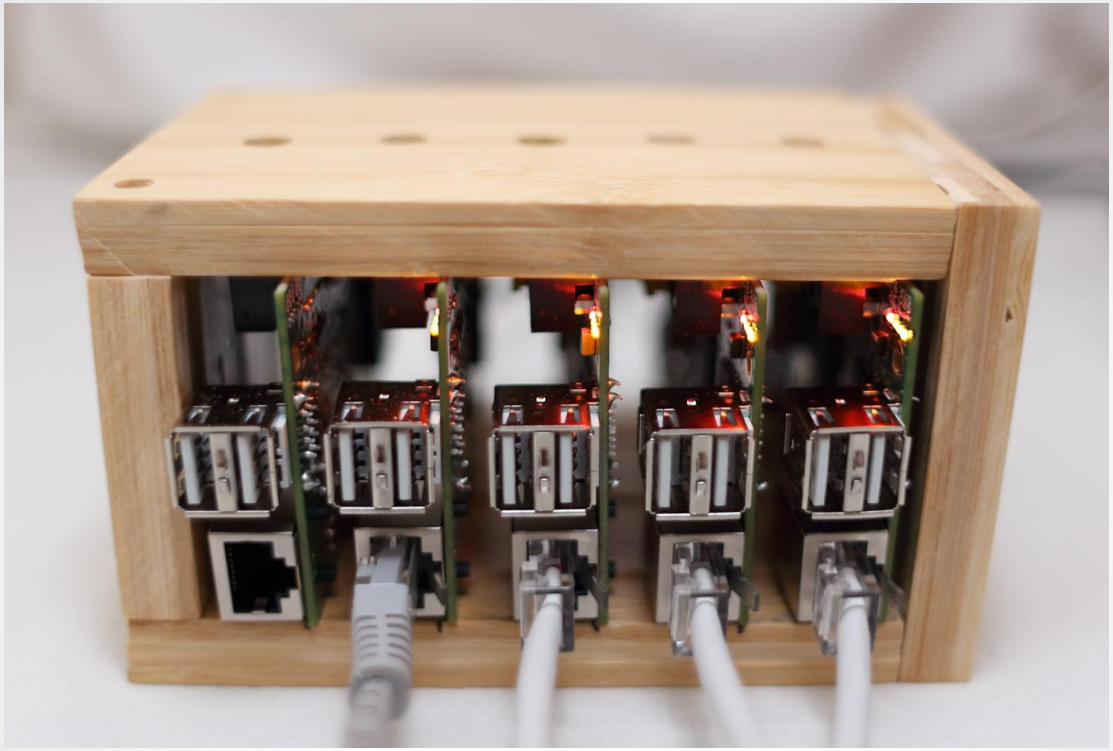
Paweł Sidoryk

RIAK

The most powerful open-source, distributed database you'll ever put into production

Raspberry PI Cluster running Riak



- 4 nodes running Riak, fully operational
- Energy efficient: max 3.5 watts per node
- Based on the **ARM** architecture
- Very cheap
- Not a production grade cluster :-)

The Agenda

1. Riak: basic facts, main features
2. Objects and buckets
3. A demo: a classic distributed shopping cart
4. Data distribution
5. CAP theorem
6. Concurrency control: vector clocks, siblings
7. Tunable consistency
8. Operations on Riak with REST
9. Secondary Indexes (2i)
10. MapReduce in Riak
11. Riak Search
12. Riak Java Client
13. Conflict resolution with Java
14. Convergent Replicated Data Types
15. Wrap-up
16. Questions

Riak: Basic Facts

- A distributed key-value NoSQL database
- Based on Amazon Dynamo
- Fault tolerant
- No single point of failure
- Peer-to-peer architecture
- Easily scalable
- Highly available
- Eventually consistent (tunable consistency)
- Written in Erlang with elements in C and JavaScript

Riak: main features

- Data distribution and replication
- Interfaces: REST-ful API, Protocol Buffers
- Client libraries: Java, Python, Perl, Erlang, Ruby, PHP, .NET
- MapReduce
- Riak Search (full text search)
- Secondary Indexes

Objects in Riak

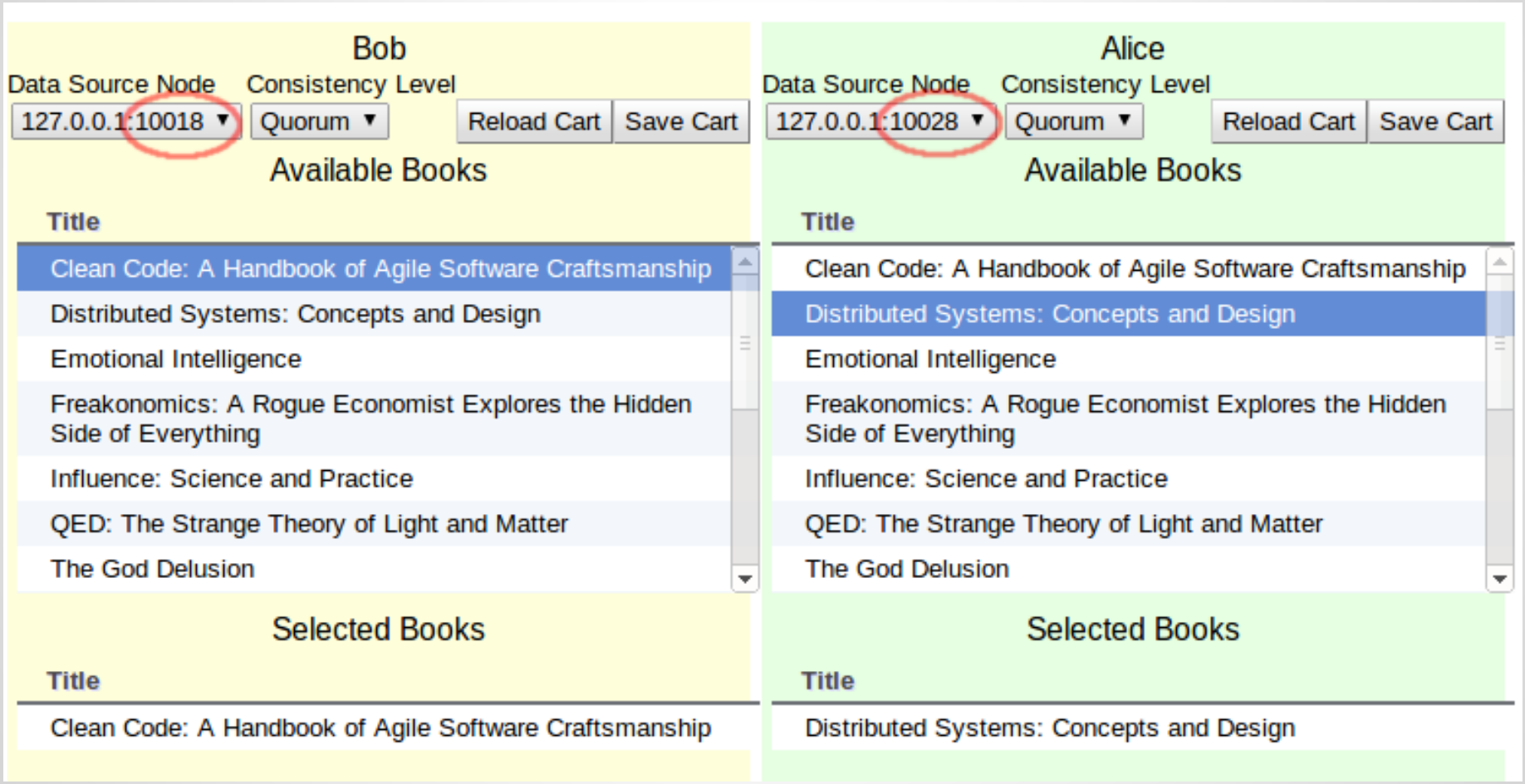
Riak stores values as opaque **binary objects**.

The most convenient way of storing objects: **JSON**-encoded objects, e.g. we have a "book" object:

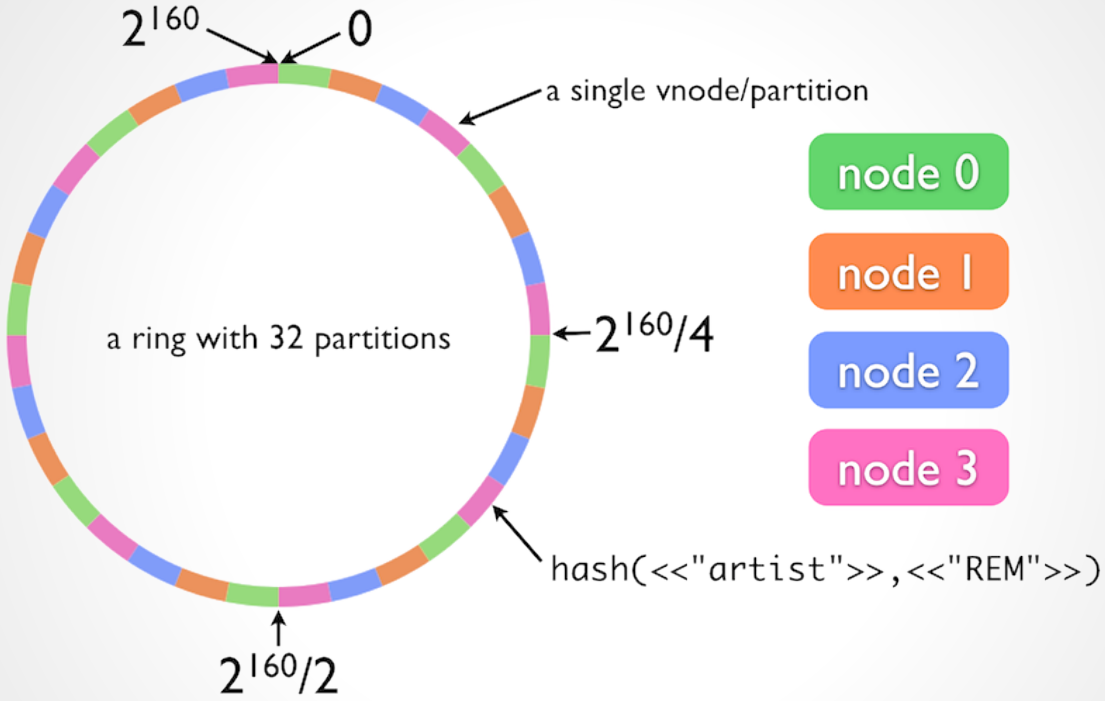
```
{
    "title": "The Prince",
    "author": "Niccolo Machiavelli"
}
```

Objects are grouped into **buckets**. Objects in a bucket share replication level, consistency parameters, storage backend. Objects in a bucket do not have to be related to themselves in any other way.

Demo: Distributed Shopping Cart

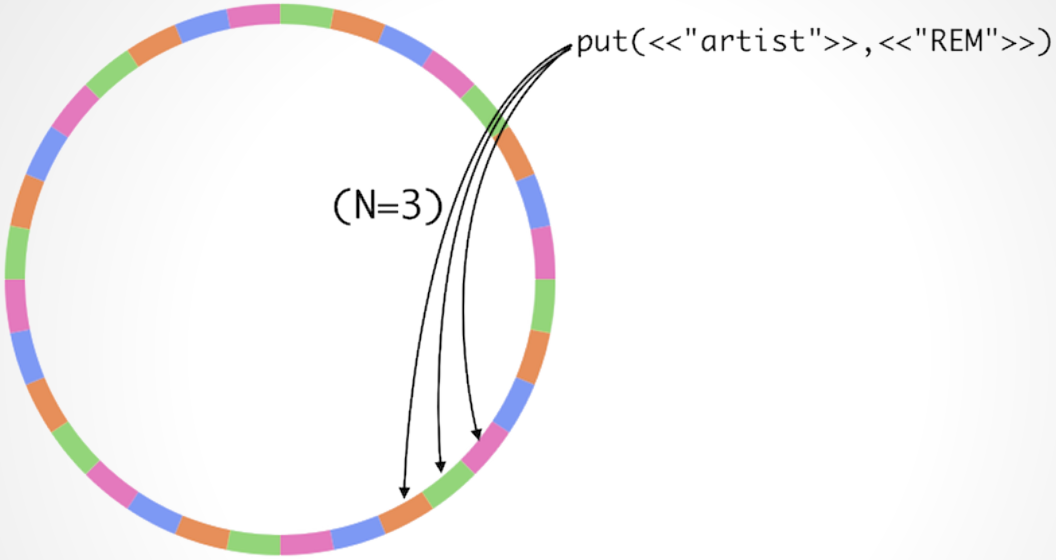


Riak: Data Distribution



Key space is divided into partitions (vnodes) and forms a "ring".
Partitions are distributed over Riak nodes.
There are 32 partitions and 4 nodes in the example above

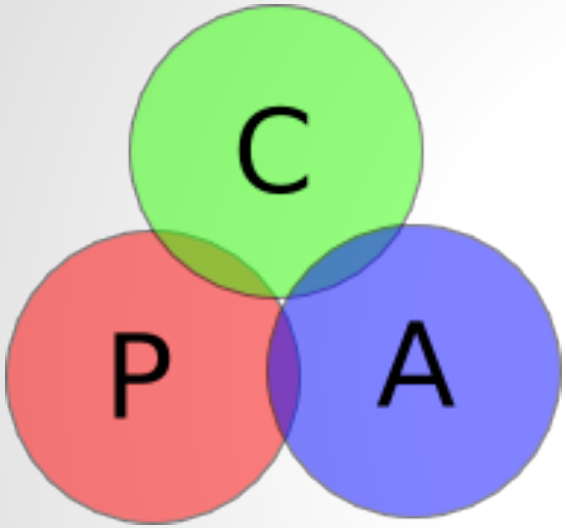
Riak: Data Replication



Each object is replicated to N vnodes (by default N = 3).

When a given node is unavailable then an object is saved to next available node (a fallback node). This is called a "hinted hand-off".

CAP Theorem



Consistency

Availability

Partition Tolerance

- We have to give up **one** of the properties: C, A or P
- Father of the CAP Theorem: Dr. Eric Brewer, Basho board member

CAP in Riak

- Availability and Partition Tolerance guaranteed
- Eventually consistent
- Tunable consistency
- Strong consistency can be easily achieved:

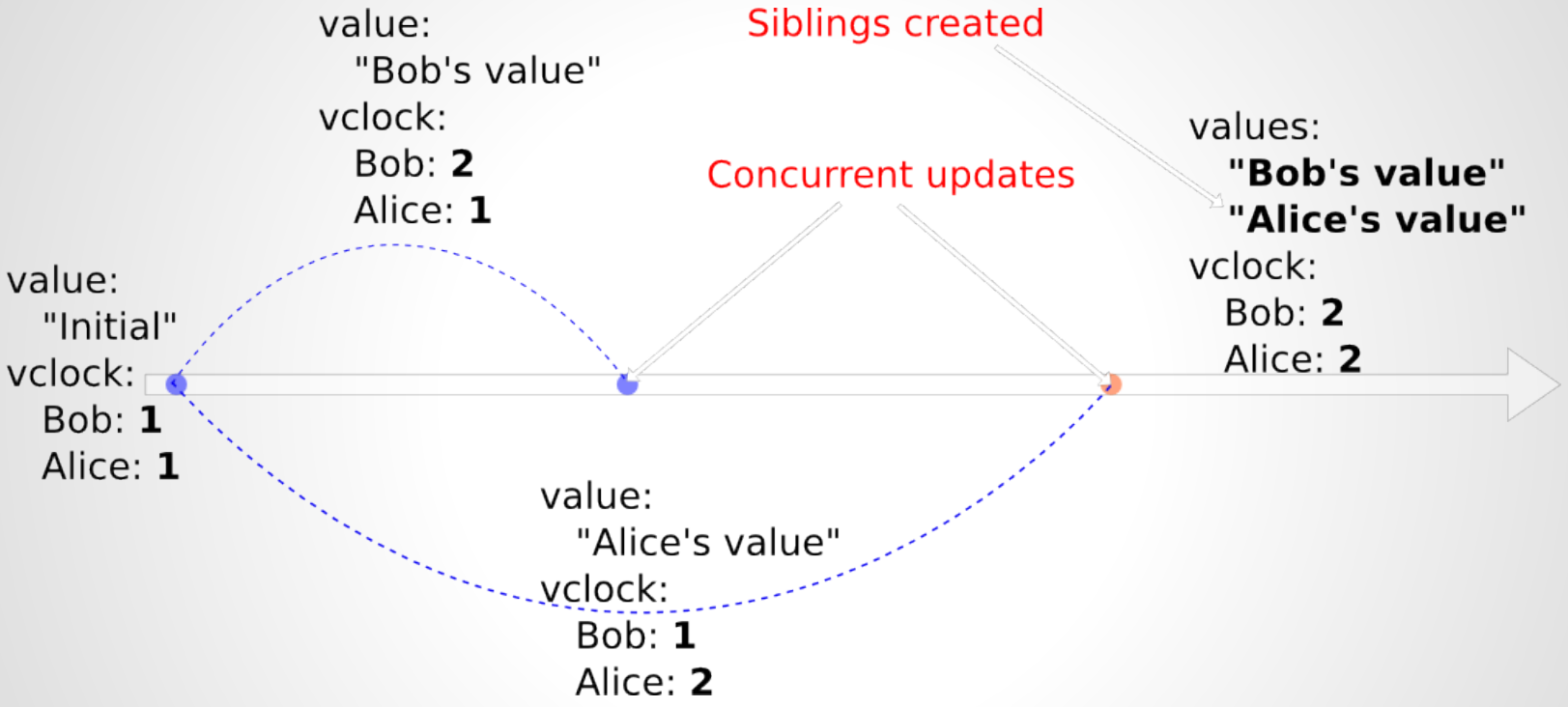
R + W > N

N - number of replicas

R - how many replicas are used for reading

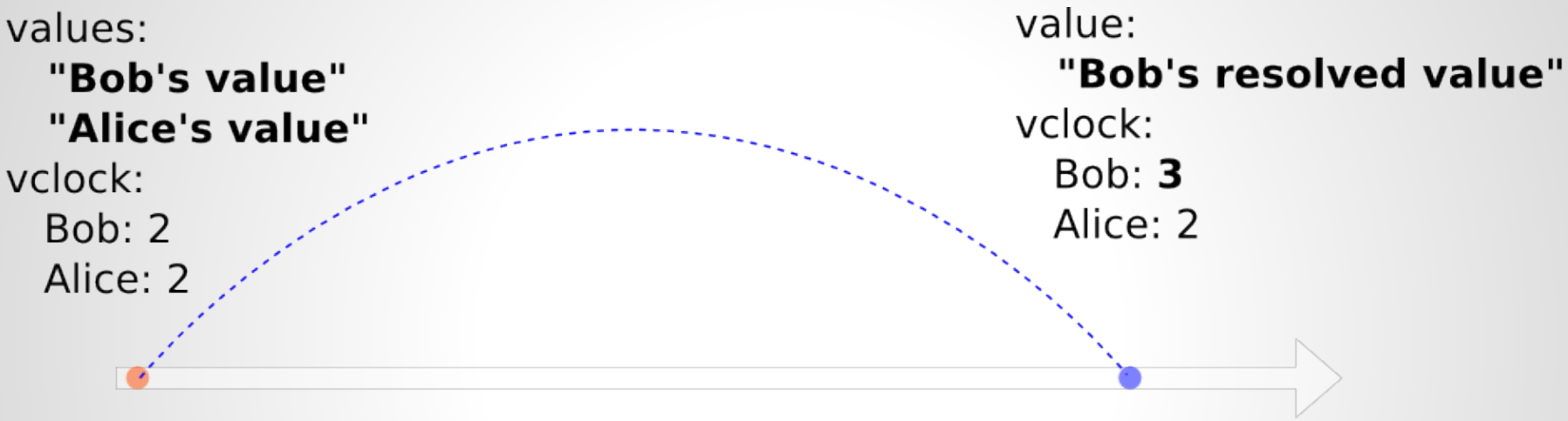
W - how many replicas are used for writing

Vector Clocks in Riak



- **Concurrent operations**: when we cannot order them sequentially
- In Riak: Bob = node1, Alice = node2

Sibling Resolution



- Siblings are detected **while reading**
- Best general strategy to resolve siblings: **merge**
- Dedicated data types optimal for merging: **C**onvergent **R**eplicated **D**ata **T**ypes

More on Siblings

If we want to control **concurrent operations** with siblings then we have to set **allow_mult = true** at the bucket level (it is turned off by default).

Siblings are created when:

- an update on an object is done in parallel
- a stale vector clock is used during an update
- a network partition occurs, object is updated in separate partitions and network partitions reconnect
- when we use a cluster-to-cluster replication

Riak: CAP Controls

CAP controls = query parameters

- **r** - (read quorum) how many replicas need to agree when retrieving the object
- **pr** - (primary read quorum) how many replicas to commit to **primary** nodes before returning a successful response
- **w** - (write quorum) how many replicas to write to before returning a successful response
- **dw** - (durable write quorum) how many replicas to commit to durable storage before returning a successful response
- **pw** - (primary write quorum) how many replicas to commit to **primary** nodes before returning a successful response
- **rw** - how many replicas need to agree for both operations (get and put) involved in deleting an object

Riak: Tunable Consistency

Possible values of CAP controls:

- **all** - All replicas must reply
- **one** - One replica myst reply
- **quorum** - A majority of the replicas must reply (half plus one)
- **default** - uses per-bucket consistency property
- an **arbitrary** integer value - not recommended since "all", "one", "quorum", "default" are enough

By default query parameters (r, w, ...) are set at the **bucket level** to **quorum** which is a very reasonable value.

Querying with REST

Creating/updating an object with PUT:

curl -v -XPUT -H "Content-Type: **application/json**" -d '{"title": "The Prince","author": "Niccolo Machiavelli"}' http://localhost:10018/riak/**book**/**book1**

book: the bucket

book1: the key

Querying an object with GET:

curl -v -XGET http://localhost:10018/riak/**book**/**book1**

Deleting an object with DELETE:

curl -v -XDELETE http://localhost:10018/riak/**book**/**book1**

Creating an object and generating they key with POST:

curl -v -XPOST -H "Content-Type: application/json" -d '{"title": "The Prince","author": "Niccolo Machiavelli"}' http://localhost:10018/riak/**book**

Querying Buckets With REST

Listing buckets:

curl -v http://localhost:10018/buckets?buckets=true

Listing keys in a bucket:

curl -v http://localhost:10018/buckets/**book**/keys?keys=true

Querying bucket properties:

curl -v http://localhost:10018/riak/**book**

Updating bucket properties:

curl -v -XPUT -H "Content-Type: application/json" -d '{"props":{"**allow_mult**": true}}' http://localhost:10018/riak/**book**

Property "allow_mult" is set true above.

Secondary Indexes (2i)

- Objects are tagged with values stored as metadata
- Two types of secondary attributes: integers and strings
- Querying by exact match or range on **one** index
- Query results can be used as input to a MapReduce query

Storing tags:

```
curl -v -XPUT -H "x-riak-index-email_bin: sidorykp@gmail.com" \
 -H "x-riak-index-age_int: 39" \
 -H "Content-Type: application/json" \
 -d '{"firstName": "Pawel","lastName": "Sidoryk"}' http://localhost:10018/riak/user/user1
```

Querying by tag value:

```
curl -v http://localhost:10018/buckets/user/index/email_bin/sidorykp@gmail.com
```

Querying by a range of values:

```
curl -v http://localhost:10018/buckets/user/index/age_int/18/100
```

MapReduce in Riak

Map phase operates on single objects, generates a list of values. Map phase is used to filter and extract data from single objects. A variable length list of values can be produced from a single object

Reduce phase takes a list of values from the Map phase and **aggregates** them. Reduce can count values, group values, sort values.

MapReduce: Counting Objects

```
curl -XPOST http://localhost:10018/mapred \
-H 'Content-Type: application/json' \
-d @- \
<<EOF
{
"inputs":"book",
"query":[{
        "map":{
                "language":"javascript",
                "source":"function(riakObject) {
                        return [1];
                }"
        }},{
        "reduce":{
                "language":"javascript",
                "source":"function(values, arg){
                        return [values.reduce(function(acc, item){ return acc + item; }, 0)];
                }"
        }
}]
}
EOF
```

MapReduce: Map inputs

- Can be all objects in a bucket: "inputs":"book"
- Can be objects in a bucket filtered by a key:

```
"inputs": {
    "bucket":"book",
    "key_filters":[["string_to_int"],["greater_than", 5]]
}
```

- Can be a bucket filtered by index

```
"inputs": {
    "bucket":"user",
    "index":"age_int",
    "start":"18",
    "end":"100"
}
```

- Can be a full text search result

MapReduce: continued

We can have just Map without Reduce

Map and Reduce phases can be chained, e.g.:

map -> reduce1 -> reduce2

filter input (map)-> group by (reduce) -> order by (reduce)

Erlang and JavaScript functions can be used for Map and Reduce.

Extracting a member variable in Map is easy:

```
"source":"function(riakObject) {
    var m = Riak.mapValuesJson(riakObject)[0];
    return [m.firstName];
}"
```

Riak Search

- Distributed, full-text search engine
- Objects are indexed in a precommit hook
- Text extraction based on a mime type
- Can feed data into MapReduce
- Query syntax the same as in Lucene, most of Lucene operators are supported

Examples:

bin/**search-cmd** search book "author:Robert"

curl http://localhost:10018/**solr**/book/select?q=author:Robert

Riak Search: Weaknesses

- Uses timestamps, rather than vector clocks
- Does not use quorum values when writing (indexing) data and reading (querying) data
- Has no read-repair mechanism. If Search index data is lost, the entire data set must be re-indexed

Riak developers are working on a next-generation full text search engine: Yokozuna.

Java Client: Objects (POJOS)

```
public class Book {
    @RiakKey
    private String id;
    @RiakVClock
    private byte[] vclockBytes;
    private String title;
    @RiakIndex(name = "created_by_user_id_bin")
    private String createdByUser;
}
```

@RiakKey: Riak ID of an object

@RiakVClock: a serialized Riak VClock (we do not touch it)

@RiakIndex: member variable "createdByUser" stores **index values**. The index name is "created_by_user_id_bin"

Java Client: Buckets

Creating a bucket (with "allow_mult = true"):

```
Bucket bucket = riakClient.createBucket("book").allowSiblings(true).execute();
```

Fetching a bucket:

```
Bucket bucket = riakClient.fetchBucket("book").execute();
```

Listing keys in a bucket:

```
for (String key: bucket.keys()) {...}
```

Java Client: Objects

Storing a POJO:

```
Book book = new Book(id);
bucket.store(book).execute();
```

Fetching a POJO by a key:

```
bucket.fetch(id, Book.class).execute();
```

Specifying CAP controls:

```
bucket.fetch(id, Book.class).r(Quora.QUORUM).execute();
```

Fetching with the use of a Conflict Resolver:

```
BookConflictResolver cr = new BookConflictResolver();
bucket.fetch(id, Book.class).withResolver(cr).execute();
```

Java Client: Conflict Resolver

- The task of a Conflict Resolver is to **merge** siblings into one object
- Conflict Resolvers work on a client
- A Conflict Resolver must be defined for each class for which siblings may be produced
- A Conflict Resolver is active during reading objects
- Using dedicated data types (CRDT-s) is recommended for the merge to make sense

Conflict Resolver: Shopping Cart

```
public ShoppingCart resolve(Collection<ShoppingCart> siblings) {
        ShoppingCart shoppingCartM = null;
        for (ShoppingCart shoppingCart: siblings) {
            if (shoppingCartM == null) {
                shoppingCartM = shoppingCart;
            } else {
                Set<BookUidPair> addSet = shoppingCartM.getAddSet();
                Set<BookUidPair> addSetToMerge = shoppingCart.getAddSet();
                addSet.addAll(addSetToMerge);

                Set<BookUidPair> removeSet = shoppingCartM.getRemoveSet();
                Set<BookUidPair> removeSetToMerge = shoppingCart.getRemoveSet();
                removeSet.addAll(removeSetToMerge);
            }
        }
        return shoppingCartM;
}
```

Java Client: Indexes

Returning object **keys** with exact match on index "email_bin":

```
List<String> userKeys = bucket.fetchIndex(BinIndex.named("email_bin")).
withValue("sidorykp@gmail.com").execute();
```

Range query on index "age_int", index is used as an input filter to MapReduce, MapReduce must be used to return whole objects instead of just their keys:

```
IndexQuery indexQuery = new IntRangeQuery(IntIndex.named("age_int"),
"person", 18, 100);
MapReduceResult result = riakClient.mapReduce(indexQuery).addMapPhase
(mapFunction).execute();
```

Java Client: MapReduce

Example: returning all objects in a bucket "book":

```
Function map = new JSSourceFunction(
        "function(riakObject) {" +
                "var m = Riak.mapValuesJson(riakObject)[0];" +
                "return [m];" +
        "}");

MapReduceResult result = riakClient.mapReduce("book").addMapPhase(map).
execute();
```

Convergent Replicated Data Types

Eventually consistent replicas easily **converge** when we use data types for which we define a **merge** function that is:

- commutative
- associative
- idempotent

Such data types are called **C**onvergent **R**eplicated **D**ata **T**ypes (**CRDT**-s).

The simplest example: grow-only set (a set that supports "add" and does not support "remove"). The "merge" function is implemented by union of sets.

Important CRDT-s:

- vector clock
- distributed counter
- observed-remove set (OR-Set), good for a distributed shopping cart
- multi-value register
- Treedoc, good for cooperative document editing

CRDT example: OR-Set

Consists of 2 sets: "**add set**" and "**remove set**". "Add set" and "remove set" contain pairs {elem, uid} where "elem" is the actual element being stored in the OR-Set and "uid" is a unique identifier allocated during an "add" operation.

We remove an "elem" from an OR-Set by adding all {elem, uid1}, {elem, uid2} etc. pairs found in the "add set" to the "remove set".

Merging OR-Sets is easy: we make a union of "add sets" to get a merged "add set" and we make a union of "remove sets" to get a merged "remove set".

- adding "book1":
  - add set: [{"book1", **uid1**}], remove set: []
- removing "book1:"
  - add set [{"book1", uid1}], remove set: [{"book1", uid1}]
- adding "book1" once more:
  - add set [{"book1", uid1}, {"book1", **uid2**}], remove set: [{"book1", uid1}]

CRDT-s: continued

Riak developers are working on implementation of a library of CRTD-s straight in Riak. Currently one has to implement his/her own CRDT-s on top of Riak.

Some data operations cannot be implemented by a CRDT, e.g. a non-negative counter. Some data operations require a global synchronization and they cannot be implemented in a eventually-consistent manner.

Riak: Strengths

- Efficient concurrency control with vector clocks
- High scalability
- Fault tolerance
- Tunable consistency
- Efficient query options: secondary indexes, MapReduce, Riak Search
- Many client interfaces: Java, Python, Ruby etc.
- Supports REST and Protocol Buffers

Riak: Weaknesses

- No security: you must implement your own security layer
- Secondary Indexes do not scale well beyond > 512 nodes
- Failed MapReduce jobs are not restarted (rather use Hadoop for long running MapReduce jobs)
- No high level query language
- Various weaknesses of Riak Search

Conclusions

- Don't be afraid of eventual consistency
- Consider Riak if you want a complete data solution for the Cloud or Big Data
- Scale your applications at ease

Further Reading

- basho.com
- **Eventually Consistent - Revisited**: www.allthingsdistributed.com/2008/12/eventually_consistent.html
- M. Shapiro et al. **A comprehensive study of convergent and commutative replicated data types.** Technical Report RR-7506, INRIA, 2011.
- Nuno Preguica et a. **A commutative replicated data type for cooperative editing.** Int. Conf. on Distributed Computing Systems (ICDCS), 2009.

Contact me: **sidorykp@gmail.com**

Questions ?