

NIE-PDB: Advanced Database Systems

<http://www.ksi.mff.cuni.cz/~svoboda/courses/NIE-PDB/>

Lecture 7

Key-Value Stores: RiakKV

Martin Svoboda

martin.svoboda@fit.cvut.cz

11. 11. 2025

Charles University, Faculty of Mathematics and Physics

Czech Technical University in Prague, Faculty of Information Technology

Lecture Outline

Key-value stores

- Introduction

RiakKV

- Data model
- HTTP interface
- **CRUD operations**
- **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 $\Rightarrow$ **great performance, easily scaled, ...**
- Simple model $\Rightarrow$ **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

Query Patterns

Basic **CRUD** operations

- Only when a key is provided
- $\Rightarrow$ 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

Key-value store

- <https://www.tiot.jp/en/solutions/riak/>
- Features
 - Open source, incremental scalability, automatic **sharding**, **peer-to-peer replication**, high availability, fault tolerance, ...
- Originally developed by **Basho Technologies**
- Implemented in **Erlang**
 - General-purpose functional programming language and runtime system with garbage collection
 - Its main strength is concurrency and distribution
- Operating systems: **Linux**, Mac OS X, ... (not Windows)
- Initial release in 2009
 - Version we cover is 3.0.10 (May 2022)

Data Model

Dataspace structure

Instance ($\rightarrow$ **bucket types**) $\rightarrow$ **buckets** $\rightarrow$ **objects**

- **Bucket type**

- Optional logical **collection of buckets**
 - When not stated explicitly, the `default` type is assumed
- Primarily allows for shared **configuration of buckets**
 - But also forms a **namespace for buckets**
 - As well as allows to define **user permissions**

- **Bucket**

- Logical **collection of key-value objects**
- Allows to override inherited bucket type properties
 - E.g., replication factor, read / write quora, ...

Data Model

Dataspace structure (cont'd)

- **Object** = one **key-value pair**
 - **Key**: Unicode string **unique within a bucket**
 - **Value**: basically anything (text, binary object, image, ...)
- Each object is also associated with additional **metadata**
 - Especially **content type**
 - I.e., data format of the value part
 - **Media types** (MIME types) are used for this purpose
 - E.g.: `text/plain`, `application/json`, `image/jpeg`, ...
 - But also certain **internal metadata**
 - Causal context (vector clock), timestamp of the last modification, ...

Data Model: Design Questions

Possible data modeling strategies

- **Multiple buckets**
 - Each for objects of just a **single entity type**
 - E.g., one bucket for actors, one for movies, each actor and movie has its own object
 - Allows for easier key management
- **Single bucket**
 - Serves for objects of **various entity types**
 - E.g., one bucket for both actors and movies, each actor and movie still has its own object
 - **Structured keys** might thus help
 - Distinct prefix can be used for each entity type
 - E.g., `actor_trojan`, `movie_medvidek`

Riak Usage: Querying

Basic **CRUD** operations

- Create, Read, Uppdate, and Deleate
 - All 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: Interfaces

Application interfaces

- **HTTP API**
 - 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, ...

HTTP API

cURL = tool for **sending requests and receiving responses** via **HTTP**

- **-u** **user:password** (alternatively also **--user**)
 - **User credentials** to be used for **server authentication**
- **-X** **command** (**--request**)
 - **Request method** to be used (GET, PUT, ...)
- **-H** **header** (**--header**)
 - **Extra headers** to be included when sending the request
- **-d** **data** (**--data**)
 - **Data to be sent** to the server
- **-i** (**--include**)
 - Whether response headers should also be printed

Basic Operations

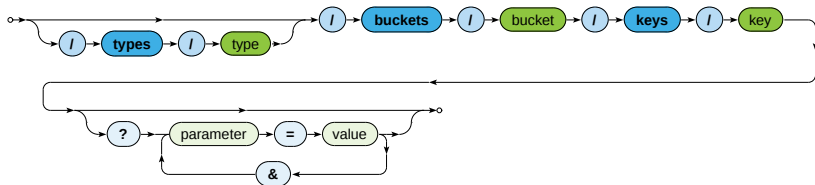
CRUD Operations

Basic object operations

- Create: **POST** or **PUT** methods
 - **Inserts** a key-value pair into a given bucket
- 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

Generic **URL pattern** for all basic object operations



Optional parameters

- Allow to override bucket-level properties for a given request
 - **r**, **w**: read / write quorum to be attained
 - ...
- Permitted parameters depend on the particular operation

CRUD Operations: Create and Update

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

- **Key is specified** $\Rightarrow$ **PUT** method
 - Transparently **inserts / updates** (replaces) a given object
 - I.e., when updating, everything really must be specified again
- **Key is missing** $\Rightarrow$ **POST** method (insertion only)
 - Key will be generated automatically and returned via a header
 - E.g.: 4zmJhCNhM4h6mUJVw35Ck0uNZ28
- Buckets as such are created transparently, bucket types not

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

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

```
...  
Content-Type: text/plain  
Content-Length: 17  
X-Riak-Vclock: a85hYGBgzGDKBVI8XxW02dii9T4wMKgLZjAlMuWxMti+WXXHLwsA  
Last-Modified: Sun, 25 Sep 2022 15:14:05 GMT  
...
```

```
Ivan Trojan, 1964
```

CRUD Operations: Delete

Removes a key-value pair from a given bucket

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

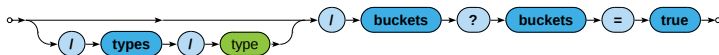
Example

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

Bucket Operations

List of all existing buckets

- I.e., buckets with at least one existing object
- Should not be used in production environments
 - Because of inefficiency, every cluster node needs to be involved



Example

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

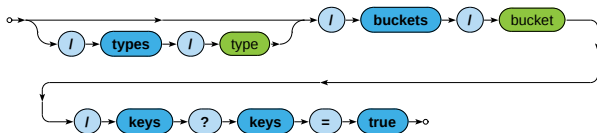
```
Content-Type: application/json
```

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

Bucket Operations

List of all existing keys in a given bucket

- Should not be used in production environments, once again



Example

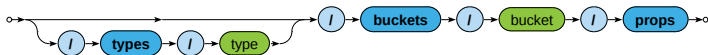
```
curl -i -X GET http://localhost:8098/buckets/actors/keys?keys=true
```

Content-Type: application/json

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

Bucket Properties

Setting and retrieval of **bucket properties**



- **Retrieval** $\Rightarrow$ **GET** method
 - Lists current values of all bucket properties
- **Update** $\Rightarrow$ **PUT** method
 - Updates values of selected bucket properties
 - I.e., values of not mentioned properties are preserved intact
- **Reset** $\Rightarrow$ **DELETE** method
 - Resets all or just selected bucket properties
 - I.e., removes them or replaces them with bucket type defaults

Bucket Properties: Examples

Update of selected properties

```
curl -i -X PUT \
  -H 'Content-Type: application/json' \
  -d '{ "props" : { "n_val" : 3, "w" : "all", "r" : 1 } }' \
  http://localhost:8098/buckets/actors/props
```

Reset of selected properties

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

Reset of all properties

```
curl -i -X DELETE \
  http://localhost:8098/buckets/actors/props
```

Bucket Properties

Important properties

- **n_val**: replication factor
- **r / w**: read / write quorum
 - Particular value, **all** (all replicas), **quorum** ($n_val/2 + 1$)
- **search_index**
 - Name of the associated **search index**, if any
- **datatype**
 - Name of the associated **data type**, if any
 - E.g.: **counter**, **set**, **map**, ...
- **allow_mult**
 - Whether to allow **sibling objects** to be created

Data Types

Motivation

Replica **conflict**

- Situation when **not all replicas** of a given object are **identical**
 - I.e., two or even more of them are mutually inconsistent

Riak is an **AP system** $\Rightarrow$ such **conflicts are unavoidable**

- And so they need to be **resolved** somehow...
 - Either automatically or manually
- Until now we only worked with **ordinary objects**
 - With atomic values only
 - And both resolution strategies possible
- But we also have an alternative in a form of **data type objects**
 - Inspired by the concept of **CRDTs**

CRDTs

Convergent Replicated Data Types

- **Generic concept** introducing a couple of data types
 - Each useful for a different real-world use case
 - **G-Counter** (Grow-only Counter), **PN-Counter** (Positive-Negative Counter), **G-Set** (Grow-only Set), ...
- Particular **CRDT definition** involves a description of...
 - Permitted **content** – can be atomic as well as structured
 - Permitted **operations**
 - **Convergence rule**
 - Specifically tailored mechanism used for **conflict resolution**

CRDTs implemented in Riak

- **Counter, set, map, register, flag, ...**
 - Not all of them can be used at the top level, though

Data Types: Counters

Counter

- **Integer counter**
 - Both positive and negative values are permitted
 - When a new counter is first used, its value is **initialized to 0**
- **Operations**
 - **Increment / decrement** by a given value
 - I.e., it is not possible to set the counter to a particular value
 - Just relative changes are permitted
- **Convergence rule**
 - **All requested increments / decrements are eventually applied**

Data Types: Sets

Set

- **Unordered collection of unique binary values**
 - E.g., strings
 - When a new set is first used, it is **initialized as an empty set**
- Operations
 - **Addition / removal** of one or more elements
- Convergence rule
 - **Addition wins over removal**
 - At the level of individual elements

Data Types: Maps

Map

- **Unordered collection of embedded name-value pairs**
 - **Names** are strings
 - **Values** can be anything
 - I.e., **registers**, **flags**, but also **counters**, **sets** and even **maps**
 - **Complex data structures** can therefore be easily created
 - Names must be suffixed according to the types of values
 - E.g., `field_register`, `field_flag`, ...
- **Operations**
 - **Addition / update / removal** of a given element
- **Convergence rule**
 - **Addition / update wins over removal**
 - Values themselves are treated recursively based on their types

Data Types: Registers & Flags

Register

- Allows to store **any binary value** (e.g., string)
- Convergence rule
 - **The most chronologically recent value wins**
- Registers can only be stored within maps
 - I.e., not at the top level for entire objects

Flag

- **Boolean value**
 - `enable` (true), and `disable` (false)
- Convergence rule: **enable wins over disable**
- Flags can only be stored within maps, too

Usage of Data Types

Activation

- Via **bucket type properties** (i.e., not individual buckets)
 - Property **datatype** is set to the desired data type
 - Possible values: **counter**, **set**, **map**, ...
 - Property **allow_mult** must be enabled

Usage

- Different URL pattern for requests is assumed
 - Keyword **datatypes** is expected instead of **keys**



Example: Counters

Initialization / update

- Operations **increment** and **decrement** can be used
 - Both actually with positive / negative values

```
curl -i -X POST \  
  -H 'Content-Type: application/json' \  
  -d '{ "increment" : 0 }' \  
  http://localhost:8098/types/counters/buckets/movies/datatypes/en
```

Retrieval

```
curl -i -X GET \  
  http://localhost:8098/types/counters/buckets/movies/datatypes/cs
```

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

```
{ "type" : "counter", "value" : 4 }
```

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

Mechanisms

- **Indexation**
 - Triggered whenever Riak object is changed (inserted, ...)
 - **Riak object** $\xrightarrow{\text{extractor}}$ **Solr document** $\xrightarrow{\text{schema}}$ **Solr index**
- **Querying**
 - **Riak search query** $\rightarrow$ Solr search query $\rightarrow$ Solr response
 - List of matching Solr **documents with scores**
 - Each providing identification of the associated **source object**

Extractors

Extractor = parser for object values

- Produces **fields** to be indexed
- Chosen automatically based on a **content type**
 - E.g.: `application/json` $\Rightarrow$ JSON extractor

Available extractors

- For common data formats...
 - Plain text, XML, JSON, *noop* (unknown content type)
- For Riak **data types**...
 - Counter (`application/riak_counter`)
 - Set (`application/riak_set`)
 - Map (`application/riak_map`)

User-defined custom extractors (implemented in Erlang)

Extractors: Plain Text

Plain text extractor (`text/plain`)

- Single field with the whole value content is extracted

Example

```
Dira u Hanusovic, 2014
```

```
[  
  { <<"text">>, <<"Dira u Hanusovic, 2014">> }  
]
```

Extractors: XML

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

- One field is extracted for each simple **element** or **attribute**
 - But only when enabled, i.e., its name contains a **type suffix**
- Available type suffixes
 - **Single-value**
 - `_s` (string), `_i` (integer), `_f` (float), `_b` (boolean), ...
 - **Multi-value**
 - When multiple values are expected
 - E.g., for several sibling elements of the same name
 - `_ss` (strings), `_is` (integers), `_fs` (floats), `_bs` (booleans), ...
- **Dot notation** is used for flattened names of extracted fields
 - `.` for embedded elements (e.g., `movie.title_s`)
 - `@` for attributes (e.g., `movie@year_i`)

Extractors: XML

Example

```
<?xml version="1.1" encoding="UTF-8"?>
<movie year_i="2014" language="cs">
  <title_s>Dira u Hanusovic</title_s>
  <details>
    <length>102</length>
    <rating_s>**</rating_s>
  </details>
  <genre_ss>comedy</genre_ss>
  <genre_ss>drama</genre_ss>
</movie>
```

```
[
  { <<"movie@year_i">>, <<"2014">> },
  { <<"movie.title_s">>, <<"Dira u Hanusovic">> },
  { <<"movie.details.rating_s">>, <<"**">> },
  { <<"movie.genre_ss">>, [ <<"comedy">>, <<"drama">> ] }
]
```

Extractors: JSON

JSON extractor (`application/json`)

- Similar principles as the XML extractor applies

Example

```
{  
  "title_s" : "Dira u Hanusovic",  
  "language" : "cs",  
  "year_i" : 2014,  
  "details" : { "length" : 102, "rating_s" : "***" },  
  "genre_ss" : [ "comedy", "drama" ]  
}
```

```
[  
  { <<"title_s">>, <<"Dira u Hanusovic">> },  
  { <<"year_i">>, <<"2014">> },  
  { <<"details.rating_s">>, <<"*">> },  
  { <<"genre_ss">>, [ <<"comedy">>, <<"drama">> ] }  
]
```

Indexing Schema

Solr document

- Extracted fields + **auxiliary fields**
 - `_yz_rt` (**bucket type**), `_yz_rb` (**bucket**), `_yz_rk` (**key**), ...
 - Allow for the identification of the source Riak object

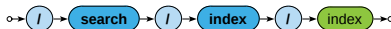
Solr schema

- Describes how values of fields are indexed within Solr
 - Values are **analyzed, tokenized, and filtered**
 - E.g., stop words removed, stemmers applied, ...
 - **Triples** (`token value`, `field name`, `document id`) are **indexed**
- `_yz_default` = default predefined schema
 - Suitable for debugging
 - Does not support specific national characters, ...
 - **Custom schemas** can also be created

Index Initialization

Step 1: index creation

- Default (`_yz_default`) schema is assumed when not specified



Example

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

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

Index Initialization

Step 2: index association

- Index must then be associated with particular buckets
 - Via `search_index` bucket property
- Note that the already existing objects will not be indexed

Example

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

Search Requests

Search queries



- Parameters

- **q**: **search query conditions** to be satisfied
- **wt**: **response writer** to be used, i.e., **data format** of the result
 - E.g.: `json`, `csv`, `xml`, `php`, ...
- **sort**: **ordering criteria**
 - Document **scores** or both single-/multi-value **fields** can be used
 - By default (when not specified), `score desc` is assumed
 - Multiple criteria are separated by commas
 - E.g.: `year_i desc, title_s asc`
- **start** / **rows**: **pagination** of matching documents

Search Conditions

Term searches

- Value of a given field must be equal to the provided term
 - In case of a multi-value field, at least one of its values
- E.g.: `title_s:Samotari`

Phrase searches

- Group of more terms needs to be wrapped by double quotes
- E.g.: `title_s:"Dira u Hanusovic"`

Wildcard searches

- Available wildcards
 - `?` matches exactly one arbitrary character
 - `*` matches zero or more arbitrary characters
- E.g.: `title_s:*Bob?le` matching `Bobule`, `2Bobule`, ...

Search Conditions

Range searches

- Range of values between a pair of bounds
 - `[` and `]` denote **inclusive** bounds, `{` and `}` **exclusive** bounds
 - `*` denotes positive / negative infinity
- E.g.: `year_i:[2015 TO *}`

Logical expressions

- Logical connectives can be used for more complex queries
 - **AND** for **conjunction**, **OR** **disjunction** and **NOT** **negation**
 - Auxiliary **parentheses** `()` can also be utilised
- E.g.: `genre_ss:action OR genre_ss:fantasy`

Search Requests

URL encoding issues

- Step 1: preparing the intended **search condition**
 - E.g.: `title_s:*Bobule OR (year_i:[2020 TO *} AND stars_s:\*\*)`
 - Undesired **Solr metacharacters** are deactivated by escaping
 - E.g.: `:`, `*`, `?`, `(`, `)`, `[`, `]`, `{`, `}`, ...
- Step 2: encoding **unsafe and reserved URL characters**
 - Each needs to be replaced with the corresponding code
 - At least those necessary...
 - E.g.: space `%20`, `"` `%22`, `\` `%5C`, `:` `%3A`, `*` `%2A`, `?` `%3F`, `(` `%28`, `)` `%29`, `[` `%5B`, `]` `%5D`, `{` `%7B`, `}` `%7D`, ...
 - E.g.: `title_s%3A%2ABobule%20OR%20%28year_i%3A%5B2020%20TO%20%2A%7D%20AND%20stars_s%3A%5C%2A%5C%2A%29`

Search Requests

URL encoding issues (cont'd)

- Step 3: preparing curl request
 - Undesired **shell metacharacters** also need to be suppressed
 - E.g.: `&`, `?`, ...
 - E.g.: `...\&q=...` instead of `...&q=...`

Example

```
curl -i -X GET \  
  http://localhost:8098/search/query/imovies\?wt=json\&q=year_i%3A2020
```

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

CAP properties

- **AP system: availability + partition tolerance**
 - I.e., availability is preferred to consistency
- Strong consistency can also be achieved
 - When activated within the whole cluster
 - And appropriate quora are set:
 - $w > n_val/2$ for write quorum
 - $r > n_val - w$ for read quorum
 - However, such an approach is deprecated

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

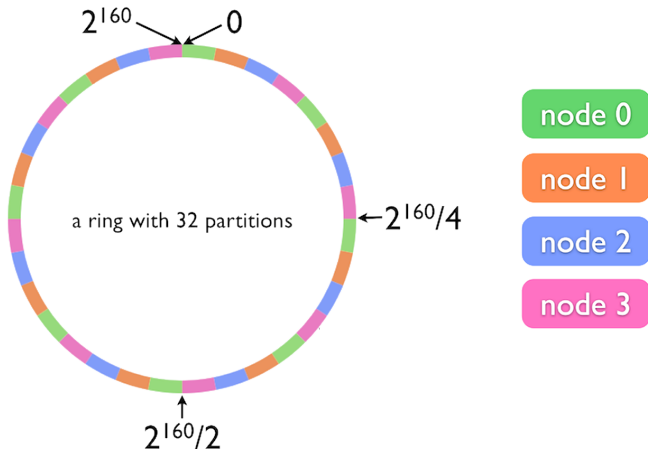
Riak Ring

- 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 node 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**
- **Search 2.0** full-text based on Apache Solr