Key-value Stores: RiakKV

Big Data Management

Faculty of Computer Science & Information Technology
University of Malaya

October 29, 2019

Borrowed from http://www.ksi.mff.cuni.cz/~svoboda/courses/171-NDBI040/

Lecture Outline

Key-value stores

- Introduction

RiakKV

- Data model
- HTTP interface
- CRUD operations
- Links and Link walking
- Data types
- Search 2.0
- Internal details

Outline

Key-Value Stores

Data model

- Most **simple** NoSQL database type
  - a simple **hash table** (mapping)
- **Key-value pairs**
  - **Key.** (id, identifier, primary key)
  - **Value.** binary object, black box for the database system

Query patterns

- **Create**, **update** or **remove** value for a given key
- **Get** value for a given key

Characteristics

- Simple model ⇒ great **performance**, easily **scaled**, ...
- Simple model ⇒ **not** for **complex queries** nor **complex data**

Key Management

How the keys should actually be designed?

1. **Real-world identifiers**
   - E.g. e-mail addresses, login names, ...
2. **Automatically generated** values
   - Auto-increment integers
     - Not suitable in **peer-to-peer** architectures!
   - Complex keys
     - **Multiple components/combinations** of time stamps, cluster node identifiers, ...
     - Used in **practice**
3. Prefixes describing **entity types** are often used as well
   - E.g. movie_zombieland, movie_inception, ...

Query Patterns

Basic CRUD operations

- Only when a key is provided
- Knowledge of the keys is essential
  - It might even be difficult for a particular database system to provide a list of all the **available keys!**

Accessing the contents of the value part is not possible in general

- But we could **instruct** the database how to parse the values
- ... so that we can index them based on certain **search criteria**

Batch/sequential processing

- MapReduce

Other Functionality

Expiration of key-value pairs

- Objects are **automatically removed** from the database after a **certain interval of time**
- Useful for user sessions, shopping carts etc.

Links between key-value pairs

- Values can be **mutually interconnected** via links
- These links can be **traversed** when querying

Collections of values

- Not only ordinary values can be stored, but also their collections (e.g. ordered lists, unordered sets, ...)

Particular functionality always **depends** on the store we use!

RiakKV Key-value store

Features

- Open source, incremental scalability, high availability, operational simplicity, decentralized design, automatic data distribution, advanced replication, fault tolerance, ...

- `http://basho.com/products/riak-kv/`
- Developed by Basho Technologies
- Implemented in Erlang
  - General-purpose, concurrent, garbage-collected programming language and runtime system
- Operating system: Linux, Mac OS X, ... (not Windows)
- Initial release in 2009

Data Model

Riak database system structure

Instance (→ bucket types) → buckets → objects

1. **Bucket** = collection of objects (logical, not physical collection)
   - Various **properties** are set at the level of buckets
   - E.g. default replication factor, read/write quora, ...
2. **Object = key-value pair**
   - **Key** is a Unicode string
     - **Unique** within a bucket
   - **Value** can be **anything** (text, binary object, image, ...)
   - Each object is also associated with **metadata**
     - E.g. its content type (text/plain, image/jpeg, ...),
     - and other internal metadata as well

Data Model: Design Questions

How buckets and objects should be modeled?

1. Buckets with objects of a **single** **entity type**
   - E.g. one bucket for actors, one for movies, each actor and movie has its own object
2. Buckets with objects of **various** **entity types**
   - E.g. one bucket for both actors and movies, each actor and movie has its **own object** once again
   - **Structured keys** might then help
   - E.g. actor_trojan, movie_medvidek
3. Buckets with **complex objects** containing various data
   - E.g. **one object** for all the actors, one for all the movies

Riak Usage: Querying

Basic CRUD operations

- Create, Read, Update, and Delete
- Based on a key look-up

Extended functionality

- **Links** – relationships between objects and their traversal
- **Search 2.0** – full-text queries accessing values of objects
- **MapReduce**
- ...

Riak Usage: API

Application interfaces

1. **HTTP API**
   - All the user requests are submitted as HTTP requests with appropriately selected/constructed methods, URLs, headers, and data
2. **Protocol Buffers API**
3. **Erlang API**
   - Client libraries for a variety of programming languages
4. **Official.** Java, Ruby, Python, C#, PHP, ...
5. **Community.** C, C++, Haskell, Perl, Python, Scala, ...

Riak Usage: HTTP API

cURL tool

- Allows to transfer data from/to a server using HTTP (or other supported protocols)

Options

1. **-X command, –request command**
   - HTTP request method to be used (GET, ...)
2. **-d data, –data data**
   - Data to be sent to the server (implies the POST method)
3. **-H header, –header header**
   - Extra headers to be included when sending the request
4. **-i, –include**
   - Prints both headers and (not just) body of a response

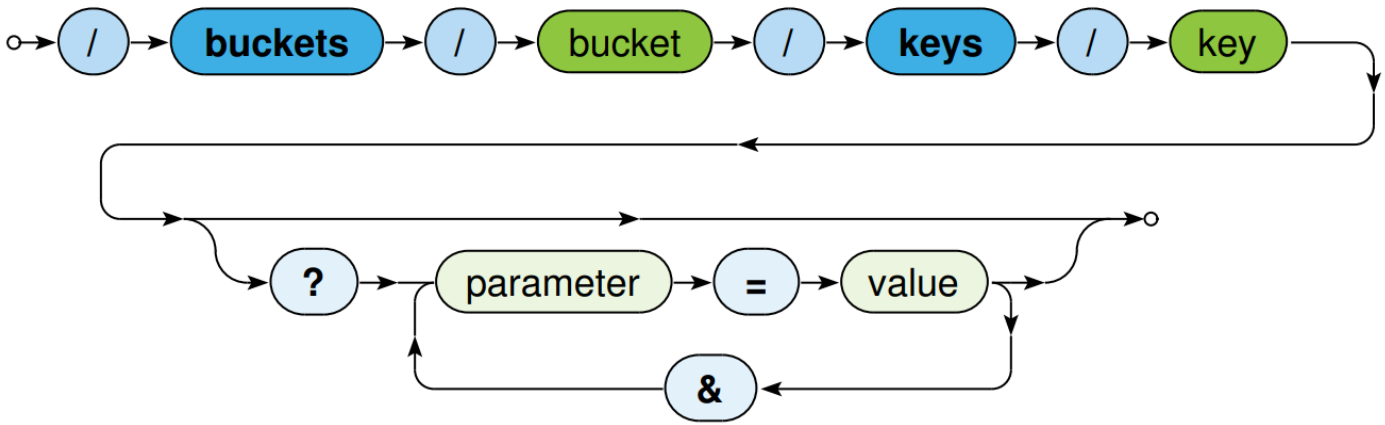
CRUD Operations

Basic operations on objects

1. **Create: POST or PUT methods**
   - Inserts a **key-value pair** into a given bucket
   - Key is specified manually, or will be generated automatically
2. **Read: GET method**
   - Retrieves a key-value pair from a given bucket
3. **Update: PUT method**
   - Updates a key-value pair in a given bucket
4. **Delete: DELETE method**
   - Removes a key-value pair from a given bucket

CRUD Operations

URL pattern of HTTP requests for all the CRUD operations



Optional parameters (depending on the operation)

- r, w: read/write quorum to be attained
- ...

CRUD Operations: Create and Update

Inserts/updates a key-value pair in a given bucket

1. **PUT method**
   - Should be used when a **key is specified explicitly**
   - Transparently inserts/updates (replaces) a given object
2. **POST method**
   - When a key is to be generated **automatically**
   - Always **inserts** a new object

Buckets are created **transparently** whenever needed

Example

```
1 curl -i -X PUT
2 -H 'Content-Type: text/plain'
3 -d 'Woody Herrelson, 1964'
4 http://localhost:8098/buckets/actors/keys/herrelson
```

CRUD Operations: Read

Retrieves a key-value pair from a given bucket

- **Method: GET**

Example

- **Request**

```
curl -i -X GET
http://localhost:8098/buckets/actors/keys/herrelson
```

- **Response**

```
...
Content-Type: text/plain
...
```

```
Woody Herrelson, 1964
```

CRUD Operations: Delete

Removes a key-value pair from a given bucket

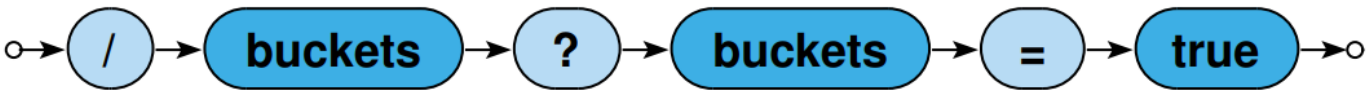
- **Method: DELETE**
- If a given object does **not exist**, it does **not matter**

Example

```
1 curl -i -X DELETE
2 http://localhost:8098/buckets/actors/keys/herrelson
```

Bucket Operations

Lists all the buckets (buckets with at least one object)




```
curl -i -X GET http://localhost:8098/buckets?buckets=true
```

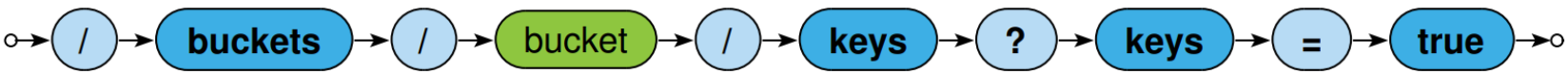
```
Content-Type: application/json
```

```
{
"buckets" : [ "actors", "movies" ]
}
```

Bucket Operations

Lists all the keys within a given bucket

- **Not recommended** to be used in production environments since it is a **expensive** operation




```
curl -i -X GET http://localhost:8098/buckets/actors/keys?keys=true
```

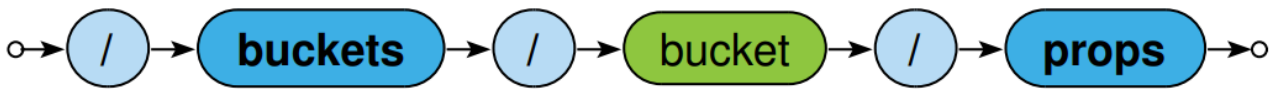
```
Content-Type: application/json
```

```
{
"keys" : [ "herrelson", "eisenberg", "stone", "wasikowska" ]
}
```

Bucket Operations

Setting and **retrieval** of bucket properties

- **Properties**
  - n_val: replication factor
  - r, w, ...: read/write quora and their alternatives
  - ...
- **Requests**
  - **GET/PUT method:** retrieve/set bucket properties



Example

```
1 {
2 "props" : { "n_val" : 3, "w" : "all", "r" : 1 }
3 }
```

Outline

Riak Python client: Prerequisites

Install and use docker

- https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04

Riak quick start with docker

- https://riak.com/posts/technical/riak-quick-start-with-docker/index.html

Riak python client dependencies

1. python-dev — Header files and a static library for Python
2. libffi-dev — Foreign function interface library
3. libssl-dev — libssl and libcrypto development libraries

```
1 sudo apt-get install python-dev libffi-dev libssl-dev
```

Riak Python client: Client Setup

1. Using easy_install

```
easy_install riak
```

2. Using pip

```
pip install riak
```

3. From source

```
python setup.py install
```

Connecting to Riak

Start with

```
1 import riak
```

Create a new client instance

```
1 myClient = riak.RiakClient(pb_port=8087, protocol='pbc')
```

Creating Objects In Riak

Create a bucket

```
1 myBucket = myClient.bucket('test')
```

Insert a few objects

1. Integer 1 with the lookup key of one

```
1 val1 = 1
2 key1 = myBucket.new('one', data=val1)
3 key1.store()
```

2. String value of two with a matching key

```
1 val2 = "two"
2 key2 = myBucket.new('two', data=val2)
3 key2.store()
```

3. Bit of JSON

```
1 val3 = {"myValue": 3}
2 key3 = myBucket.new('three', data=val3)
3 key3.store()
```

Reading Objects From Riak

Request the objects by key

```
1 fetched1 = myBucket.get('one')
2 fetched2 = myBucket.get('two')
3 fetched3 = myBucket.get('three')
4
5 assert val1 == fetched1.data
6 assert val2 == fetched2.data
7 assert val3 == fetched3.data
```

Updating Objects In Riak

Update the value of myValue in the 3rd example to 42.

```
1 fetched3.data["myValue"] = 42
2 fetched3.store()
```

Deleting Objects From Riak

Delete fetched objects

```
1 fetched1.delete()
2 fetched2.delete()
3 fetched3.delete()
```

Verify that the objects have been removed from Riak.

```
1 assert myBucket.get('three').exists == False
```

Working With Complex Objects

Object that encapsulates some knowledge about a book.

```
1 book = {
2   'isbn': "1111979723",
3   'title': "Moby Dick",
4   'author': "Herman Melville",
5   'body': "Call me Ishmael. Some years ago...",
6   'copies_owned': 3
7 }
```

Storing this to Riak:

```
1 booksBucket = myClient.bucket('books')
2 newBook = booksBucket.new(book['isbn'], data=book)
3 newBook.store()
```

How does the Python Riak client encode/decode the object?

Working With Complex Objects

Fetch our book back and print the raw encoded data, we shall know:

```
1 fetchedBook = booksBucket.get(book['isbn'])
2 print(fetchedBook.encoded_data)
```

JSON! The Riak Python client library encodes things as JSON when it can.

```
1 {"body": "Call me Ishmael. Some years ago...",
2 "author": "Herman Melville", "isbn": "1111979723",
3 "copies_owned": 3, "title": "Moby Dick"}
```

To get a deserialized object back use:

```
1 print(fetchedBook.data)
```

Finally, clean up:

```
1 fetchedBook.delete()
```

Examples

Store that object in the key rufus in the bucket dogs, which bears the animals bucket type:

```
bucket = client.bucket_type('animals').bucket('dogs')
obj = RiakObject(client, bucket, 'rufus')
obj.content_type = 'text/plain'
obj.data = 'WOOF!'
obj.store()
```

Store an object under the key viper in the bucket dodge, which bears the type cars, with w set to 3:

```
bucket = client.bucket_type('cars').bucket('dodge')
obj = RiakObject(client, bucket, 'viper')
obj.content_type = 'text/plain'
obj.data = 'vroom'
obj.store(w=3, return_body=True)
```

Examples

Store an object in the bucket random_user_keys, which bears the bucket type users:

```
1 bucket = client.bucket_type('users').bucket('random_user_keys')
2 obj = RiakObject(client, bucket)
3 obj.content_type = 'application/json'
4 obj.data = '{"user":"data"}'
5 obj.store()
6
7 obj.key
```

Attempt a read with r set to 3:

```
1 bucket = client.bucket_type('animals').bucket('dogs')
2 obj = bucket.get('rufus', r=3)
3 print obj.data
```

Examples

Read the object stored there and modify the value:

```
1 bucket = client.bucket_type('sports').bucket('nba')
2 obj = bucket.get('champion')
3 obj.data = 'Harlem Globetrotters'
```

Delete the genius key from the oscar_wilde bucket bearing the type quotes:

```
1 bucket = client.bucket_type('quotes').bucket('oscar_wilde')
2 bucket.delete('genius')
```

For all writes to Riak, specify a content type, e.g. text/plain or application/json (default):

```
1 bucket = client.bucket_type('quotes').bucket('oscar_wilde')
2 obj = RiakObject(client, bucket, 'genius')
3 obj.content_type = 'text/plain'
4 obj.data = 'I have nothing to declare but my genius'
5 obj.store()
```

Outline

Links and Link Walking

Links. are metadata that establish **one-way** relationships between pairs of objects

- Act as **lightweight pointers** between individual key-value pairs
- I.e. represent and extension to the pure key-value data model

Each link...

- is defined within the **source object**
- is associated with a **tag** (sort of link type)
- can be traversed in a given **direction only**
- may connect objects even from **different buckets**

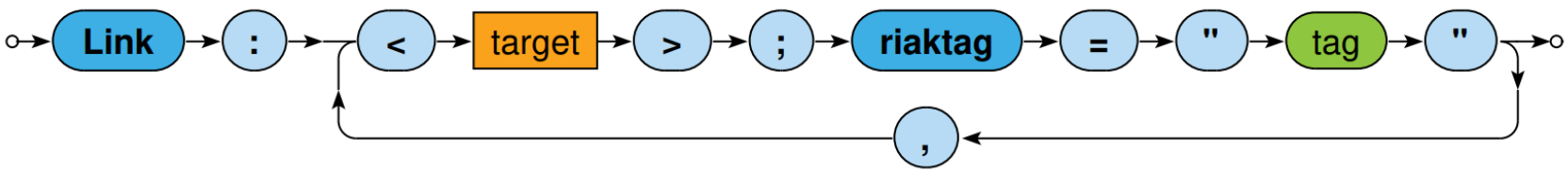
Multiple links can lead from/to a given object

Link walking. New way of querying – navigation between objects using links

Links

How are links defined?

1. **Special** Link header is used for this purpose
2. **Multiple** link headers can be provided, or equivalently multiple links within one header



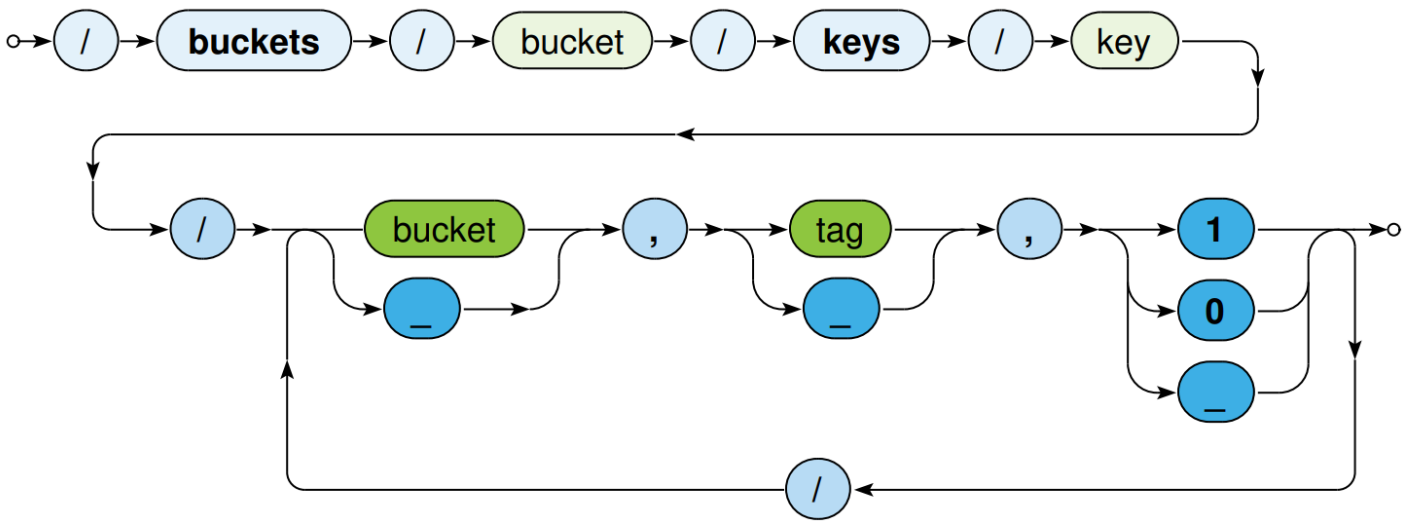
Example

```
curl -i -X PUT
-H 'Content-Type: text/plain'
-H 'Link: </buckets/actors/keys/herrelson>; riaktag="tactor"'
-H 'Link: </buckets/actors/keys/eisenberg>; riaktag="tactor"'
-d 'Zombieland, 2007'
http://localhost:8098/buckets/movies/keys/zombieland
```

Link Walking

How can links be traversed?

- Standard **GET** requests with link **traversal description**
  - Exactly **one** object where the traversal is **initiated**
  - Single or multiple **navigational steps** then follow



Link Walking

Parameters of navigation steps

1. **Bucket**
   - Only objects from a certain **target bucket** are selected
   - _ when **not** limited to any particular bucket
2. **Tag**
   - Only links of a given tag are considered
   - _ when not limited to any particular tag
3. **Keep**
   - 1 when the discovered objects should be included in the result
   - 0 otherwise
   - _ means 1 for the very last step, 0 for all the other preceding

Link Walking

Examples.

- Actors who played in Zombieland movie

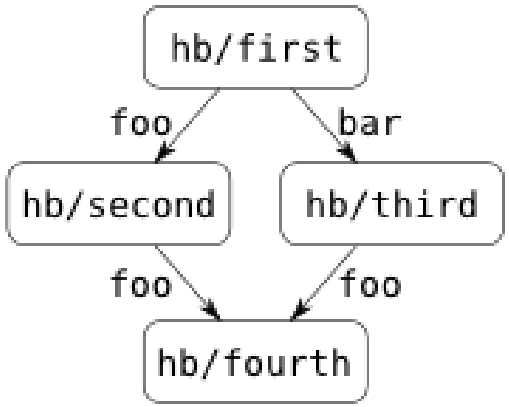
```
curl -i -X GET
http://localhost:8098/buckets/movies/keys/zombieland
/actors,tactor,1
```

- Movies in which appeared actors from Zombieland movie (assuming that the corresponding actor ⇒ movie links also exist)

```
curl -i -X GET
http://localhost:8098/buckets/movies/keys/zombieland
/actors,tactor,0/movies,tmovie,1
```

Link Walking

Example. Consider




```
1 curl http://localhost:8098/riak/hb/first/_,_,_
```

```
1 curl http://localhost:8098/riak/hb/first/_,foo,_
```

```
1 curl http://localhost:8098/riak/hb/first/_,_,_/_,_,_
```

Outline

Data Types: Motivation

Riak began as a **pure** key-value store

- I.e. was completely **agnostic** toward the contents of values

If **availability is preferred** to consistency,

- **mutually conflicting replicas might exist**
- such conflicts can be resolved at the application level,
- but this is often (only too) difficult for the developers

Thus, concept of **Riak Data Types** was introduced

- When used (it is not compulsory),
- Riak is able to **resolve conflicts automatically**

Data Types

Convergent Replicated Data Types (CRDT)

- **Generic** concept
- **Various types** for several common scenarios
- Specific **conflict resolution rules**

Available data types

1. Register, flag
   - Can only be used embedded in maps
2. Counter, set, and map
   - Can be used embedded in maps
   - as well as directly at the bucket level

Data Types

Register

- Allows to store any binary value (e.g. string, ...)
- **Convergence rule:** the most **chronologically recent** value wins

Flag

- **Boolean values:** enable (true), and disable (false)
- **Convergence rule:** **enable wins** over disable

Counter

- **Operations:** increment/decrement by a given integer value
- **Convergence rule:** all requested increments and decrements are **eventually applied**

Data Types

Set

- Collection of **unique binary values**
- **Operations:** addition/removal of one/multiple elements
- **Convergence rule:** addition wins over removal of elements

Map

- Collection of fields with **embedded elements** of any data type (including other nested maps)
- **Operations:** addition/removal of an element
- **Convergence rule:** addition/update wins over removal

Outline

Search 2.0

Riak Search 2.0 (Yokozuna)

- Full-text search over object values
- **Harnesses Apache Solr**
  - Distributed, scalable, failure tolerant, real-time search platform

How does it work?

1. **Indexation**
   - Riak object ⇒ (extractor) ⇒ Solr document ⇒ (schema) ⇒ Solr index
2. **Querying**
   - Riak search query ⇒ Solr search query ⇒ Solr response: list of bucket-key pairs ⇒ Riak response: list of objects

Search 2.0: Extractors

Extractor

- Parses the object value and produces fields to be indexed
- Chosen automatically based on a MIME type

Available extractors

1. Common predefined extractors
   - Plain text, XML, JSON, noop (unknown content type)
2. Built-in extractors for Riak Data Types
   - Counter, map, set
3. User-defined custom extractors
   - Implemented in Erlang, registered with Riak

Search 2.0: Extractors

Plain text extractor (text/plain)

- **Single field** with the **whole content** is extracted

Example

- Input Riak object

```
1 Woody Herrelson, 1964
```

- Output Solr document

```
1 [
2 { text, <<"Woody Herrelson, 1964">> }
3 ]
```

Search 2.0: Extractors

XML extractor (text/xml, application/xml)

- One field is created for **each element and attribute**
- Dot notation is used to compose names of nested items

Example

- Input Riak object

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <actor year="1964">
3 <name>Woody Herrelson</name>
4 <actor>
```

- Output Solr document

```
1 [
2 { <<"actor.name">>, <<"Woody Herrelson">> },
3 { <<"actor.@year">>, <<"1964">> }
4 ]
```

Search 2.0: Extractors

JSON extractor (application/json)

- Similar principles as for XML documents are applied

Example

- Input Riak object

```
{
name : "Woody Herrelson",
year : 1964
}
```

- Output Solr document

```
[
{ <<"name">>, <<"Woody Herrelson">> },
{ <<"year">>, <<"1964">> }
]
```

Search 2.0: Indexation

Solr document

- Automatically **extracted** fields + a few **auxiliary** fields such as:
  - _yz_rb (bucket name), _yz_rk (key), ...

Solr schema

- **Describes** how **fields are indexed** within Solr
  - Values of fields are **analyzed** and **split** into **terms**
  - Terms are **normalized**, stop words **removed**
  - ...
  - **Triples** (token, field, document) are **produced** and **indexed**
- **Default schema** available (_yz_default)
  - Suitable for **debugging**,
  - but **custom** schemas should be used in production

Search 2.0: Index Creation

How is index created?

- Index must be **created** first, then **associated** with a **single bucket**

Example

```
1 curl -i -X PUT
2 -H 'Content-Type: application/json'
3 -d '{ "schema" : "_yz_default" }'
4 http://localhost:8098/search/index/iactors
```

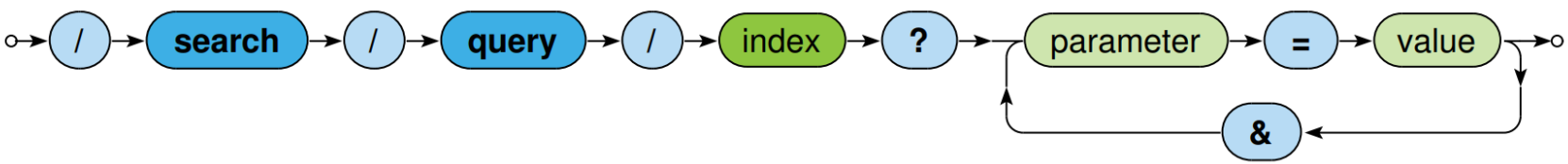
```
1 curl -i -X PUT
2 http://localhost:8098/search/index/iactors
```

```
1 curl -i -X PUT
2 -H 'Content-Type: application/json'
3 -d '{ "props" : { "search_index" : "iactors" } }'
4 http://localhost:8098/buckets/actors/props
```

Search 2.0: Index Usage

Search queries. Parameters

1. **q** – search query (correctly encoded)
   - Individual search criteria
2. **wt** – response write
   - Query result format
3. **start / rows** – pagination of matching objects
4. ...



Search 2.0: Index Usage

Available search functionality

1. **Wildcards**
   - E.g. name:Woo*, name:Wood?
2. **Range queries**
   - E.g. year:[2010 TO *]
3. **Logical connectives and parentheses**
   - AND, OR, NOT
4. **Proximity searches**
   - ...

Example

Create an index called famous using the default schema:

```
client.create_search_index('famous')
```

Explicitly defines the default schema:

```
client.create_search_index('famous', '_yz_default')
```

Populate the cat bucket with values:

```
bucket = client.bucket('cats')
bucket.set_properties({'search_index': 'famous'})

bucket = client.bucket_type('animals').bucket('cats')

cat = bucket.new('liono', {'name_s': 'Lion-o', 'age_i': 30, 'leader_b': True})
cat.store()

cat = bucket.new('cheetara', {'name_s':'Cheetara', 'age_i':28, 'leader_b': True})
cat.store()

cat = bucket.new('snarf', {'name_s':'Snarf', 'age_i':43})
cat.store()

cat = bucket.new('panthro', {'name_s':'Panthro', 'age_i':36})
cat.store()
```

Example

Search for all documents where name_s value begins with Lion by means of a glob (wildcard) match:

```
results = client.fulltext_search('famous', 'name_s:Lion*')
print results
print results['docs']
```

Find the ages of all famous cats who are 30 or older:

```
client.fulltext_search('famous', 'age_i:[30 TO *]')
```

Supports logical conjunctive (AND), disjunctive (OR), and negative (NOT) operations on query elements:

```
client.fulltext_search('famous', 'leader_b:true AND age_i:[30 TO *]')
```

Indexes may be deleted if they have no buckets associated with them:

```
client.delete_search_index('famous')
```

Outline

Architecture

Sharding + peer-to-peer replication architecture

- Any node can serve any **read** or **write** user request
- Physical nodes run (several) virtual nodes (vnodes)
  - Nodes can be **added** and **removed** from the cluster **dynamically**
- **Gossip protocol**
  - Each node **periodically** sends its **current view** of the cluster, its **state** and **changes**, **bucket properties**, ...

CAP properties

- **AP system:** availability + partition tolerance

Consistency

BASE principles

- **Availability** is **preferred** to consistency
- **Default properties** of buckets
  - **n_val:** replication factor
  - **r:** read quorum
  - **w:** write quorum (node participation is sufficient)
  - **dw:** write quorum (write to durable storage is required)
- Specific options of requests override the bucket properties

Strong consistency can be achieved when quora set carefully, i.e.

- w > n_val/2 for write quorum
- r > n_val - w for read quorum

Causal Context

Conflicting replicas are unavoidable (with eventual consistency) ⇒ how are they resolved?

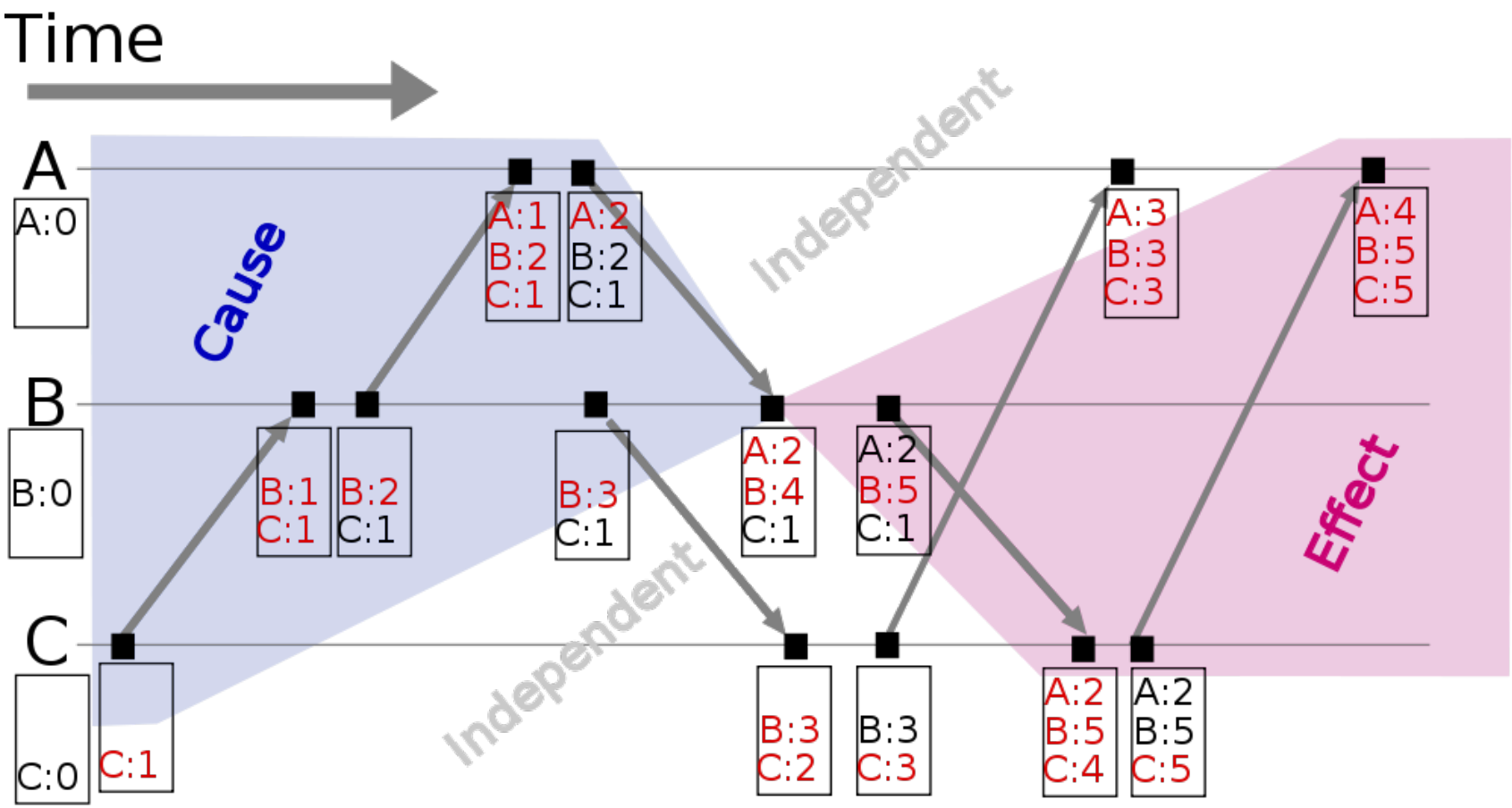
- **Causal context** = auxiliary data and mechanisms that are necessary in order to resolve the conflicts
- **Low-level techniques**
  - Timestamps, vectors clocks, dotted version vectors
  - They can be used to **resolve** conflicts **automatically**
    - Might fail, then we must make the choice by ourselves
  - Or we can resolve the conflicts **manually**
    - **Siblings** then need to be enabled (allow_mult) = **multiple versions** of object values
- **User-friendly CRDT data types** with **built-in resolution**
  - Register, flag, counter, set, map

Causal Context

Vector clocks

- Mechanism for tracking object **update causality** in terms of logical time (not chronological time)
- Each node has its own logical clock (integer counter)
  - **Initially** equal to 0
  - **Incremented** by 1 whenever any event takes place
- **Vector clock** = vector of logical clocks of all the nodes
  - Each node maintains its **local copy** of this vector
  - Whenever a **message is sent**, the local vector is sent as well
  - Whenever a **message is received**, the local vector is updated
    - Maximal value for each individual node clock is taken

Causal Context



Riak Ring

Replica placement strategy

- Consistent hashing function
  - **Consistent** = does not change when cluster changes
  - **Domain.** pairs of a bucket name and object key
  - **Range.** 160-bit integer space = Riak Ring

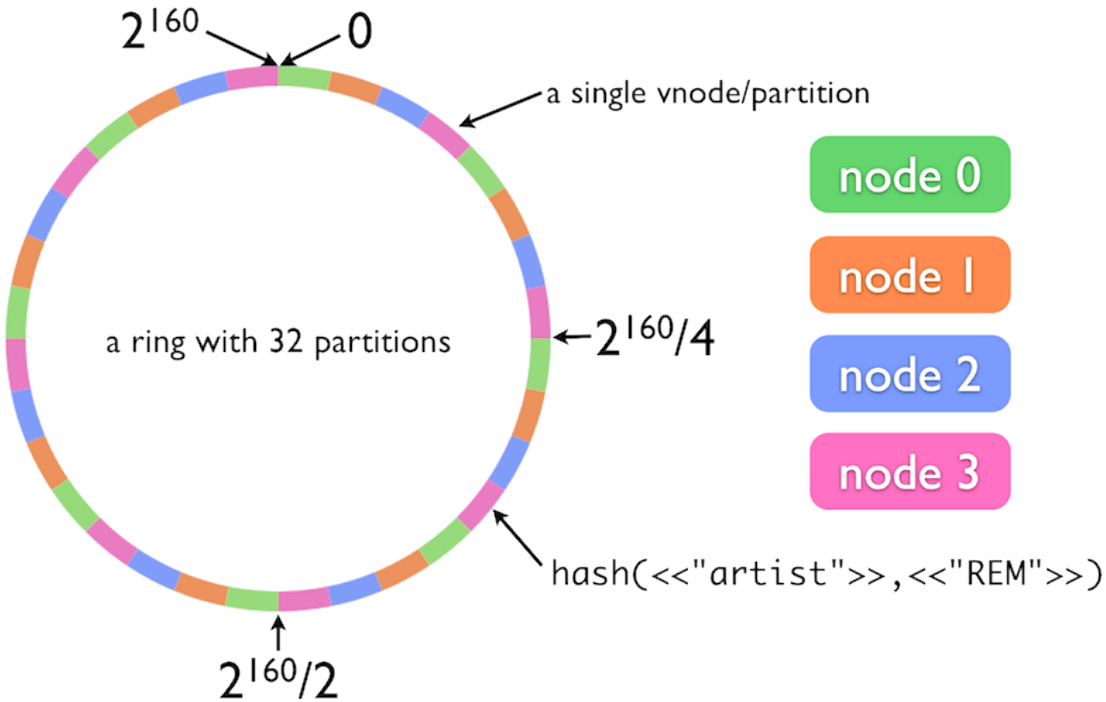
Riak Ring

- The whole ring is split into **equally-sized disjoint partitions**
  - Physical nodes are **mutually interleaved** ⇒ **reshuffling** when cluster changes is **less demanding**
- **Each virtual node** is **responsible** for exactly **one partition**

Riak Ring

Example

- Cluster with 4 physical nodes, each running 8 virtual nodes
- i.e. 32 partitions altogether



Source: `http://docs.basho.com/`

Riak Ring

Replica placement strategy

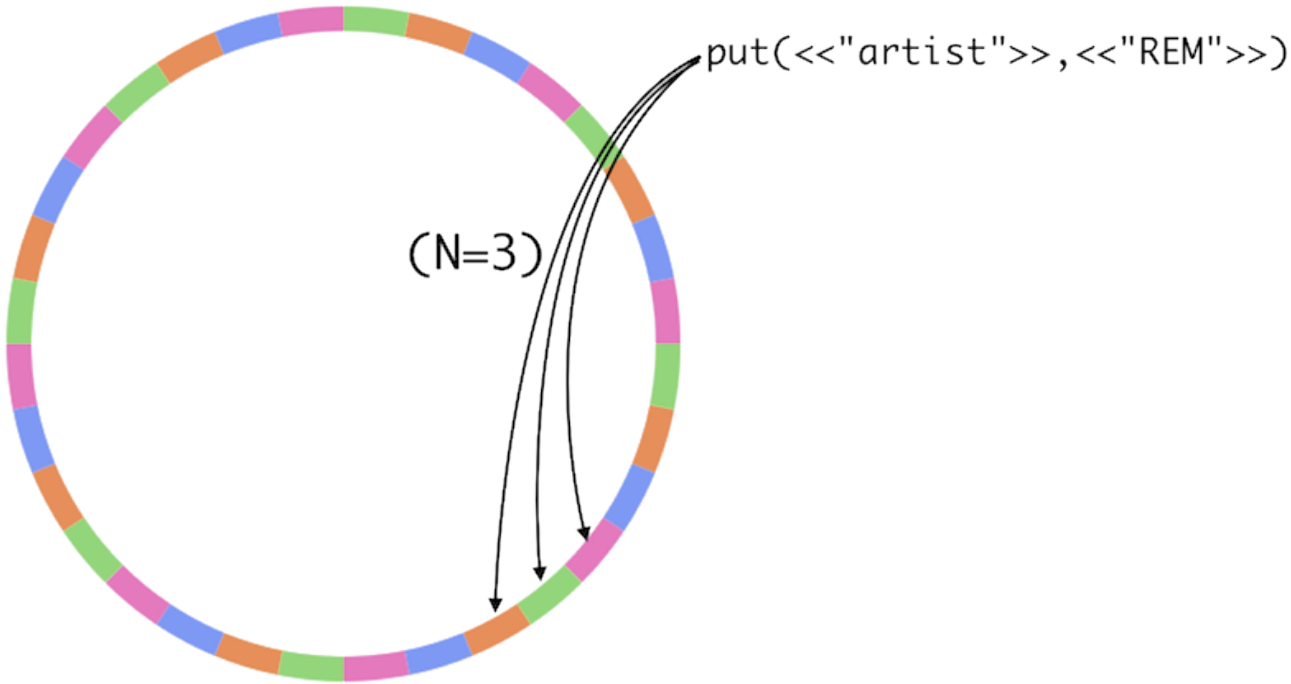
1. The **first** replica...
   - Its location is directly **determined by** the **hash function**
2. All the **remaining** replicas...
   - Placed to the **consecutive partitions** in a **clockwise direction**

What if a virtual node is failing?

- **Hinted handoff**
  - Failing nodes are simply **skipped**, neighboring nodes temporarily take responsibility
  - When resolved, replicas are **handed off** to the proper locations
- **Motivation:** high availability

Riak Ring

Intelligent Replication



Source: `http://docs.basho.com/`

Request Handling

Read and write requests can be submitted to any node

- This nodes is called a **coordinating node**
- **Hash function** is **calculated**, i.e. replica locations determined
- **Internal requests** are **sent** to all the corresponding nodes
- Then the coordinating node **waits** until **sufficient number of responses** is received
- **Result/failure** is **returned** to the user

But what if the cluster changes?

- The value of the **hash** function does **not change**, only the **partitions** and their **mapping** to virtual nodes change
- However, the **Ring knowledge** a given node has might be **obsolete**!

Lecture Conclusion

RiakKV

- Highly available distributed key-value store
- Sharding with peer-to-peer replication architecture
- Riak Ring with consistent hashing for replica placement

Query functionality

- Basic CRUD operations
- Link walking
- Search 2.0 full-text based on Apache Solr

More on: `https://docs.riak.com/riak/kv/2.2.3/index.html`