

NDBI040

# Big Data Management and NoSQL Databases

Lecture 5. Key-value stores

Doc. RNDr. Irena Holubova, Ph.D.

[holubova@ksi.mff.cuni.cz](mailto:holubova@ksi.mff.cuni.cz)

<http://www.ksi.mff.cuni.cz/~holubova/NDBI040/>

# Key-value store

## Basic characteristics

- The simplest NoSQL data store
  - A hash table (map)
  - When all access to the database is via primary key
- Like a table in RDBMS with two columns:
  - ID = key
  - NAME = value
    - BLOB with any data
- Basic operations:
  - get the value for the key
  - put a value for a key
    - If the value exists, it is overwritten
  - delete a key from the data store
- simple → great performance, easily scaled
- simple → not for complex queries, aggregation needs, ...

# Key-value store

## Representatives

riak



redis



MemcachedDB



ORACLE®

BERKELEY DB

Hamster DB  
embedded database



not open-source

open-source  
version



Project  
Voldemort

# Key-value store

## Suitable Use Cases

### Storing Session Information

- Every web session is assigned a unique `session_id` value
- Everything about the session can be stored by a single PUT request or retrieved using a single GET
- Fast, everything is stored in a single object

### User Profiles, Preferences

- Every user has a unique `user_id`, `user_name` + preferences (e.g., language, colour, time zone, which products the user has access to, ... )
- As in the previous case:
  - Fast, single object, single GET/PUT

### Shopping Cart Data

- Similar to the previous cases



# Key-value store

## When Not to Use

### Relationships among Data

- Relationships between different sets of data
- Some key-value stores provide link-walking features
  - Not usual

### Multioperation Transactions

- Saving multiple keys
  - Failure to save any one of them → revert or roll back the rest of the operations

### Query by Data

- Search the keys based on something found in the value part

### Operations by Sets

- Operations are limited to one key at a time
- No way to operate upon multiple keys at the same time

# Key-value store

## Query

- We can query by the **key**
- To query using some attribute of the value column is (typically) not possible
  - We need to read the value to figure out if the attribute meets the conditions
- What if we do not know the key?
  - Some systems enable to retrieve the list of all keys
    - Expensive
  - Some support searching inside the value
    - Using, e.g., a kind of full text index
      - The data must be indexed first
      - Riak search (see later)



# Key-value store

## Query

- How to design the key?
  - Generated by some algorithm
  - Provided by the user
    - e.g., userID, e-mail
  - Derived from time-stamps (or other data)
- Typical candidates for storage: session data (with the session ID as the key), shopping cart data (user ID), user profiles (user ID), ...
- Expiration of keys
  - After a certain time interval
  - Useful for session/shopping cart objects



# RIAK





# Key-value store



## Riak

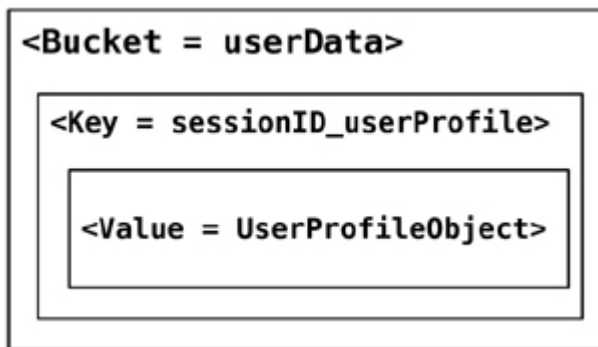
- Open source, distributed database
  - First release: 2009
  - Implementing principles from Amazon's Dynamo
- OS: Linux, BSD, Mac OS X, Solaris
- Language: Erlang, C, C++, some parts in JavaScript
- Built-in MapReduce support
- Stores keys into **buckets** = a namespace for keys
  - Like tables in a RDBMS, directories in a file system, ...
  - Have set of common properties for its contents
    - e.g., number of replicas

# Riak Buckets

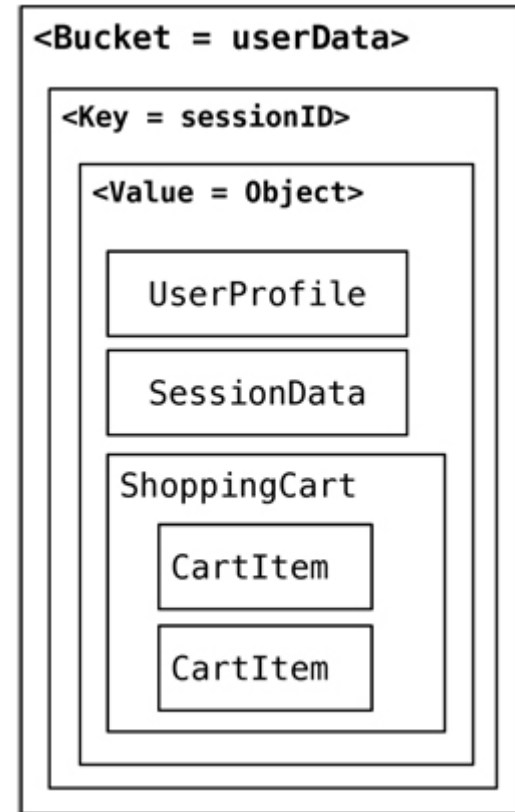
namespace  
for keys

Oracle	Riak
database instance	Riak cluster
table	bucket
row	key-value
rowid	key

## Terminology in Oracle vs. Riak



Adding type of data to the key,  
still everything in a single bucket



Single object for all data,  
everything in a single bucket

Separate buckets for different  
types of data

# Key-value store


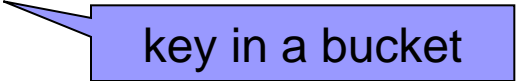
## Example



```
Bucket bucket = getBucket(bucketName);  
IRiakObject riakObject =  
    bucket.store(key, value).execute();
```

```
Bucket bucket = getBucket(bucketName);  
IRiakObject riakObject =  
    bucket.fetch(key).execute();  
byte[] bytes = riakObject.getValue();  
String value = new String(bytes);
```

# Riak Usage

- HTTP – default interface
  - GET (retrieve), PUT (update), POST (create), DELETE (delete)
  - Other interfaces: Protocol Buffers, Erlang interface
  - We will use curl (**curl --help**)
    - Ccommand-line tool for transferring data using various protocols
- Keys and buckets in Riak:
  - Keys are stored in buckets (= namespaces) with common properties
    - `n_val` – replication factor
    - `allow_mult` – allowing concurrent updates
    - ...
  - If a key is stored into non-existing bucket, it is created
  - Keys may be user-specified or generated by Riak
- Paths:
  - `/riak/<bucket>`  a particular bucket
  - `/riak/<bucket>/<key>`  key in a bucket

# Riak Usage – Examples

## Working with Buckets

- List all the buckets:

```
curl http://localhost:10002/riak?buckets=true
```

- Get properties of bucket `foo`:

```
curl http://localhost:10002/riak/foo/
```

- Get all keys in bucket `foo`:

```
curl http://localhost:10002/riak/foo?keys=true
```

- Change properties of bucket `foo`:

```
curl -X PUT http://localhost:10002/riak/foo -H  
  "Content-Type: application/json" -d '{"props" : {  
    "n_val" : 2 } }'
```

# Riak Usage – Examples

## Working with Data

- Storing a plain text into bucket `foo` using a generated key:

```
curl -i -H "Content-Type: plain/text" -d "My text"  
http://localhost:10002/riak/foo/
```

HTTP POST

- Storing a JSON file into bucket `artist` with key `Bruce`:

```
curl -i -H "Content-Type: application/json" -d  
'{"name": "Bruce"}'  
http://localhost:10002/riak/artists/Bruce
```

HTTP GET

- Getting an object:

```
curl http://localhost:10002/riak/artists/Bruce
```

# Riak Usage – Examples

## Working with Data

- Updating an object:

HTTP PUT

```
curl -i -X PUT -H "Content-Type: application/json" -  
d '{"name":"Bruce", "nickname":"The Boss"}'  
http://localhost:10002/riak/artists/Bruce
```

```
curl http://localhost:10002/riak/artists/Bruce
```

- Deleting an object:

HTTP DELETE

check the value

```
curl -i -X DELETE  
http://localhost:10002/riak/artists/Bruce
```

```
curl http://localhost:10002/riak/artists/Bruce
```

# Riak Links

- Allow to create relationships between objects
  - Like, e.g., foreign keys in relational databases, or associations in UML
- Attached to objects via Link header

- Add albums and links to the performer:

```
curl -H "Content-Type: text/plain" -H 'Link:
  </riak/artists/Bruce>; riaktag="performer"' -d
  "The River"
http://localhost:10002/riak/albums/TheRiver
```

```
curl -H "Content-Type: text/plain" -H 'Link:
  </riak/artists/Bruce>; riaktag="performer"' -d
  "Born To Run"
http://localhost:10002/riak/albums/BornToRun
```



# Riak Links

- Find the artist who performed the album The River

```
curl -i
```

```
http://localhost:10002/riak/albums/TheRiver/artists,performer,1
```

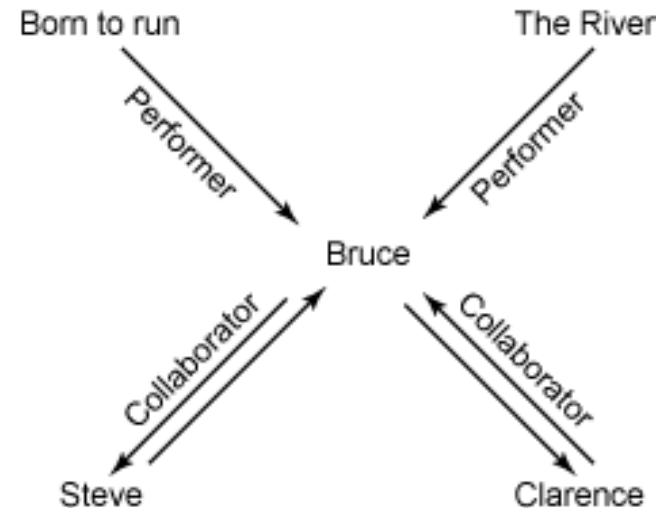
- ☐ Restrict to bucket `artists`
- ☐ Restrict to tag `performer`
- ☐ `1` = include this step to the result

# Riak Links

- Which artists collaborated with the artist who performed The River

```
curl -i  
http://localhost:10002/  
riak/albums/TheRiver/artists,_,0/artists,collaborator,1
```

- ☐ \_ = wildcard (any relationship)
- ☐ 0 = do not include this step to the result



Assuming  
such data



# Riak Search

- A distributed, full-text search engine
- Provides the most advanced query capability next to MapReduce
- Key features:
  - Support for various mime types
    - JSON, XML, plain text, ...
  - Support for various analyzers (to break text into tokens)
    - A white space analyzer, an integer analyzer, a no-op analyzer, ...
  - Exact match queries
  - Scoring and ranking for most relevant results
  - ...

# Riak Search

- First the data must be indexed:
  1. Reading a document
  2. Splitting the document into one or more fields
  3. Splitting the fields into one or more terms
  4. Normalizing the terms in each field
  5. Writing {Field, Term, DocumentID} to an index
- Indexing: `index <INDEX> <PATH>`
- Searching: `search <INDEX> <QUERY>`

# Riak Search

## ■ Queries:

- Wildcards: `Bus*`, `Bus?`

- Range queries:

  - `[red TO rum]` = documents with words containing “red” and “rum”, plus any words in between

  - `{red TO rum}` = documents with words in between “red” and “rum”

- AND/OR/NOT and grouping: `(red OR blue) AND NOT yellow`

- Prefix matching

- Proximity searches

  - `"See spot run"~20` = documents with words within a block of 20 words

# Key-value store

## Transactions in Riak

$$W > N/2$$

$$R + W > N$$



- BASE (**B**asically **A**vailable, **S**oft state, **E**ventually consistent)

- Uses the concept of **quorums**

- ☐ **N** = replication factor

- Default  $N = 3$

- ☐ Data must be written at least at **W** nodes

- ☐ Data must be found at least at **R** nodes

- Values **W** and **R**:

- ☐ Can be set by the user for every single operation

- ☐ `all / one / quorum / default / an integer value`

- Example:

- ☐ A Riak cluster with  $N = 5$ ,  $W = 3$

- ☐ Write is reported as successful  $\leftrightarrow$  reported as a success on  $> 3$  nodes

- ☐ Cluster can tolerate  $N - W = 2$  nodes being down for write operations

- **dw** = durable write

- ☐ More reliable write, not just “promised” that started

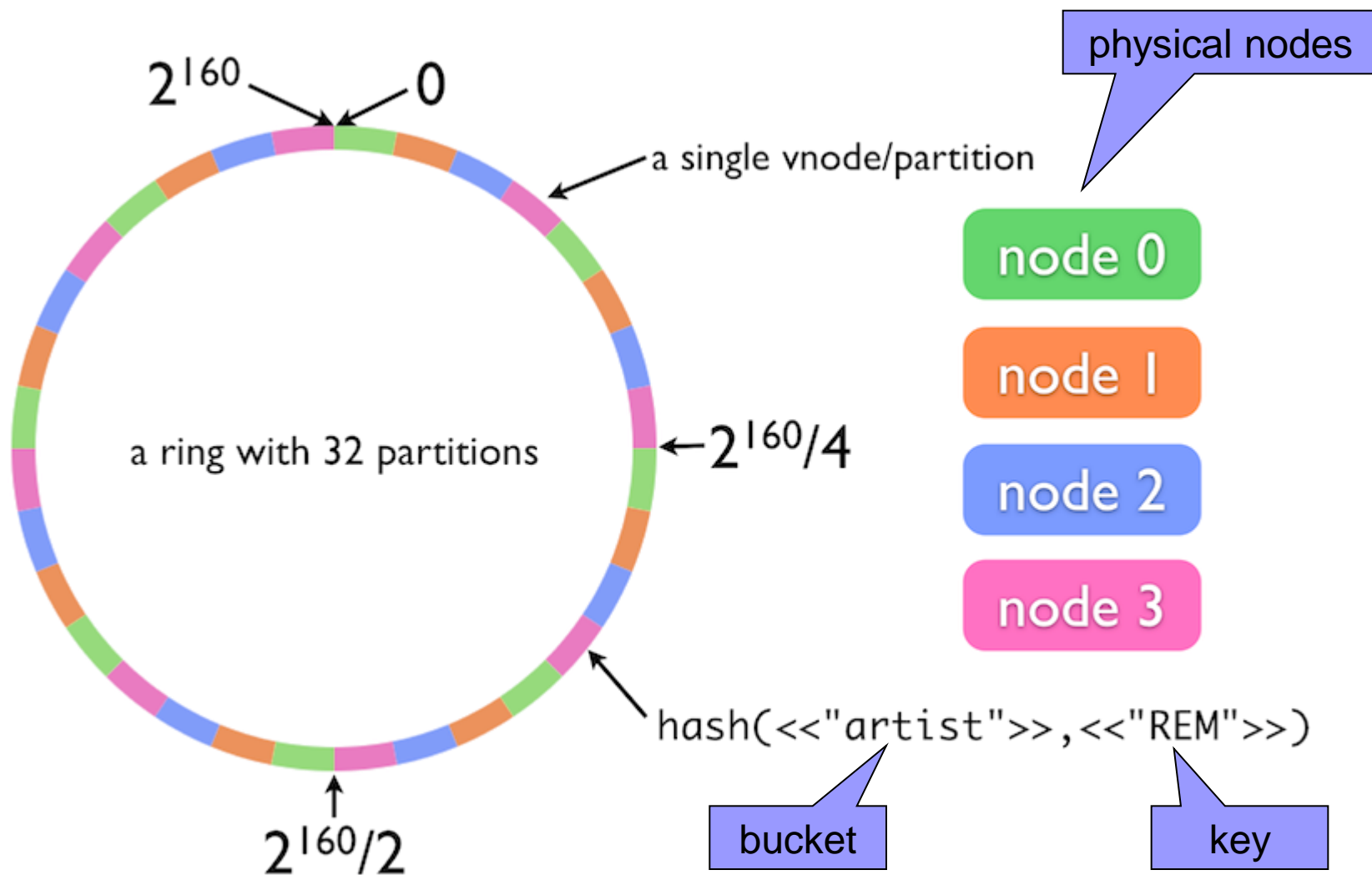
- **rw** = for deletes (read and delete)

```
{  
  "props": {  
    ...  
    "dw": "quorum",  
    "n_val": 5,  
    "name": "cart",  
    "postcommit": [],  
    "pr": 0,  
    "precommit": [],  
    "pw": 0,  
    "r": "quorum",  
    "rw": "quorum",  
    "w": "quorum",  
    ...  
  }  
}
```

# Key-value store

## Clustering in Riak

- Center of any cluster: 160-bit integer space ([Riak ring](#)) which is divided into equally-sized partitions
- [Physical nodes](#) run [virtual nodes](#) (vnodes)
  - Each physical node in the cluster is responsible for:  
$$1 / (\text{total number of physical nodes})$$
of the ring
  - Number of vnodes on each node:  
$$(\text{number of partitions}) / (\text{number of physical nodes})$$
- Nodes can be added and removed from the cluster dynamically
  - Riak will redistribute the data accordingly
- Example:
  - A ring with 32 partitions
  - 4 physical nodes
  - 8 vnodes per node





# Key-value store

## Clustering in Riak

### ■ No master node

- Each node is fully capable of serving any client request
- Uses **consistent hashing** to distribute data around the cluster
  - Minimizes reshuffling of keys when a hash-table data structure is rebalanced
  - Only  $k/n$  keys need to be remapped on average
    - $k$  = number of keys
    - $n$  = number of slots

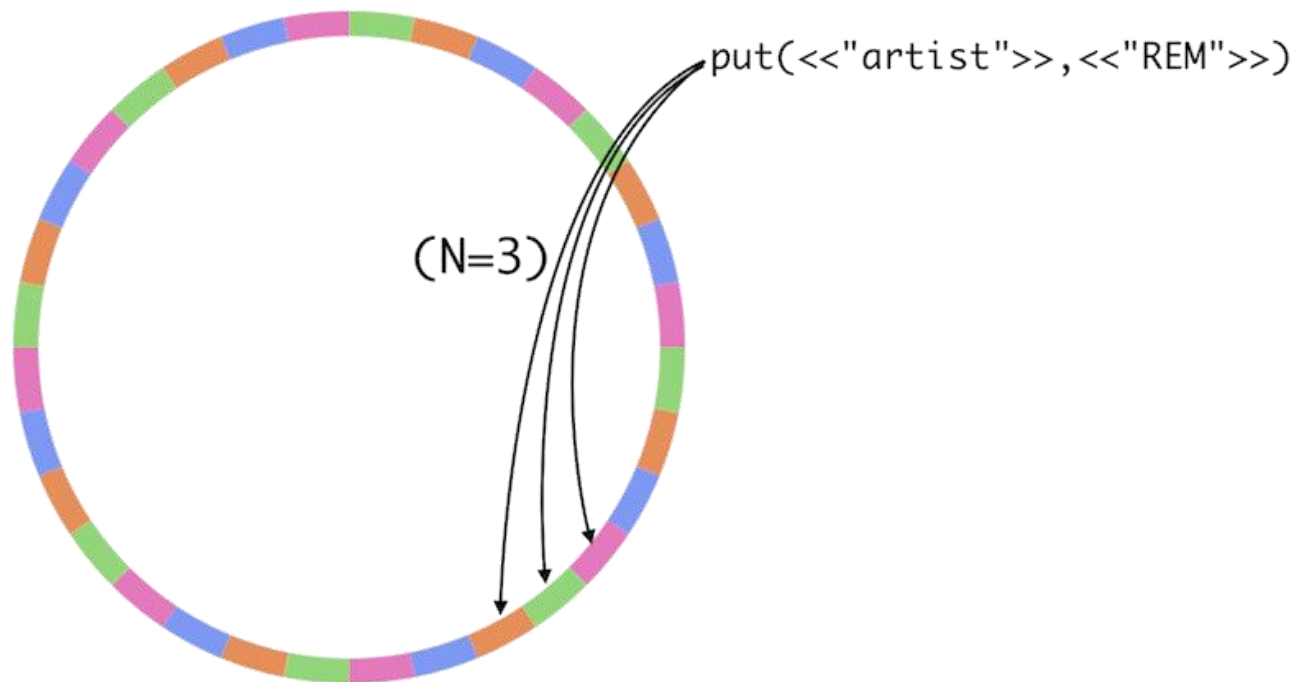
### ■ Gossip protocol

- To share and communicate ring state and bucket properties around the cluster
- Each node „gossips“:
  - Whenever it changes its claim on the ring
    - Announces its change
  - Periodically sends its current view of the ring state
    - To a randomly-selected peer
    - For the case a node missed previous updates

# Key-value store

## Replication in Riak

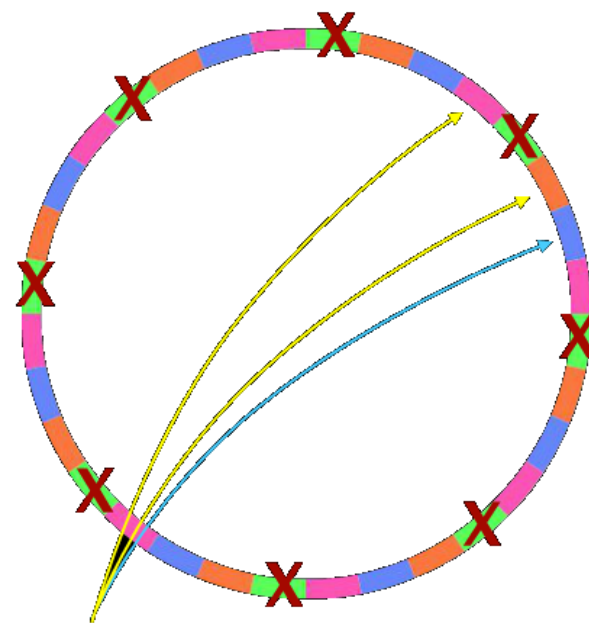
- Setting called **N value**
  - Default: N=3
- Riak objects inherit the N value from their bucket



# Key-value store

## Replication in Riak

- Riak's key feature: high availability
- Hinted handoff
  1. Node failure
  2. Neighboring nodes temporarily take over storage operations
  3. When the failed node returns, the updates received by the neighboring nodes are handed off to it

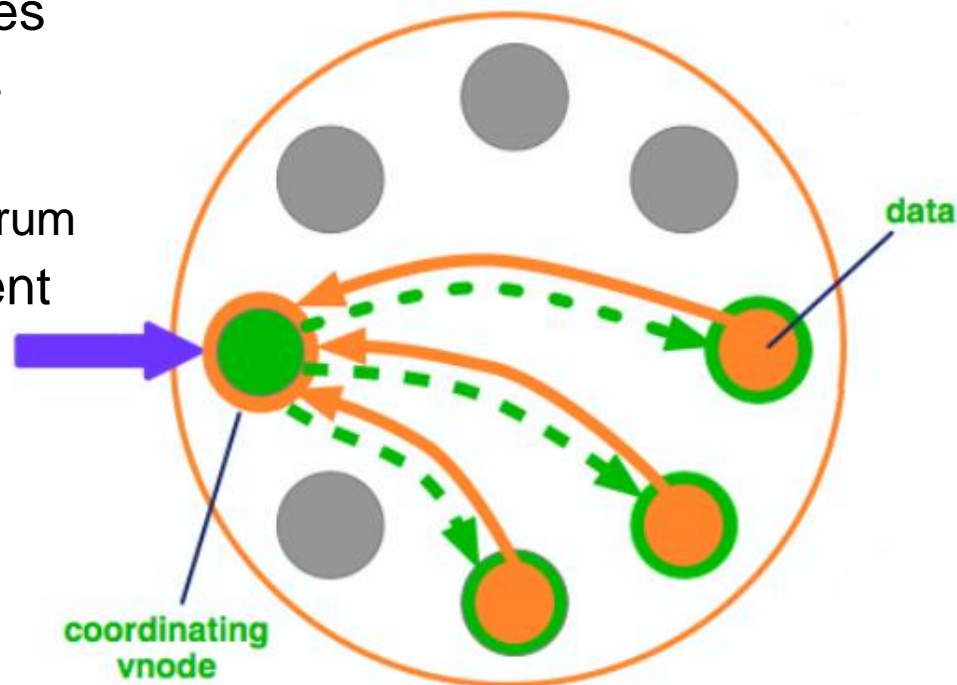


```
put(<<"artist">>, <<"REM">>)
```

# Key-value store

## Riak Request Anatomy

- Each node can be a **coordinating vnode** = node responsible for a request
  1. Finds the vnode for the key according to hash
  2. Finds vnodes where other replicas are stored – next N-1 nodes
  3. Sends a request to all vnodes
  4. Waits until enough requests returned the data
    - To fulfill the read/write quorum
  5. Returns the result to the client



# Key-value store

## Riak Vector Clocks

### ■ Problem:

- Any node is able to receive any request
- Not all nodes need to participate in each request

→ We need to know which version of a value is current

non human  
readable

a85hYGBgzGDKBVlcR4M2cgcZH7HPYEpkzGNIsP/VfYYvCwA=

### ■ When a value is stored in Riak, it is tagged with a **vector clock**

- A part of object's header

### ■ For each update it is updated to determine:

- Whether one object is a direct descendant of the other
- Whether the objects are direct descendants of a common parent
- Whether the objects are unrelated in recent heritage

# Key-value store

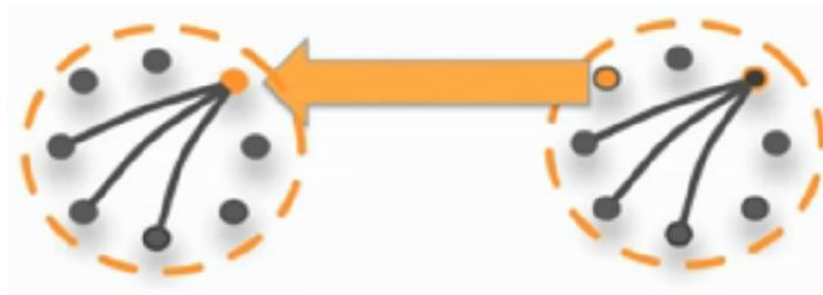
## Riak Siblings

- **Siblings** = multiple objects in a single key
  - To have different values on different nodes
  - Allowed by `allow_mult = true` setting of a bucket
- Siblings of objects are created in case of:
  - Concurrent writes – two writes occur simultaneously from two clients
  - Stale vector clock – write from a client with an old vector clock value
    - It was changed in the mean time by another node
  - Missing vector clock – write without a vector clock
- When retrieving an object we can:
  - Retrieve just the list of siblings (their **V-tags** = IDs)
  - Retrieve all siblings
  - Resolve the inconsistency
    - When `allow_mult = false` Riak resolves internally
      - timestamp-based, last-write-wins (using vector clocks), ...

Less  
probable,  
but can  
occur

# Key-value store

## Riak Enterprise



- Commercial extension of Riak
- Adds support for:
  - Multi-datacenter replication
    - Using more clusters and replication between them
    - Real-time replication – incremental synchronization
    - Full-sync replication – entire data set is synchronized
  - SNMP (Simple Network Management Protocol) monitoring
    - A built-in SNMP server
      - Allows an external system to query the Riak node for statistics
        - E.g., average get / put times, number of puts / gets...
  - JMX (Java Management Extensions) monitoring
    - Java technology for managing and monitoring applications
    - Resources represented as objects
    - Classes can be dynamically loaded and instantiated

# REDIS





# Key-value store



## Redis

- Open-source database
  - First release: 2009
  - Development sponsored by VMware
- OS: most POSIX systems like Linux, \*BSD, OS X, ...
  - Win32-64 experimental version
- Language: ANSI C
  - Clients in many languages: C, PHP, Java, Ruby, Perl, ...
- Not standard key-value features (rather a kind of document database):
  - Keys are binary safe = any binary sequence can be a key
  - The stored value can be any object → “data structure server”
    - strings, hashes, lists, sets and sorted sets
  - Can do range, diff, union, intersection, ... operations
    - Atomic operations
    - Not usual, not required for key-value stores

# Key-value store

## Redis



### ■ In-Memory Data Set

- Good performance
  - For datasets not larger than memory → distribution
- Persistence: dumping the dataset to disk periodically / appending each command to a log

### ■ Pipelining

- Allows to send multiple commands to the server without waiting for the replies + finally read the replies in a single step

### ■ Publish/subscribe

- Published messages are sent into channels and subscribers express interest in one or more channels
- e.g., one user subscribes to a channel
  - e.g., `subscribe warnings`another sends messages
  - e.g., `publish warnings "it's over 9000!"`

### ■ Cache-like behavior

- Key can have assigned a time to live, then it is deleted

# Redis Cache-like Behaviour

## Example

```
> SET cookie:google hello
OK
> EXPIRE cookie:google 30
(integer) 1
> TTL cookie:google           // time to live
(integer) 23
> GET cookie:google           // still some time to live
„hello“
> TTL cookie:google           // key has expired
(integer) -1
> GET cookie:google           // and was deleted
(nil)
```

# Redis Data Types

## Strings

- Binary safe = any binary sequence
  - e.g., a JPEG image
- Max length: 512 MB
- Operations:
  - Set/get the string value of a key: `GET/SET`, `SETNX` (set if not set yet)
  - String-operation: `APPEND`, `STRLEN`, `GETRANGE` (get a substring), `SETRANGE` (change a substring)
  - Integer-operation: `INCR`, `INCRBY`, `DECR`, `DECRBY`
    - When the stored value can be interpreted as an integer
  - Bit-operation: `GETBIT`, `BITCOUNT`, `SETBIT`

# Redis Data Types

## Strings – Example

```
> SET count 10
```

```
OK
```

```
> GET count
```

```
„10“
```

```
> INCR count
```

```
(integer) 11
```

```
> DECRBY count 10
```

```
(integer) 1
```

```
> DEL count
```

```
(integer) 1
```

```
// returns the number of keys removed
```

# Redis Data Types

## List

- Lists of strings, sorted by insertion order
- Possible to push new elements on the head (on the left) or on the tail (on the right)
- A key is removed from the key space if a list operation will empty the list (= value for the key)
- Max length:  $2^{32} - 1$  elements
  - 4,294,967,295 = more than 4 billion of elements per list
- Accessing elements
  - Very fast near the extremes of the list (head, tail)
  - Slow accessing the middle of a very big list
    - $O(N)$  operation

# Redis Data Types

## List

### ■ Operations:

- Add element(s) to the list:
  - **LPUSH** (to the head)
  - **R PUSH** (to the tail)
  - **LINSERT** (inserts before or after a specified element)
  - **LPUSHX** (push only if the list exists, do not create if not)
- Remove element(s): **LPOP**, **RPOP**, **LREM** (remove elements specified by a value)
- **LRANGE** (get a range of elements), **LLEN** (get length), **LINDEX** (get an element at index)
- **BLPOP**, **BRPOP** remove an element or block until one is available
  - Blocking version of LPOP/RPOP

# Redis Data Types

## List – Example

```
> LPUSH animals dog
(integer) 1      // number of elements in the list
> LPUSH animals cat
(integer) 2
> RPUSH animals horse
(integer) 3
> LRANGE animals 0 -1 // -1 = the end
1) „cat“
2) „dog“
3) „horse“
> RPOP animals
„horse“
> LLEN animals
(integer) 2
```



# Redis Data Types

## Set

- Unordered collection of non-repeating strings
- Possible to add, remove, and test for existence of members in  $O(1)$
- Max number of members:  $2^{32} - 1$
- Operations:
  - Add element: `SADD`, remove element: `SREM`
  - Classical set operations: `SISMEMBER`, `SDIFF`, `SUNION`, `SINTER`
  - The result of a set operation can be stored at a specified key (`SDIFFSTORE`, `SINTERSTORE`, ...)
  - `SCARD` (element count), `SMEMBER` (get all elements)
  - Operations with a random element: `SPOP` (remove and return random element), `SRANDMEMBER` (get a random element)
  - `SMOVE` (move element from one set to another)

# Redis Data Types

## Set – Example

```
> SADD friends:Lisa Anna
(integer) 1
> SADD friends:Dora Anna Lisa
(integer) 2
> SINTER friends:Lisa friends:Dora
1) „Anna“
> SUNION friends:Lisa friends:Dora
1) „Lisa“
2) „Anna“
> SISMEMBER friends:Lisa Dora
(integer) 0
> SREM friends:Dora Lisa
(integer) 1
```

# Redis Data Types

## Sorted Set

- Non-repeating collection of strings
- Every member is associated with a **score**
  - Used in order to make the set ordered
    - From the smallest to the greatest
  - May have repeated values
    - Then lexicographical order
- Possible to add, remove, or update elements in  $O(\log N)$
- Operations:
  - Add element(s): **ZADD**, remove element(s): **ZREM**, increment the score of a member: **ZINCRBY**
  - Number of elements in a set: **ZCARD**
  - Elements with a score in a specified range: **ZCOUNT** (count), **ZRANGEBYSCORE** (get the elements)
  - Set operations (store result at a specified key): **ZINTERSTORE**, **ZUNIONSTORE**, ...

# Redis Data Types

## Sorted Set – Example

```
> ZADD articles 1 Anna 2 John 5 Tom
(integer) 3
> ZCARD articles
(integer) 3
> ZCOUNT articles 3 10 // members with score 3-10
(integer) 1
> ZINCRBY articles 1 John
„3“ // returns new John's score
> ZRANGE articles 0 -1 // outputs all members
1) „Anna“ // sorted according score
2) „John“
3) „Tom“
```

# Redis Data Types

## Hash

- Maps between string fields and string values
- Max number of field-value pairs:  $2^{32} - 1$
- Optimal data type to represent objects
  - e.g., a user with fields name, surname, age, ...
- Operations:
  - **HSET** *key field value* (set a value to the field of a specified key), **HMSET** (set multiple fields)
  - **HGET** (get the value of a hash field), **HMGET**, **HGETALL** (get all fields and values in a hash)
  - **HKEYS** (get all fields), **HVALS** (get all values)
  - **HDEL** (delete one or more hash fields), **HEXISTS**, **HLEN** (number of fields in a hash)

# Redis Data Types

## Hash – Example

```
> HSET users:sara id 3
(integer) 1
> HGET users:sara id
"3"
> HMSET users:sara login sara group students
OK
> HMGET users:sara login id
1) "sara"
2) "3"
> HDEL users:sara group
(integer) 1
> HGETALL users:sara
1) "id"
2) "3"
3) "login"
4) "sara"
```

# Key-value store



## Transactions in Redis

- Every command is atomic
- Support for transactions when using multiple commands
  - The commands will be executed in order
  - The commands will be executed as a single atomic operation
  - Either all or none of the commands in the transaction will be executed

```
> MULTI
```

queue the commands

```
OK
```

```
> INCR foo
```

```
QUEUED
```

```
> INCR bar
```

```
QUEUED
```

```
> EXEC
```

execute the queued commands

```
1) (integer) 1
```

```
2) (integer) 1
```

# Key-value store

## Transactions in Redis

- Two kinds of command errors:
  - A command may fail to be queued
    - An error before EXEC is called
    - e.g., command may be syntactically wrong, out of memory condition, ...
    - Otherwise the command returns QUEUED
  - A command may fail after EXEC is called
    - e.g., an operation against a key with the wrong value (e.g., calling a list operation against a string value)
- Even when a command fails, all the other commands in the queue are processed



# Key-value store

## Transactions in Redis

```
> MULTI
```

```
OK
```

```
> SET a 3
```

```
QUEUED
```

```
> LPOP a
```

```
QUEUED
```

```
> SET a 4
```

```
QUEUED
```

```
> EXEC
```

```
1) OK
```

```
2) WRONGTYPE Operation
```

```
against a key holding the  
wrong kind of value
```

```
3) OK
```

```
> GET a
```

```
"4"
```

```
> SET foo 1
```

```
OK
```

```
> MULTI
```

```
OK
```

```
> INCR foo
```

```
QUEUED
```

```
> DISCARD
```

```
OK
```

```
> GET foo
```

```
"1"
```

# Key-value store

## Redis Replication

### ■ Master-slave replication

- A master can have multiple slaves
- A slave can serve as master for other slaves
  - Can form a graph
- Slaves are able to automatically reconnect when the master-slave link goes down for some reason

### ■ Replication is **non-blocking** on the master side

- Master continues to serve queries when slaves perform synchronization

### ■ Replication is non-blocking on the slave side

- While the slave is performing synchronization, it can reply to queries using the old version of the data
  - Optionally can block if required
- There is a moment where the old dataset must be deleted and the new one must be loaded = **blocking**

# Key-value store

## Redis Synchronization of Replicas

1. Upon (re-)connection to master slave sends SYNC command
2. The master starts background saving
  - Buffers all new commands received that modify the dataset
3. When the background saving is complete, the master transfers the database file to the slave
4. Slave saves it on disk, and then loads it into memory
5. Master sends to the slave also the buffered commands

Full (re-)synchronization

- Since Redis 2.8 **partial synchronization**:
  - In-memory backlog of the replication stream on master side
  - Master and slave agree on replication offset and master run ID
  - Replication starts from the offset if the ID is the same after re-connect



# Key-value store

## Redis Partitioning

### ■ Redis Cluster

- Future standard Redis partitioning
- Currently not production-ready (work in progress)
  - Unstable version available

### ■ Twemproxy

- Developed at Twitter
- Suggested way to handle partitioning with Redis
  - An intermediate layer between clients and Redis instances ensuring partitioning
- Supports automatic sharding among multiple Redis instances
- Supports **consistent hashing**

# Key-value store

## Redis High-Availability – Redis Sentinel

- **Redis Sentinel** – a system designed to help managing Redis instances
  - Monitoring: checks if master and slave instances are working
  - Notification: notifies the system via an API if not
  - Automatic failover: If a master is not working as expected, Sentinel can promote a slave to master
    - Other slaves are reconfigured to use the new master
    - Applications using the server are informed about the new address
- Currently a work in progress = still changes a lot
- Distributed system
  - Multiple processes run in the infrastructure
  - Use agreement protocols in order to understand if a master is down and to perform the failover

# Key-value store

## Redis High-Availability – Redis Sentinel Settings

```
sentinel monitor mymaster 127.0.0.1 6379 2
// monitor this server, two sentinels must agree on
// failure
sentinel down-after-milliseconds mymaster 60000
// when a server is considered as failed
sentinel failover-timeout mymaster 900000
// maximum time for failover (to recognize its failure)
sentinel can-failover mymaster yes
// can failover be done?
sentinel parallel-syncs mymaster 1
// number of slaves that can be reconfigured to use
// the new master after a failover at the same time
```

# References

- Eric Redmond – Jim R. Wilson: **Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement**
- Pramod J. Sadalage – Martin Fowler: **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**
- Karl Seguin: **The Little Redis Book**  
<http://openmymind.net/2012/1/23/The-Little-Redis-Book/>
- Eric Redmond: **A Little Riak Book**  
<http://littleriakbook.com/>