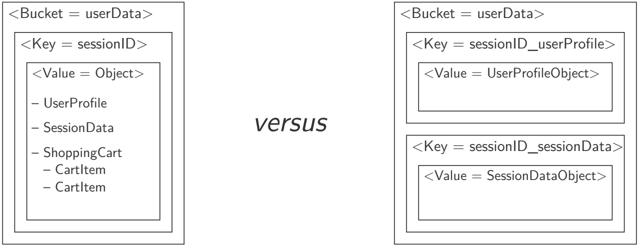
Bucket

- Riak can store keys in buckets

  Acts as a namespace for keys

- Several possibilities to operate buckets

  Composed values or separation as "specific objects"



Domain Bucket

- Domain bucket can store a precise type of data

  Automatic serialisation/deserialisation by the client

- Separation in buckets to segment data
  - Possible to only read objects that you want to read
  - Possible to use the same key through different buckets

- Fight against impedance mismatch

  Store directly contains application objects

Installing Riak

- Riak is a program written in Erlang
- Several programs proposed after installation
  - `riak` to control Riak nodes
  - `riak-admin` for administration operations

Starting a Node

- Starting a Riak node with the `riak` executable

  Starting with the `start` option and stopping with the `stop` option

```
& riak start
```

```
& riak ping
pong
```

riak Python Module

- riak Python module to query the store

  Opening a connection and then methods to make queries

```python
import riak

client = riak.RiakClient(protocol='http', http_port=8098)

print(client.ping())
print(client.get_buckets())
```

```
True
[]
```

Creating a Bucket

- Creating a new bucket with the `bucket` method

  To be called on the Riak client

- Return a `RiakBucket` object

  Used to add and read key-value pairs

```python
import riak

client = riak.RiakClient(protocol='http', http_port=8098)

bucket = client.bucket('students')
print(bucket)
```

```
<RiakBucket 'students'>
```

Data Manipulation

- Creating a new data with the `new` method

  Return a `RiakObject` object that can be stored

```python
import riak

client = riak.RiakClient(protocol='http', http_port=8098)
bucket = client.bucket('students')

print(bucket.get('16050').data)

yassine = bucket.new('16050', 'Yassine')
yassine.store()
print(bucket.get('16050').data)
```

```
None
Yassine
```

Riak Cluster

- Distributing data with a consistent hash
  - Minimises keys remapping when the number of nodes changes
  - Distributed the data well and minimises hotspots
- Using SHA-1 and the 160 bits spaces as ring
  - Cutting the ring in partitions called “virtual nodes”
  - Each physical node hosts several `vnodes`

Memcached

Memcached

- General purpose distributed cache system

  Speed up a website by caching objects in RAM

- Used in combination with another database

  For example from PHP as a cache to a MySQL database

- Memcached is a program written in C

Architecture (1)

- Built on a client/server architecture

  Server services exposed on the 11211 port by default

- The client makes queries by key on the store

  Keys are at most 250 bytes and values are up to 1 Mio

- A client knows all the servers
  - Servers do not communicate between them
  - Computation of a hash on the key to chose the server

Architecture (2)

- Store data are stored in RAM
  - Oldest values deleted if not enough RAM
  - Memcached to be used as a transient cache
- Act as a big hashtable

  Key-value pairs are stored in this hashtable

memcache Python Module

- `memcache` Python module to query the store

Opening a connection and methods for commands

```python
import memcache

mc = memcache.Client(['127.0.0.1:11211'])

print(mc.get('16133'))
print(mc.set('16133', 'Yannis'))
print(mc.get('16133'))
print(mc.delete('16133'))
print(mc.get('16133'))
```

```
None
True
Yannis
1
None
```

The Trivago Example

- Trivago uses Memcached for its cache layer

  Avoid a lot of direct requests to the main database

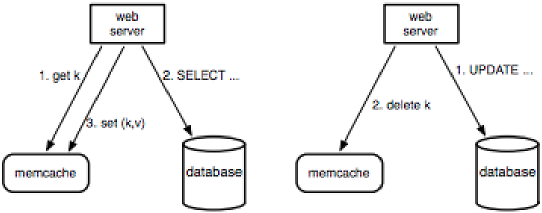
- Big sudden issue with logs filled with Memcached errors
  - Failures of `get` and overload of the database
  - Botnet from more than 200 countries with 70K unique IPs...
  - Memcached network interface saturation beyond 1 Gbit/s

The Facebook Example (1)

- Facebook uses Memcached for a distributed store

  Distributed storage of key-value pairs in memory

- Two different usages for request or generic
  - Used as a *demand-filled look-aside cache*
  - And also deployment of a generic distributed store



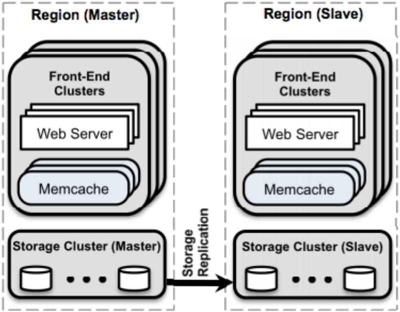
The Facebook Example (2)

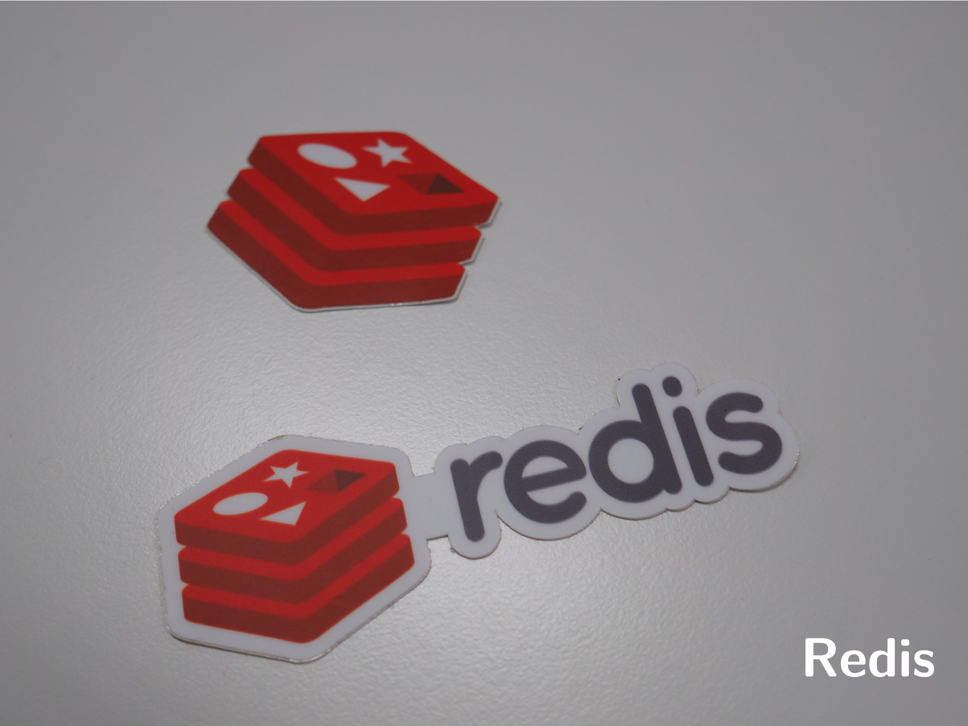
- No coordination server-server with Memcached

  "Only" a local in-memory hashtable of a server

- Replication inside a server cluster

  Data flow from the master to the slaves





Redis

Redis

- Database engine in memory

  Manipulate data structure as quickly as possible

- Also plays the role of a data cache

  Similar to Memcached with a richer and stronger model

- Restriction on the manipulated values

  Five possible kinds of values stored in the database

Value Type

- Possible to manipulate specific data types with Redis

  And do not manipulate documents like other databases

- Five different types of data
  - Strings, and numeric or binary value
  - Lists of strings (insertion order maintained)
  - Set of strings, unsorted and without duplicate
  - Hash (dictionary), not hierarchical
  - Sorted set with association of a note for each element

Installing Redis

- Redis is a program written in C
- Several programs proposed after installation
  - `redis-server` to start a Redis server
  - `redis-cli` is a command-line client
  - `redis-benchmark` makes a performance test

Starting the Server

- Starting the server and testing the connection

  Test of a ping to the server from the command line

```
& redis-server
```

```
& redis-cli
127.0.0.1:6379> ping
PONG
```

Manipulating String

- Several basic commands to manipulate strings
  - `SET` adds a new string in the store
  - `GET` retrieves the value associated to a key
  - `DEL` deletes a key from the store

```
& redis-cli
127.0.0.1:6379> GET 15089
(nil)
127.0.0.1:6379> SET 15089 "Maxime"
OK
127.0.0.1:6379> GET 15089
"Maxime"
127.0.0.1:6379> DEL 15089
(integer) 1
127.0.0.1:6379> GET 15089
(nil)
```

redis Python Module

- redis Python module to query the store

Opening a connection then methods for commands

```python
import redis

r = redis.StrictRedis(host='localhost', port=6379, db=0)

print(r.get('15089'))
print(r.set('15089', 'Maxime'))
print(r.get('15089'))
print(r.delete('15089'))
print(r.get('15089'))
```

```
None
True
b'Maxime'
1
None
```

Manipulating Hash

- Several basic commands to manipulate hashes
  - `HSET` adds an entry in the hash table of a key
  - `HVALS` retrieves the complete hash table of a key
  - `HGET` retrieves the value of an entry of a hash table
  - `HDEL` deletes an entry of a hash table

```
& redis-cli
127.0.0.1:6379> HSET 16067 firstName Théo
(integer) 1
127.0.0.1:6379> HSET 16067 favColour green
(integer) 1
127.0.0.1:6379> HVALS 16067
1) "Théo"
2) "green"
127.0.0.1:6379> HGET 16067 favColour
"green"
```

Hash/Python Dictionary Equivalence

- Direct mapping between hashes and Python dictionaries

  Initialisation of a hash with `hmset`

```python
import redis

r = redis.StrictRedis(host='localhost', port=6379, db=0)
r.hmset('10003', {
    'firstName': 'Théo',
    'favColour': 'green'
})
print(r.dbsize())
print(r.hgetall('10003'))
```

```
1
{b'firstName': b'Théo', b'favColour': b'green'}
```

Manipulating List

- Several basic commands to manipulate lists
  - `LPUSH` adds an entry to the left of a list
  - `LPOP` removes the entry to the left of a list
  - `RPUSH` adds an entry to the right of a list
  - `RPOP` removes the entry to the right of a list
  - `LRANGE` extract a sublist from a list

```
& redis-cli
127.0.0.1:6379> RPUSH students 16133
(integer) 1
127.0.0.1:6379> RPUSH students 15089
(integer) 2
127.0.0.1:6379> LRANGE students 0 -1
1) "16133"
2) "15089"
```

List/Python List Equivalence

- Direct mapping between lists and Python lists

Initialisation of a list with rpush

```python
import redis

data = ['16133', '15089']

r = redis.StrictRedis(host='localhost', port=6379, db=0)
r.delete('students')
r.rpush('students', *data)

data = r.lrange('students', 0, -1)
for elem in data:
    print(elem)
```

```
b'16133'
b'15089'
```

Data Persistence

- Redis is a in-memory only database

  Once the server exits, all data is lost

- Possibility to regularly save data on disk

  Using the RDB system by default, for regular snapshots

- Automatic reloading of the database

  If a `.rdb` file is in the right folder

Expiration

- Possible to choose the lifetime of elements

  Using the `EXPIRE` command

- An element in a cache should not live forever

Redis Social Network Example

- Storing a simple social network with Redis

  Defining the format of key-value pairs to use

- Two kinds of objects in the store
  - **User** has a name and can be followed by others
  - **Post** is a message, a picture...

- A user can have several posts

  Storing the list of posts of a user

Key Format (1)

- Defining the format of the keys to use

  Must be a simple string

- Convention to have unique keys
  - **User**

    `user:1:name` → `Mathias`
    `username:Mathias` → `1`

  - **Post**

    `post:1:content` → `Hi Théo, you rock!`
    `post:1:user` → `1`

Key Format (2)

- Posts and follow relations with lists/sets

  Integer numbers lists referring users and posts

- Using "sub-keys" from `user`
  - **Posts list**

    `user:1:posts` → `[3, 2, 1]`

  - **Follow relation**

    `user:1:follows` → `{2, 3, 4}`
    `user:1:followed_by` → `{3}`

Automatic Identifier

- Possibility to increment a value with the `INCR` command

  The value must represent an integer number

- Adding two pairs to represent the next IDs

  Keys `next_user_id` *and* `next_post_id`

```python
import redis

r = redis.StrictRedis(host='localhost', port=6379, db=0)
r.set('next_user_id', 0)
print(r.get('next_user_id'))

r.incr('next_user_id')
print(r.get('next_user_id'))
```

```
b'0'
b'1'
```

Creating a New User

- Definition of a method to create a new user

```python
import redis

r = redis.StrictRedis(host='localhost', port=6379, db=0)
r.set('next_user_id', 0)

def create_user(username):
    uid = int(r.get('next_user_id'))
    r.set('user:{}:name'.format(uid), username)
    r.set('username:{}'.format(username), uid)
    r.incr('next_user_id')

create_user('Mathias')
create_user('Théo')

print(r.get('user:0:name'))
print(r.get('user:1:name'))
```

```
b'Mathias'
b'Théo'
```

Top 5 Redis Use Cases

- Session cache and Full Page Cache (FPC)

  The advantage of Redis is persistance

- Implementation of an efficient message queue

  For example with the Celery tool for Distributed Task Queue

- Developing a leaderboard with counting

- Execution of scripts with Pub/Sub events

References

- Wishmitha S. Mendis, *From RDBMS to Key-Value Store: Data Modeling Techniques*, October 29, 2017. https://medium.com/@wishmithasmendis/from-rdbms-to-key-value-store-data-modeling-techniques-a2874906bc46
- Darren Perucci, DZone, *Redis Replication vs Sharding*, June 15, 2016. https://dzone.com/articles/redis-replication-vs-sharding
- Ivana Petrovic and Polina Pokalyukhina, *How trivago Reduced Memcached Memory Usage by 50%*, December 19, 2017. https://tech.trivago.com/2017/12/19/how-trivago-reduced-memcached-memory-usage-by-50
- Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung and Venkateshwaran Venkataramani (2013). Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013)*. https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf
- Joe Engel, *Top 5 Redis Use Cases*, November 7, 2017. https://www.objectrocket.com/blog/how-to/top-5-redis-use-cases

Credits

- Logo pictures from Wikipedia.
- SioW, July 3, 2006, https://www.flickr.com/photos/curioussiow/182224885.
- Shepherd Distribution Services, October 15, 2010, https://www.flickr.com/photos/shepherd-distribution-services/5395849861.
- https://openclipart.org/detail/94723/database-symbol.
- heschong, May 14, 2007, https://www.flickr.com/photos/heschong/510216272.
- DM, April 27, 2011, https://www.flickr.com/photos/dmott9/5662744650.
- othree, November 19, 2013, https://www.flickr.com/photos/othree/10945272436.