Key-value Stores II

Embedded, Distributed, and In-memory Stores

Lecture 5 of *NoSQL Databases* (PA195)

David Novak & Vlastislav Dohnal
Faculty of Informatics, Masaryk University, Brno

Key-value Stores: Basics

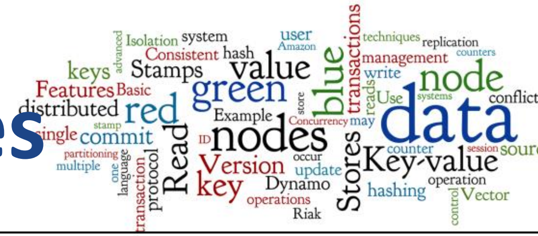
- A simple hash table (map), primarily used when all accesses to the database are via primary key
  - key-value mapping
- In RDBMS world: A table with two columns:
  - `ID` column (`primary key`)
  - `DATA` column storing the value (unstructured `BLOB`)
- Basic operations:
  - Get the value for the key `value:= get(key)`
  - Put a value for a key `put(key, value)`
  - Delete a key-value `delete(key)`

Querying

- We can query by the key
- To query using some *attribute* of the value is not possible (in general)
  - We need to read the value to test any query condition

- What if we do not know the key?
  - Some systems support additional functionality
    - Using some kind of additional index (e.g., full text)
    - The data must be indexed first
    - Example later: Riak search

Techniques in Distributed Stores



Consistent hashing	Sharding (data partitioning)
Virtual nodes	Balancing of data
Replication (consecutive nodes)	Read/write scalability & reliability
Read/write quora	Consistency and r/w efficiency
Vector stamps	Avoid/detect update conflicts
Gossip protocol	Node join/leave/failure
Multi-version concurrency control	Transaction isolation
Two-phase commit protocol (2PC)	Distributed transactions

Representatives

amazon DynamoDB

riak

levelDB

Infinispan

ORACLE BERKELEY DB 12c

redis

RocksDB

Project Voldemort

MEMCACHED

MapDB

hazelcast

EHCACHE

Ranked list: http://db-engines.com/en/ranking/key-value+store

Agenda

- Embedded local storages
  - LevelDB
    - Local storage for many systems, Log-structured Merge Tree
- Distributed key-value Stores - representatives
  - Riak
    - Basics, Riak Links & Indexes & Riak Search, Internal features
  - Infinispan
    - Basic features, example, advanced features, indexing & searching
- Memory caches
  - Memcached
- Serialization: Protocol Buffers, Apache Thrift

Embedded Stores

- The database system is actually a library
  - One programming language, possibly wrapper in other lang
- We can use it directly in our application
  - It is embedded within the application
- Advantage:
  - Speed: the fastest connection between application and DB
- Disadvantages:
  - Database cannot be distributed
    - actually, embedded database nodes can form a distributed storage
  - Database cannot be shared by two applications

Embedded Stores: Representatives

- Embedded local storages
  - LevelDB
    - Local storage for many systems, Log-structured Merge Tree
    - C++
  - MapDB
    - Java project, one-man show
    - memory-mapped file storage
  - RocksDB
    - Embeddable persistent key-value store
    - Facebook
    - C++, but also connector from Java

Representative: LevelDB

LevelDB: Basics

- Embedded key-value store (string to string)
  - Using ideas from Google's BigTable
  - Developers: Jeffrey Dean and Sanjay Ghemawat from Google
- Initial release date: 2011
- License: New BSD Licence
- Language: C++
- LevelDB is a backend for Google Chrome's IndexDB

http://github.com/google/leveldb http://db-engines.com/en/system/LevelDB

LevelDB: Fundamental Features

levelDB

- Basic architecture is a LSM Tree (see below)
- Sorted by keys
- Arbitrary byte arrays

- Basic operations: Get(), Put(), Del(), Batch()
- Bi-directional iterators

Log-structured Merge Tree

Log-structured Merge Tree (LSM Tree)

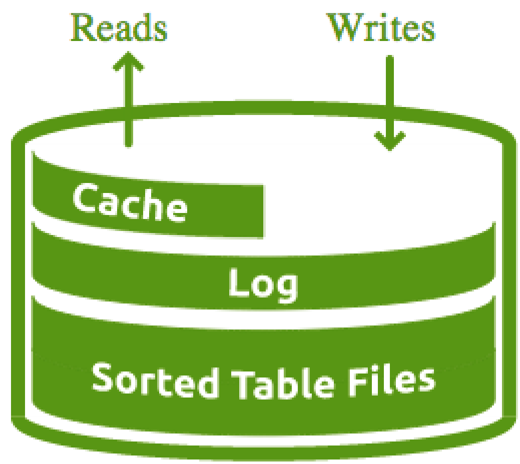
- data structure for indexed access to data files
  - can handle high write frequency
- writes applied to a sorted structure in memory
  - regularly synchronized to a sorted disk storage
- read ops merge data from memory & disk

O'Neil, Patrick E.; Cheng, Edward; Gawlick, Dieter; O'Neil, Elizabeth (June 1996). "The log-structured merge-tree (LSM-tree)". Acta Informatica 33 (4): 351–385.

LevelDB: Basic Architecture

- Writes go straight into a log
- Log is flushed to sorted table files (SSTables)
- Reads merge the log and the SSTable files

- Cache speeds up common reads

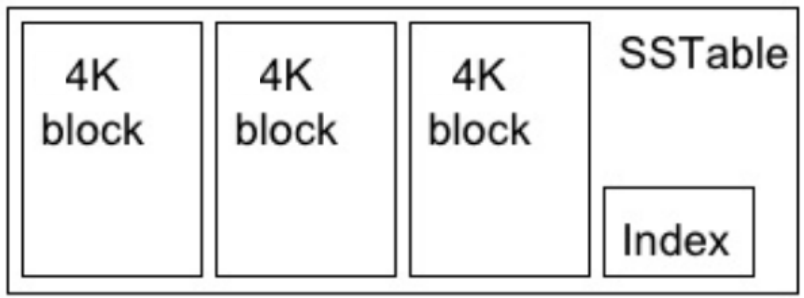


source: https://r.va.gg/presentations/nodejsdub/

Basic Storage: SSTable Files

Sorted String Table (SSTable) Files:

- Limited to ~2MB each
- Divided into 4K blocks
- Final block is an index
- Bloom filter used for lookups



Levels in LevelDB

Log: Max size of 4MB then flushed into a set of Level 0 SSTables

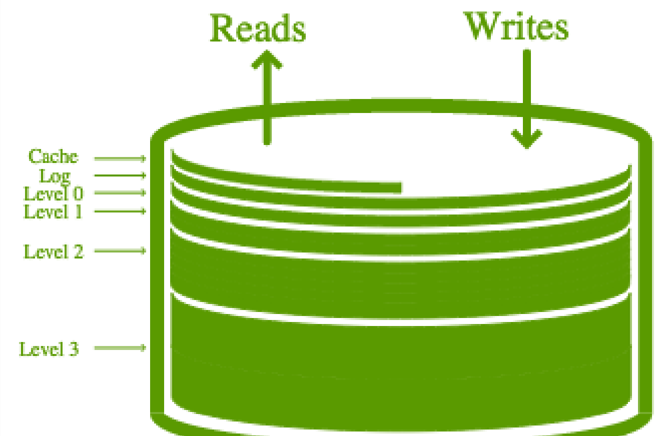
Level 0: Max of 4 SST files then the files compacted into Level 1

Level 1: Max total size of 10MB then the files compacted into L2

Level 2: Max total size of 100MB then the file compacted into L3

Level 3+: Max total size of 10x previous level size then the files compacted into next level.

0 ↠ 4 SST, 1 ↠ 10M, 2 ↠ 100M,
3 ↠ 1G, 4 ↠ 10G, 5 ↠ 100G, 6 ↠ 1T,
7 ↠ 10T, ...



source: https://r.va.gg/presentations/nodejsdub/

LevelDB: Universal Backend

- LevelDB is a popular backend storage for many (distributed) database systems
  - Web browser IndexDB (in Chrome)
  - Riak, Infinispan
  - LevelUp/LevelDown for Node.js
  - etc.

Agenda

- Embedded local storages
  - LevelDB
    - Local storage for many systems, Log-structured Merge Tree
- Distributed key-value Stores - representatives
  - Riak
    - Basics, Riak Links & Indexes & Riak Search, Internal features
  - Infinispan
    - Basic features, example, advanced features, indexing & searching
- Memory caches
  - Memcached
- Serialization: Protocol Buffers, Apache Thrift

Distributed K-V Store: Riak

Riak: Basic Information

- Open source, distributed key-value database
  - Company Basho, first release: 2009
  - OS: Linux, BSD, Mac OS X, Solaris
- Language: Erlang, C, C++, some parts in JavaScript
- Built-in support for MapReduce
- Provides a full-text search engine on the data
  - “Riak search”

basic info: http://db-engines.com/en/system/Riak website: https://riak.com/products/riak-kv/

Riak: Basic Mission

- Availability
  - Riak replicates and retrieves data intelligently so it is available for read/write operations, even in failure conditions
- Fault-Tolerance
  - You can lose access to many nodes due to network partition or hardware failure without losing data
- Operational Simplicity
  - Add new machines to your Riak cluster easily without incurring a larger operational burden
- Scalability
  - Riak automatically distributes data around the cluster and yields a near-linear performance increase as you add capacity

source: https://riak.com/

Riak: Basics

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

namespace of keys

Terminology in RDBMS vs. Riak

- Stores keys into buckets = a namespace for keys
  - Like tables in a RDBMS, directories in a file system, ...
  - Bucket has its own properties
    - `n_val` – replication factor
    - `allow_mult` – allowing concurrent updates
    - ...

Riak: Interaction with the DB

- Default: HTTP Interface (Web services)
  - GET (retrieve value), PUT (update), DELETE (delete), …
  - example:
    `http://localhost:8098/buckets/test/keys/mykey`

- Native Erlang interface
- Connectors from many (not) standard languages
  - C, C#, C++ , Clojure, Dart, Go, Groovy, Haskell, Java, JavaScript, Lisp, Perl, PHP, Python, Ruby, Scala, Smalltalk

Riak: Additional Functionality

- Riak can have several types of local storage
  - typically referred to as backends
  - memory, LevelDB, etc.

- Riak has additional functions to work with values
  - Riak links
  - Indexes
  - Riak search

Riak: Links

- A way to create relationships between objects
  - Like foreign keys in RDBMS or associations in UML
- Attached to objects via HTTP header “Link”

- Add a book and link to its author:

```
curl -X PUT http://localhost:8098/buckets/books/keys/NoSQL -
d '{"title": "Big Data a NoSQL databáze", "year": "2015"}' -
H 'Link: </buckets/authors/keys/David>; riaktag="wrote"'
```

Riak: Link Walking

- Locate a key and then continue by link(s)
  - target specification: /bucket,linktype,[0/1]
- Find the authors who wrote book NoSQL

```
curl -i http://localhost:8098
/buckets/books/keys/NoSQL/authors,wrote,1
```

  - Restrict to bucket authors
  - Restrict to tag wrote
  - *1 = include this step to the result*

Riak: Indexes

- Secondary indexes on the values
  - Search key-value pairs based on the content
- Indexes kept locally on every virtual node

- Types of indexes:
  1. integer index (search by value or interval of values)
  2. binary index (search by any type of value)

  3. fulltext index (Riak search)

Riak: Indexes (2)

- Indexes cannot be managed automatically
  - Because there is no schema on the values
- When inserting a value, one can use index
  - In HTTP API, use special HTTP headers

```
curl -X PUT http://localhost:8098/buckets/authors/keys/David
-H 'x-riak-index-surname_bin: Novak'
-H 'x-riak-index-phone_int: 5062'
-d '{"name": "David", surname "Novák", "phone ext": 5062 }'
```

Riak Search: Fulltext via Solr

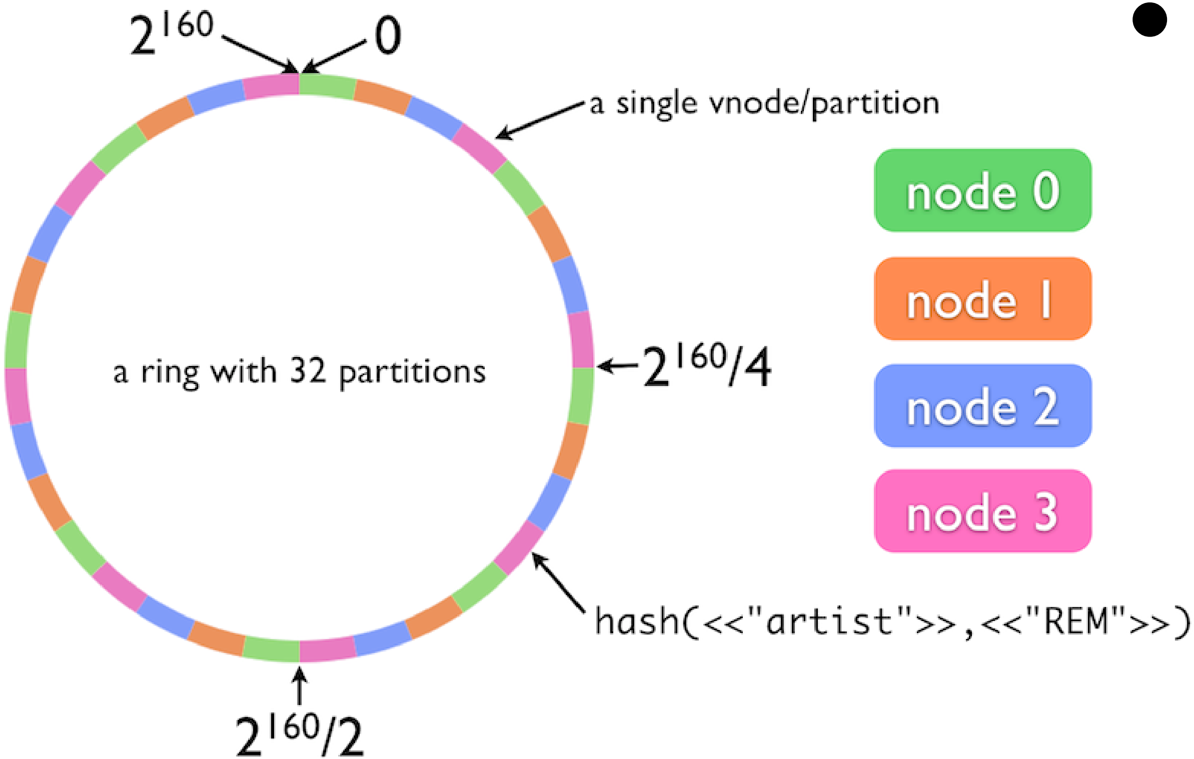
- Riak provides a distributed, full-text search engine
  - Implemented using Solr (Lucene)
  - Inserted values are indexed automatically
  - ...and then search the data by “terms”
- Key features:
  - Different parsers for different mime types
    - JSON, XML, plain text, ...
  - Exact match queries: “Bus”
  - Wildcards: “Bus*”, “Bus?”
  - Prefix matching, proximity searches, range queries...

Documentation: https://docs.riak.com/riak/kv/2.2.3/developing/usage/search/index.html

Riak: Internal Features

- Let us have a look behind the scene of Riak
  - Consistent hashing
    - and virtual nodes
  - Peer-to-peer (masterless) data replication
  - Read/Write Quorums
  - Hinted handoffs
    - High availability
  - Vector clocks
    - Riak siblings
  - Gossip protocol
  - Query processing
  - Riak Enterprise

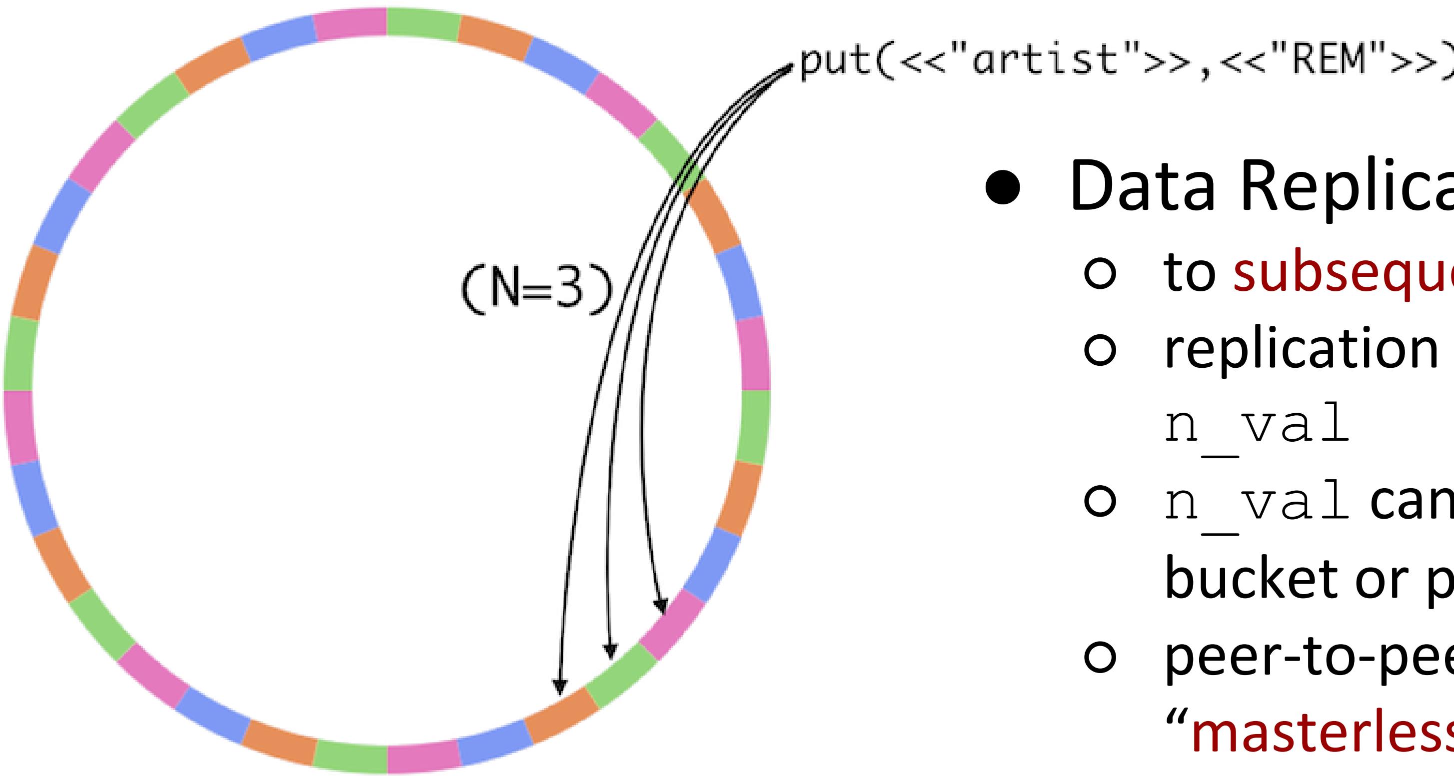
Consistent Hashing



- Data Partitioning
  - consistent hashing into $[0, 2^{160}]$
  - data balancing achieved by virtual nodes (`vnode`)

source: http://docs.basho.com/riak/latest/theory/concepts/

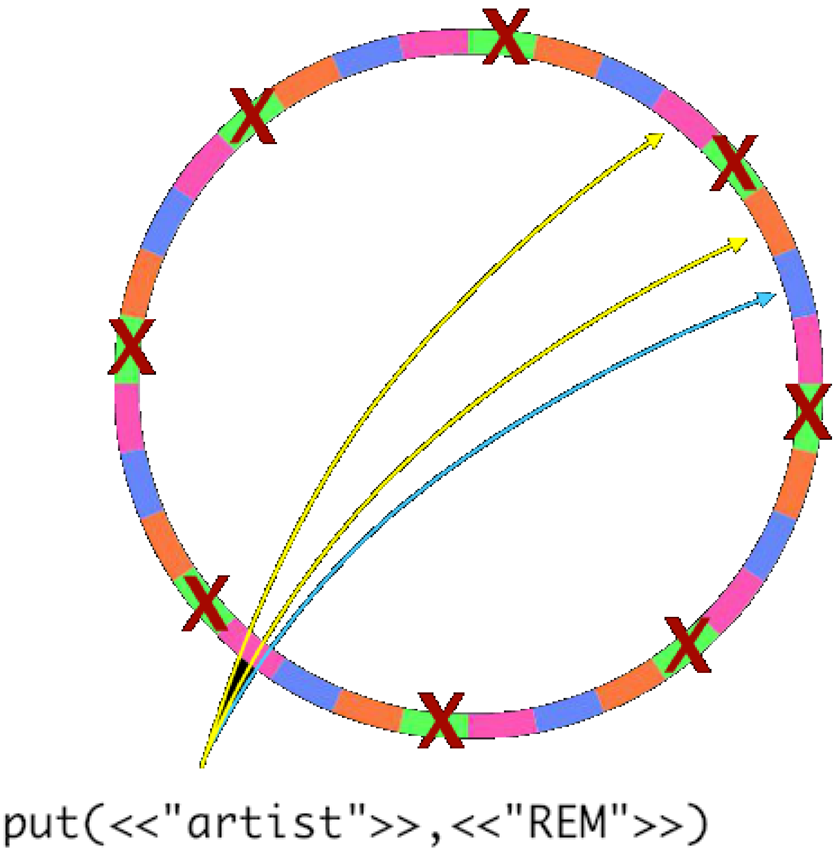
P2P Replication



- Data Replication
  - to subsequent nodes
  - replication factor `n_val`
  - `n_val` can be set per bucket or per object
  - peer-to-peer "masterless" replication

source: http://docs.basho.com/riak/latest/theory/concepts/

Hinted Handoffs



- Goal: High availability
- Hinted handoff
  1. In case of node failure
  2. Neighboring nodes temporarily take over storage operations
  3. When the failed node returns, the updates received by the neighboring nodes are handed off to it

source: http://docs.basho.com/riak/latest/theory/concepts/

Vector Clocks

- Any node is able to receive any request
  - We need to know which version of a value is current
- When a value stored, it is tagged with a *vector clock*

```
curl http://localhost:8098/raw/plans/dinner
-X PUT --data "Wednesday"
```

```
curl -i http://localhost:8098/raw/plans/dinner
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVIsrLnh3BlMiYx5rAzLJpw7wpcFAA==
Content-Type: text/plain
Content-Length: 9

Wednesday
```

Vector Clocks (2)

- For each update, Riak can determine:
  - Whether one object is a direct descendant of the other
  - Whether the objects are descendants of a common parent
  - Whether the objects are unrelated in recent heritage
- If the objects are unrelated then Riak can:
  - Auto-repair data
  - Provide the data to the user to decide

```
curl -X PUT -H "X-Riak-ClientId: Ben"
-H "X-Riak-Vclock:
a85hYGBgzGDKBVIsrLnh3BlMiYx5rAzLJpw7wpcFAA=="
http://localhost:8098/raw/plans/dinner --data "Tuesday"
```

source: http://docs.basho.com/riak/latest/theory/concepts/Vector-Clocks/

Riak: Siblings

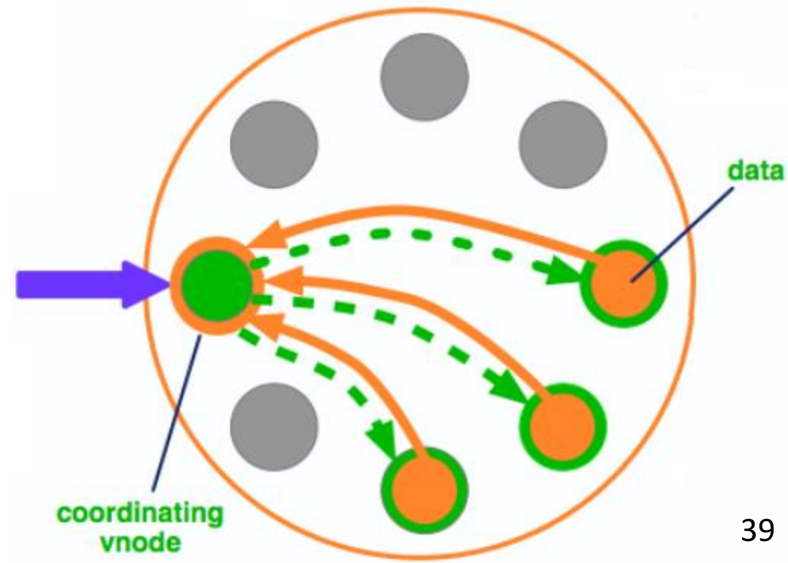
- Siblings of objects are created in case of:
  - **Concurrent writes** – two writes occur simultaneously with same vector clock value
  - **Stale vector clock** – stale v. clock value provided by client
  - **Missing vector clock** – write without a vector clock

- When retrieving an object we can:
  - Retrieve all siblings
  - Resolve the inconsistency

source: https://docs.riak.com/riak/kv/2.2.3/learn/concepts/causal-context.1.html

Riak: Request Sharing

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



Riak Enterprise

- Commercial extension of Riak
- Adds support for:
  - Multi-datacenter replication
    - Using more clusters and replication between them
    - Real-time replication – incremental synchronization
    - Full-sync replication – entire data set is synchronized
  - SNMP monitoring
    - Simple Network Management Protocol
  - JMX monitoring
    - Java Management Extensions

Distributed K-V Store: Infinispan

Basics

- Developer: Red Hat, open source community
  - Originally developed as a memory-based cache for JBoss
- Initial release date: 2009, current version 12.1
- License: Apache version 2
- Language: Java
  - embedding to Java application OR
  - external service via various APIs (REST service, Memcached protocol, Hot Rod) OR
  - connectors: Groovy, Scala

http://infinispan.org/ http://db-engines.com/en/system/Infinispan

Infinispan: Hello World

```
public static void main(String args[]){
    Cache<String, Object> store =
                    new DefaultCacheManager().getCache();
    store.put("key1", new MyClass("value1"));
    store.put("key2", "value2");
    if (store.containsKey("key1")) {
        Object result = store.get("key2");
        store.removeAsync("key2");
    }
    store.replaceAsync("key2", "value3");
    store.clear();
}
```

Infinispan: Features (1)

- Running in cluster
  - auto-sharding (distribution mode)
    - basically "consistent-hashing" (customizable)
    - fixed number of "segments" (like "vnodes" in Riak)
  - replication - master/slave (primary/backup owners)
    - synchronous (write through), asynchronous (write back/behind)

- Persistence
  - originally only memory-based, now fully configurable
    - file system store, JDBC store, LevelDB, JPA cache store,...
  - JBoss marshalling (serialization) of Java objects

Infinispan: Features (2)

- Cache features
  - eviction/expiration (remove objects automatically)
    - either when the cache is full (LRU)
    - or after some time (lifespan of an entry)
  - invalidation mode
    - a special type of cluster mode
    - when a value changes, other nodes are informed that their data is stale
  - L1 cache
    - each node keeps a local cache of key/values retrieved from other nodes

- MapReduce
  - full support of MapReduce processing
    - very efficient since version 7.0

Concurrent Operations

- Full transactional processing
  - Java Transaction API (JTA)
  - X/Open Extended Architecture (X/Open XA)
  - optimistic vs. pessimistic transactions
  - deadlock detection
  - Two-phase commit protocol (2PC)

- Distributed Execution Framework
  - executing a "Callable" on "nodes storing given set of keys"
  - compatible with standard Java Execution Framework

Concurrent Operations (2)

- Multi-version Concurrency Control (MVCC)
  - a technique to solve concurrent access to data
  - faster than strict use of r/w locks
  - popular in many (RDBMS) databases

- For transactions, user can choose isolation levels:
  - READ_UNCOMMITED
    - don’t use transactions at all
  - READ_COMMITED (default)
    - any transaction does see new value immediately after its commit
  - REPEATABLE_READS
    - using MVCC, the transaction does see the same values all the time

Infinispan: Querying

- Additional indexes
  - to provide search over stored values
  - using Hibernate Search technology
  - ...and Lucene

- Vice versa:
  - Infinispan can serve as a distributed storage for Lucene

source: http://infinispan.org/docs/7.0.x/user_guide/user_guide.html

Example: Indexing

```
// A class to be indexed is annotated with @Indexed
// then you pick which fields and how to index them
@Indexed
public class Book {
  @Field String title;
  @Field String description;
  @Field @DateBridge(resolution=YEAR) Date publicationYear;
  @IndexedEmbedded Set<> authors = new HashSet<Author>();
}

public class Author {
   @Field String name;
   @Field String surname;
}
```

source: https://infinispan.org/tutorials/

Example: Searching

Task: Find books on "book on scalable query engines"

```
SearchManager searchManager = Search.getSearchManager(store);

// create a query via Lucene APIs or using builder
QueryBuilder qBuilder =
  searchManager.buildQueryBuilderForClass(Book.class).get();

Query luceneQ = qBuilder.phrase()
  .onField("description").andField("title")
  .sentence("book on scalable query engines").createQuery();

CacheQuery res = searchManager.getQuery(luceneQ, Book.class);

// and there are your results!
List objectList = res.list();
```

source: https://infinispan.org/tutorials/

Agenda

- Embedded local storages
  - LevelDB
    - Local storage for many systems, Log-structured Merge Tree
- Distributed key-value Stores - representatives
  - Riak
    - Basics, Riak Links & Indexes & Riak Search, Internal features
  - Infinispan
    - Basic features, example, advanced features, indexing & searching
- Memory caches
  - Memcached
- Serialization: Protocol Buffers, Apache Thrift

Memory Caches

The typical cache systems are:

- In-memory, distributed key-value stores

- Can be used to speed-up:
  1. Web access to your system
  2. Data access from different components of your system

- Typical features:
  - Limited size, FIFO or LRU algorithms
  - Limited validity of the key-value pair (e.g., 1 hour)

Memory Caches: Representatives

- Memcached
  - 2003, very popular
  - used by FB in early years (MySQL + Memcached)
- Ehcache
  - Java, compatible with javax.cache API
  - Directly storing Java objects into cache
- Hazelcast
  - In-memory data grid written in Java
  - Data evenly distributed among nodes in the cluster

Memcached: Basic Info

- In-memory distributed key-value store
- Initial release date: 2003
  - by Brad Fitzpatrick for LiveJournal
- License: New BSD Licence
- Language: C
- Used by:
  - LiveJournal, Wikipedia, Flickr, WordPress.com, Craigslist

https://memcached.org/ http://db-engines.com/en/system/Memcached

Memcached: Features

- Memcached
  - store small chunks of arbitrary data (strings, objects)
  - keys up to 250 bytes, values up to 1MB
- Typical usage
  - cache results of database calls, API calls, or page rendering
- API is available for most popular languages

Memcached: Architecture

- Client-server architecture
  - Client-side libraries to contact the servers
  - Each client knows all servers
  - Servers do not communicate with each other
- Static sharding
  - The client computes a hash(key) to determine the server
  - Scalable shared-nothing architecture across the servers

Agenda

- Embedded local storages
  - LevelDB
    - Local storage for many systems, Log-structured Merge Tree
- Distributed key-value Stores - representatives
  - Riak
    - Basics, Riak Links & Indexes & Riak Search, Internal features
  - Infinispan
    - Basic features, example, advanced features, indexing & searching
- Memory caches
  - Memcached
- Serialization: Protocol Buffers, Apache Thrift

Data Formats: Text Data

- Structured Text Data
  - JSON, BSON (Binary JSON)
    - JSON is currently number one data format used on the Web
  - XML: eXtensible Markup Language
  - RDF: Resource Description Framework

Data Formats: Binary Data

- Data objects to be stored often originate from memory structures (objects, class instances)

- Before storing, these objects must be serialized
  - Key-value stores can store a binary *value*

- Serialization (marshalling) can be done
  - By your own proprietary (de)serializator
  - Using "standard" language-specific way (Java serialization)
  - Using a cross-language standard: ProtoBuf, Apache Thrift, Apache Avro

Protocol Buffers

- Technique for serializing structured data
- Developed by Google since 2008
  - BSD Licence

- Philosophy:
  1. Define the structure of the data
     - Using an ProtoBuf *interface description language*
  2. Automatically create source code in multiple programming languages for (de)serialization of such data
     - Compilers for Java, C++, Python, JavaScript, PHP, …

http://en.wikipedia.org/wiki/Protocol_Buffers

Protocol Buffers: Example


```
// file: addressbook.proto
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0; HOME = 1; WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

source: https://developers.google.com/protocol-buffers/

Protocol Buffers: Example 2 - Java

- Compile this source by:

```
protoc --java_out=jdir addressbook.proto
protoc --cpp_out=cppdir addressbook.proto
protoc --python_out=pdir addressbook.proto
```

- Result looks like this (Java):
  - you have getters; builder with setters; writeTo(outstream)

https://github.com/jgilfelt/android-protobuf-example/blob/master/src/com/example/tutorial/AddressBookProtos.java

source: https://developers.google.com/protocol-buffers/

Apache Thrift

- Interface definition language
  + binary communication protocol
- Developed by Facebook -> open source (Apache)
- Similar philosophy as ProtoBuf
  - Write data schema once
  - Generate code in multiple languages
- Many languages: C#, C++, Erlang, Go, Haskell, Java, Node.js, OCaml, Perl, PHP, Python, Ruby, Smalltalk

https://thrift.apache.org/ http://en.wikipedia.org/wiki/Apache_Thrift

Apache Thrift: Example


```
enum PhoneType {
  HOME,
  WORK,
  MOBILE,
  OTHER
}

struct Phone {
  1: i32 id,
  2: string number,
  3: PhoneType type
}
```

source: http://en.wikipedia.org/wiki/Apache_Thrift

Apache Thrift: Example


```
service PhoneBook extends shared.SharedService {

   i32 add(1:string num, 2:PhoneType type),

   void remove(1:i32 id),

   oneway void sms(1:string num, 2:string msg)
}
```

source: https://thrift.apache.org/

Apache Avro

- a row-oriented remote procedure call and
- data serialization framework

- serializes data in a compact binary format
- does not require running a code-generation program when a schema changes

source: https://en.wikipedia.org/wiki/Apache_Avro

Apache Avro: Schema Definition


```
{
  "namespace": "example.avro",
  "type": "record",
  "name": "User",
  "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number", "type": ["null", "int"]},
     {"name": "favorite_color", "type": ["null", "string"]}
  ]
}
```

- primitive types
  - null, boolean, int, long, float, double, bytes, and string
- complex types
  - record, enum, array, map, union, and fixed

source: https://en.wikipedia.org/wiki/Apache_Avro

Lecture Summary

- **Key-value** stores are popular for its **simplicity** and **efficiency**
- Most of the real key-value stores provide **additional** functionality to **search** on the **values**
- Besides **distributed** systems, there are local **embedded** stores and in-memory caches
- There are general **frameworks** to provide **serialization** of objects into **binary** data

References

- I. Holubová, J. Kosek, K. Minařík, D. Novák. Big Data a NoSQL databáze. Praha: Grada Publishing, 2015. 288 p.
- Sadalage, P. J., & Fowler, M. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 192 p.
- doc. RNDr. Irena Holubova, Ph.D. MMF UK course NDBI040: Big Data Management and NoSQL Databases
- http://www.slideshare.net/quipo/nosql-databases-why-what-and-when
- https://riak.com/products/riak-kv/
- https://infinispan.org/docs/stable/index.html