B4M36DS2, BE4M36DS2: **Database Systems 2**

http://www.ksi.mff.cuni.cz/~svoboda/courses/181-B4M36DS2/

Lecture 7

Key-Value Stores: RiakKV

Martin Svoboda
martin.svoboda@fel.cvut.cz

12. 11. 2018

Charles University, Faculty of Mathematics and Physics

Czech Technical University in Prague, Faculty of Electrical Engineering

Lecture Outline

Key-value stores

- Introduction

RiakKV

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

Key-Value Stores

Data model

- The most simple NoSQL database type
  - Works as 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?

- **Real-world** identifiers
  - E.g. e-mail addresses, login names, …
- **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 instead

Prefixes describing entity types are often used as well

- E.g. `movie_medvidek`, `movie_223123`, …

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!

Riak Key-Value Store

riakKV

RiakKV

Key-value store

- http://basho.com/products/riak-kv/
- Features
  - Open source, incremental scalability, high availability, operational simplicity, decentralized design, automatic data distribution, advanced replication, fault tolerance, …
- 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**

- 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, …
- 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?

- **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
- **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`
- 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

- **HTTP API**
  - All the user requests are submitted as **HTTP requests** with appropriately selected / constructed **methods**, **URLs**, **headers**, and **data**
- Protocol Buffers API
- Erlang API

Client libraries for a variety of programming languages

- Official: Java, Ruby, Python, C#, PHP, …
- 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

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

Basic Operations

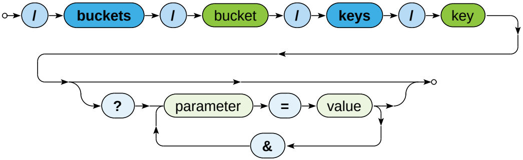
CRUD Operations

Basic operations on objects

- **Create**: `POST` or `PUT` methods
  - **Inserts a key-value pair** into a given bucket
  - Key is specified manually, or will be generated automatically
- **Read**: `GET` method
  - **Retrieves a key-value pair** from a given bucket
- **Update**: `PUT` method
  - **Updates a key-value pair** in a given bucket
- **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

- **PUT** method
  - Should be used when a **key is specified explicitly**
  - Transparently **inserts / updates** (replaces) a given object
- **POST** method
  - When a **key is to be generated automatically**
  - Always **inserts** a new object
- Buckets are created transparently whenever needed

Example

```
curl -i -X PUT
  -H 'Content-Type: text/plain'
  -d 'Ivan Trojan, 1964'
  http://localhost:8098/buckets/actors/keys/trojan
```

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/trojan
```

Response

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

```
Ivan Trojan, 1964
```

CRUD Operations

Delete

Removes a key-value pair from a given bucket

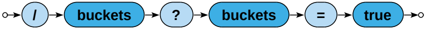
- Method: **DELETE**
- If a given object does not exist, it does not matter

Example

```
curl -i -X DELETE
  http://localhost:8098/buckets/actors/keys/trojan
```

Bucket Operations

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



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

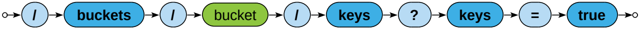
```
Content-Type: application/json
```

```
{
  "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 very expensive operation



/ buckets / bucket / keys ? keys = true

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

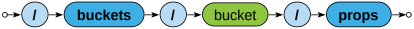
```
Content-Type: application/json
```

```
{
  "keys" : [ "trojan", "machacek", "schneiderova", "sverak" ]
}
```

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

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

Links and Link Walking

Links and Link Walking

Links

- 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

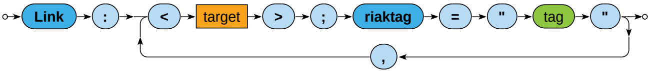
Link walking

- New way of querying – navigation between objects using links

Links

How are links defined?

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



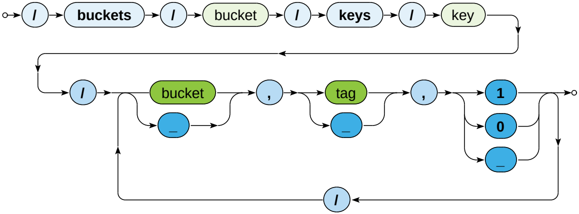
Example

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

Link Walking

How can links be traversed?

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



Link Walking

Parameters of navigation steps

- $Bucket$
  - Only objects from a certain **target bucket** are selected
  - _ when not limited to any particular bucket
- Tag
  - Only links of a given **tag** are considered
  - _ when not limited to any particular tag
- $Keep$
  - 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 *Medvídek* movie

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

```
Content-Type: multipart/mixed; boundary=...
```

Movies in which appeared actors from *Medvídek* movie

(assuming that the corresponding actor → movie links also exist)

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

Data Types

Data Types

Motivation

- Riak began as a **pure key-value store**
    - I.e. was completely agnostic toward the contents of values
- However, 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
- And so the 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** (convergence rules)

Available **data types**

- Register, flag
  - Can only be used embedded in maps
- 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**

Search 2.0

Search 2.0

Riak **Search 2.0** (Yokozuna)

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

How does it work?

- Indexation
  - **Riak object** $\xrightarrow{\text{extractor}}$ **Solr document** $\xrightarrow{\text{schema}}$ **Solr index**
- 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

- **Common predefined extractors**
  - Plain text, XML, JSON, *noop* (unknown content type)
- **Built-in extractors for Riak Data Types**
  - Counter, map, set
- **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

```
Ivan Trojan, 1964
```

Output Solr document

```
[
  { text, <<"Ivan Trojan, 1964">> }
]
```

Search 2.0: Extractors

XML extractor (`text/xml`, `application/xml`)

- One field is created for each element and attribute
  - Only fields with type suffixes are considered
  - E.g. `_s` for string, `_i` for integer, `_b` for boolean, …
  - Dot notation is used to compose flatten names of nested items

Example

Input Riak object / Output Solr document

```
<?xml version="1.0" encoding="UTF-8" ?>
<actor year_i="1964">
  <name_s>Ivan Trojan</name_s>
<actor>
```

```
[
  { <<"actor.name_s">>, <<"Ivan Trojan">> },
  { <<"actor.@year_i">>, <<"1964">> }
]
```

Search 2.0: Extractors

JSON extractor (`application/json`)

- Similar principles as for XML documents are applied

Example

Input Riak object

```
{
  name_s : "Ivan Trojan",
  year_i : 1964
}
```

Output Solr document

```
[
  { <<"name_s">>, <<"Ivan Trojan">> },
  { <<"year_i">>, <<"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

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

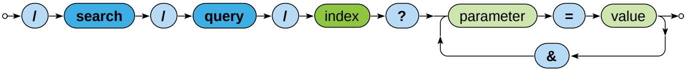
```
curl -i -X PUT
  http://localhost:8098/search/index/iactors
```

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

Search 2.0: Index Usage

Search queries

- Parameters
  - `q` – **search query** (correctly encoded)
    - Individual search criteria
  - `wt` – response write
    - Query result format
  - `start` / `rows` – **pagination** of matching objects
  - …



Search 2.0: Index Usage

Available search functionality

- **Wildcards**
  - E.g. `name:Iva*`, `name:Iva?`
- **Range queries**
  - E.g. `year:[2010 TO *]`
- **Logical connectives** and parentheses
  - `AND`, `OR`, `NOT`
- **Proximity searches**
- …

Internal Details

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.:
  - $\texttt{w} > \texttt{n\_val}/2$ for write quorum
  - $\texttt{r} > \texttt{n\_val} - \texttt{w}$ for read quorum

Causal Context

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

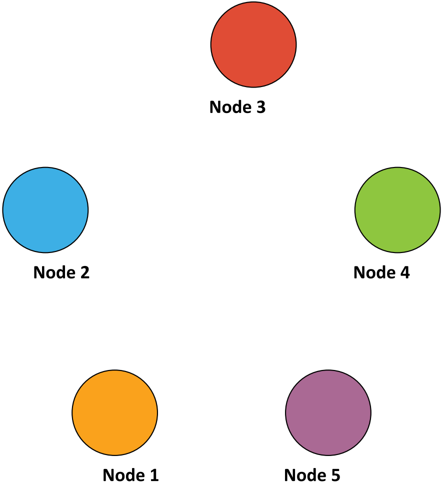
- Causal context = auxiliary data and mechanisms that are necessary in order to resolve the conflicts
- **Low-level techniques**
  - Timestamps, vectors clocks, dotted version vectors
  - They can be used to resolve conflicts **automatically**
    - Might fail, then we must make the choice by ourselves
  - 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

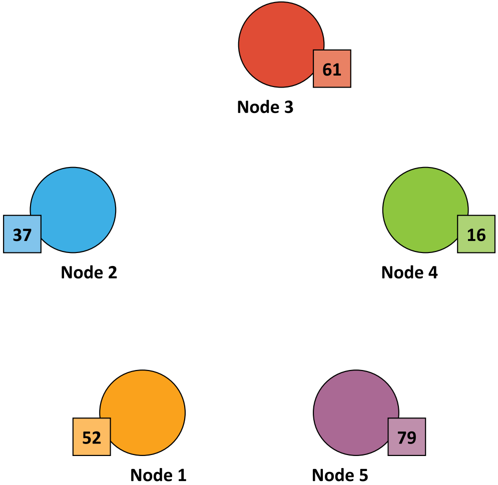
Vector clocks

- 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

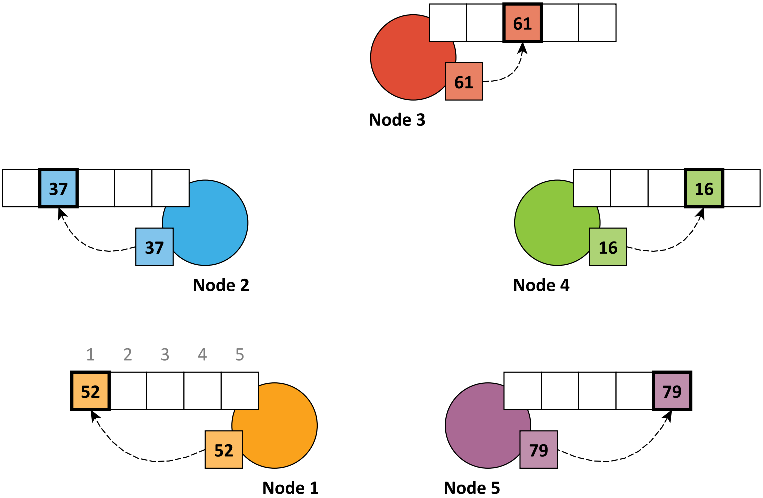
Vector Clocks



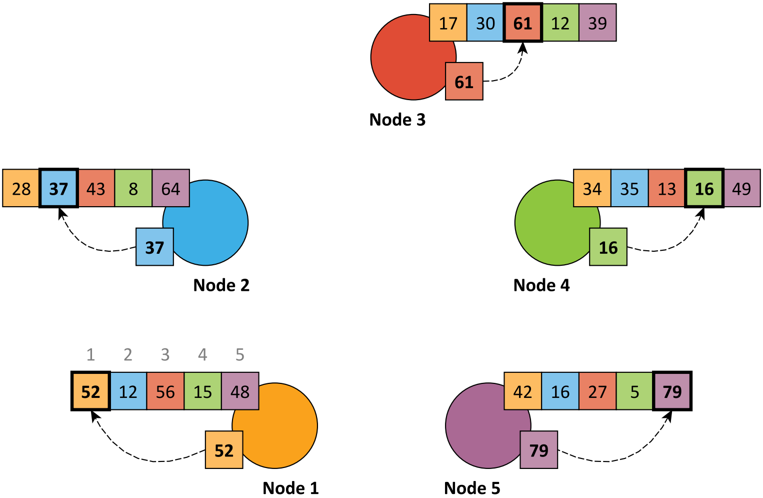
Vector Clocks



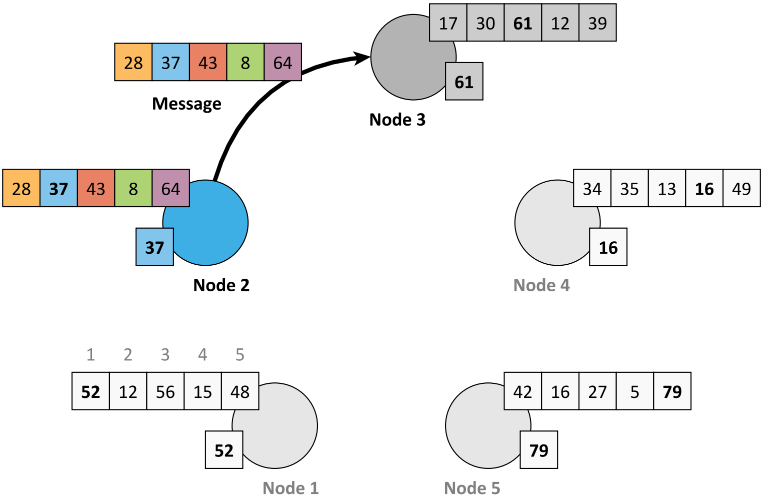
Vector Clocks



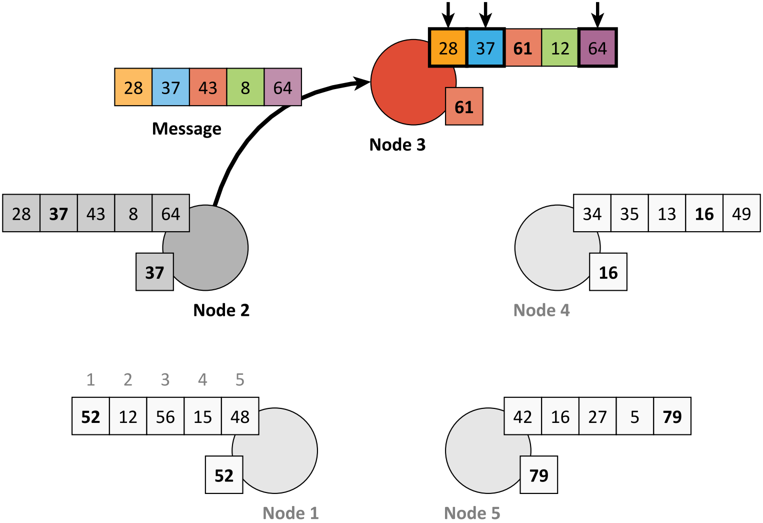
Vector Clocks



Vector Clocks



Vector Clocks



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

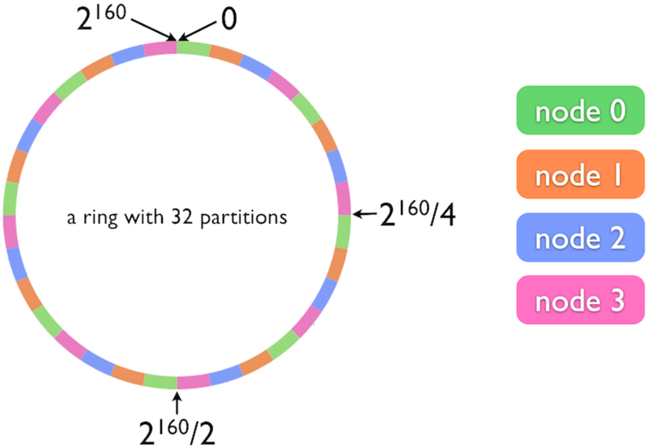
Riak Ring

- 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**

Example

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

Riak Ring



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

Riak Ring

Replica placement strategy

- The first replica…
  - Its location is **directly determined by the hash function**
- 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

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