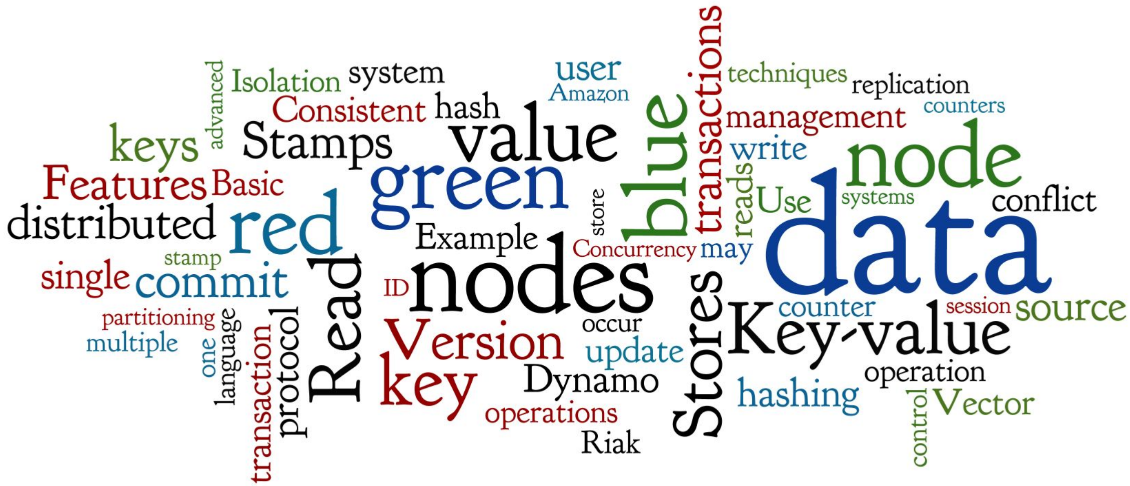


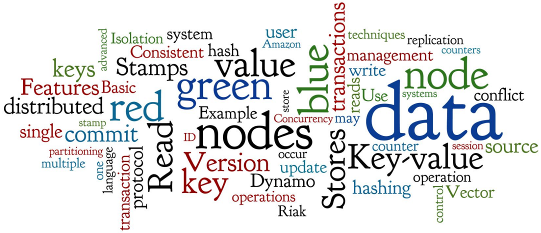
Distributed Key-value Stores

Lecture 4 of *NoSQL Databases* (PA195)

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

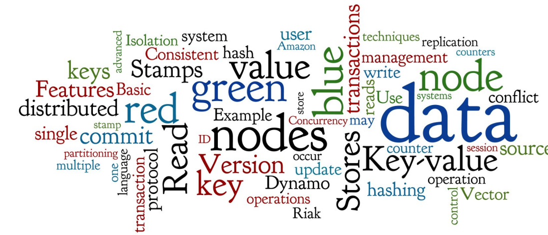
http://disa.fi.muni.cz/vlastislav-dohnal/teaching/nosql-databases-fall-2019/

Agenda



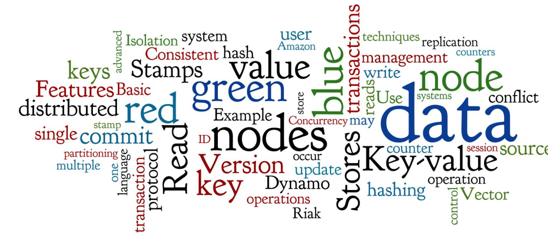
- Fundamentals of Key-value Stores
  - Basic Example: Riak
- Key Techniques of Many Key-value Stores
  - Data Sharding: Consistent hashing + virtual nodes
  - Replica management & consistency (version stamps)
  - Gossip protocols (distributed management of nodes)
  - Transactions: Two-phase commit protocol (2PC); MVCC
- Comparison of K-V Stores and Applicability
  - Features to consider - basic, advanced
  - Modes of communication with the database
  - When (not) to use Key-value Stores

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:
  - Put a value for a key `put(key, value)`
  - Get the value for the key `value:= get(key)`
  - Delete a key-value `delete(key)`

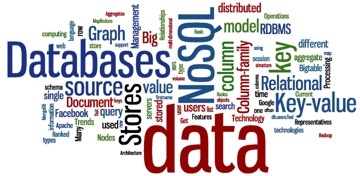
Querying



- 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

Representatives



amazon DynamoDB

riak

levelDB

Infinispan

ORACLE BERKELEY DB 12c

redis

RocksDB

Project Voldemort

MEMCACHED

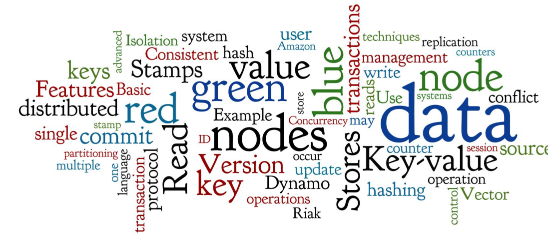
MapDB

hazelcast

EHCACHE

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

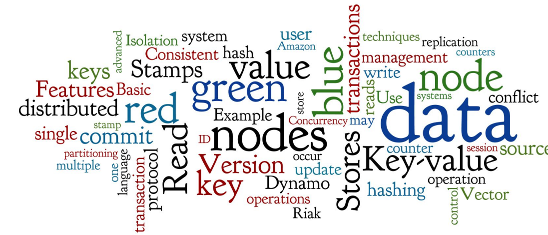
Riak: Basic Information



- Developer: Basho, open source community
  - there is a company behind
- Initial release date: 2009
  - it is not a new (shaky) technology
- License: Apache 2 + commercial enterprise
  - for free, but with option to have a support
- Language: Erlang, C, C++, some parts in JavaScript
  - Efficient; not possible to embed to e.g. Java application
- Server OS: Linux, BSD, Mac OS X, Solaris

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

Riak: HTTP REST API



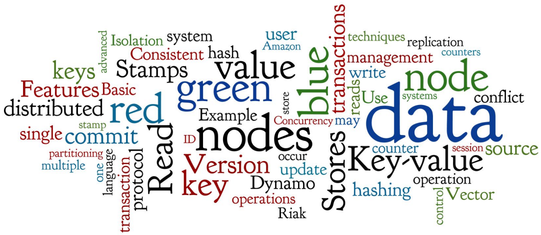
- Riak HTTP API
  - HTTP Restful service (aka HTTP REST)
  - a simple way to define Web services

- Listening on a port and providing services

`http://localhost:8098/`

- Such interface can be directly called from
  - application written in any language
  - client side of the application (e.g. AJAX request)
  - command line (simple scripts), …

Riak: Basic Operations



- We will use `curl -X method URL -d data`
  - command line tool to communicate with server (HTTP(S),...)

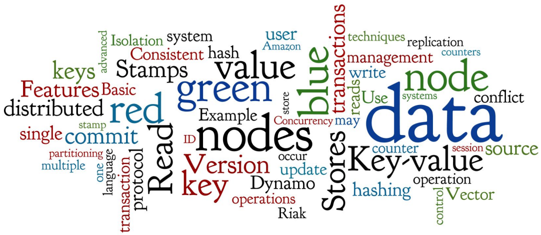
```
curl -X PUT http://localhost:8098/buckets/authors/keys/David
-d '{"name": "David Novák", "affiliation": "MU"}'
```

```
curl -X GET http://localhost:8098/buckets/authors/keys/David
```

{"name": "David Novák", "affiliation": "MU"}

```
curl -X DELETE
http://localhost:8098/buckets/authors/keys/David
```

Management of the Keys



- How to design the key?
  - Provided by the user (natural unique key):
    - shopping cart data (user ID)
    - web session data (with the session ID as the key)
    - user profiles (user ID), …
  - Generated by some algorithm
  - Derived from time-stamps (or other data)

- Expiration of keys
  - After a certain time interval
    - e.g. for caches, session/shopping cart objects,...

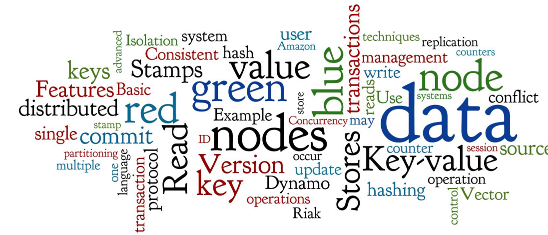
Keys: Buckets (Namespaces)

- keys can be grouped into buckets (namespaces)
  - division of the key space
  - logical differentiation of records by types
  - but physically, all keys are hashed into the same space
- e.g. Riak defines a `location` for each value

  `location = <namespace, key>`

```
put( <namespace, key>, value )
value := get( <namespace, key> )
delete( <namespace, key> )
```

Namespaces: Example



Namespace *User*

Key:	*userID*
Value:	*userProfile* *sessionData* *shoppingCart* • *item 1* • *item 2*

Single namespace:

- works well, if the application often wants to access all data

Namespace *UserProfiles*

Key:	*userID*
Value:	*userProfile*

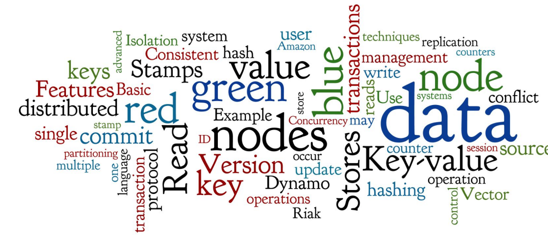
Namespace *Sessions*

Key:	*userID*
Value:	*sessionData*

Namespace *ShoppingCart*

Key:	*userID*
Value:	*shoppingCart* • *item 1* • *item 2*

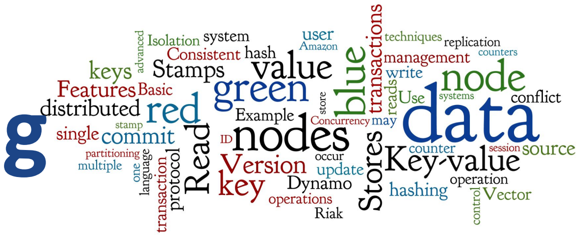
- this is an example of the aggregates-based data modelling



Key Techniques of Key-value Stores

Data Sharding: Consistent hashing + virtual nodes

Key-value Stores: The Beginning

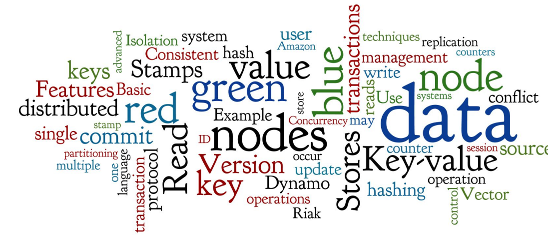


- 2007: Amazon Dynamo paper
  - DeCandia, G. et al. (2007). Dynamo: Amazon's Highly Available Key-value Store. ACM SIGOPS Operating Systems Review, 41(6), pp 205–220.
    - http://dl.acm.org/citation.cfm?id=1294281

- Amazon Dynamo: first fully-fledged distributed key-value store

- Now: DynamoDB - available as paid service
  - http://aws.amazon.com/dynamodb/

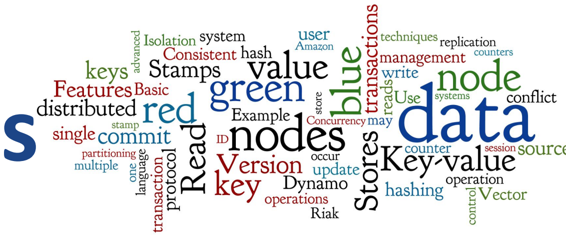
Dynamo and Other Systems



- Amazon Dynamo paper
  - Defined the fundamental challenges and their solutions
  - Laid the foundations for other systems

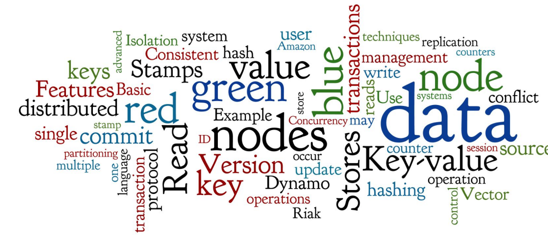
- Other Key-value Stores
  - Each has a slightly different purpose
    - The set of challenges differs
  - Each has a specific set of solutions of these challenges
  - The additional functionality may differ
    - Besides basic put/get/delete operations

Selected Challenges & Solutions

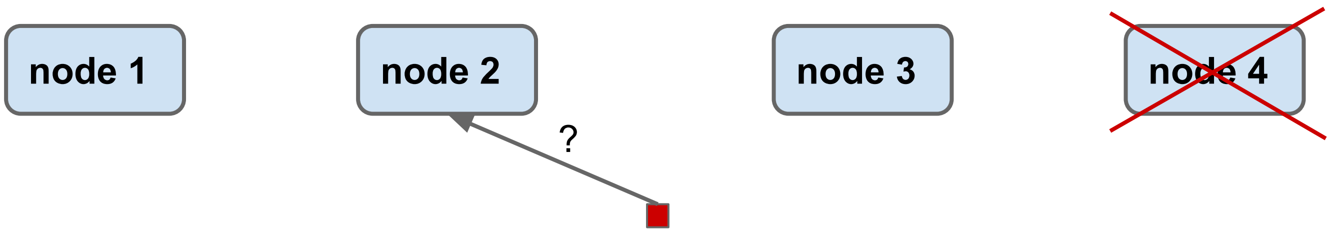


Challenge	Selected Techniques
Data partitioning (sharding)	Consistent hashing
Read scalability & reliability	Data replication
Replica management	Version stamps, vector clocks
Detection of a node join/leave/failure	Gossip protocol (no centralized registry of nodes' membership and liveness)
Concurrency, transactions	Two-phase commit protocol, MVCC

Sharding: Modulo Hashing



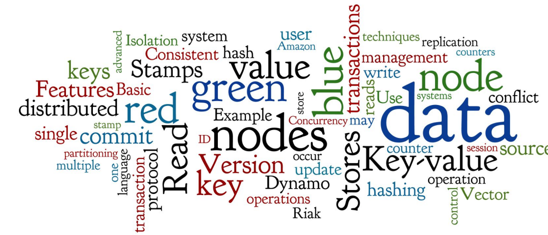
- We want to use `hash(key)` for partitioning of `key-value` pairs to nodes (auto sharding)
- Standard modulo-based hashing:

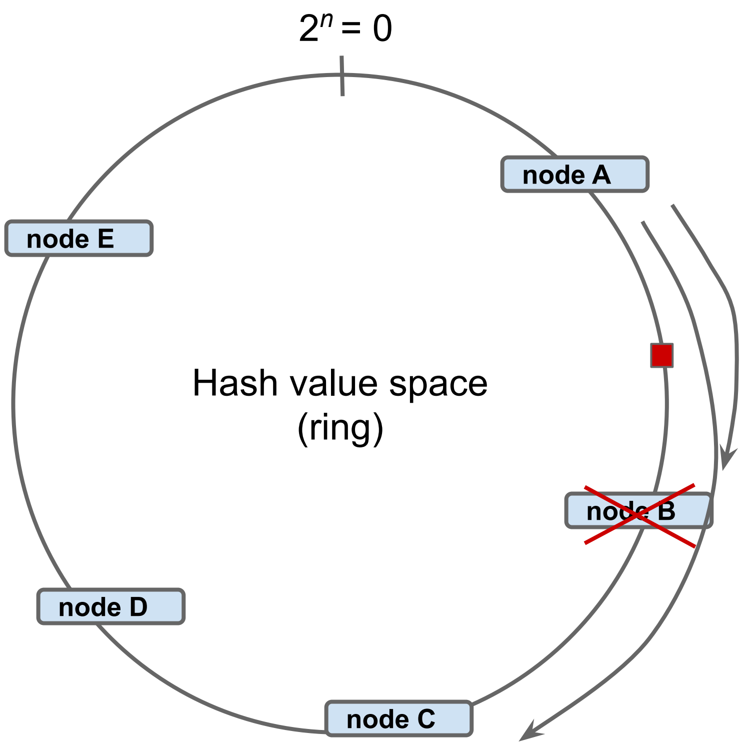


`node(key) = hash(key) mod (#_of_nodes - 1)`

- Recalculate the hashes of all objects, if `#_of_nodes` changes
  - and migrate practically all data objects to different nodes

Consistent Hashing: Principles





node "B" is responsible for interval *A-B*

node "C" is responsible for interval *A-C*

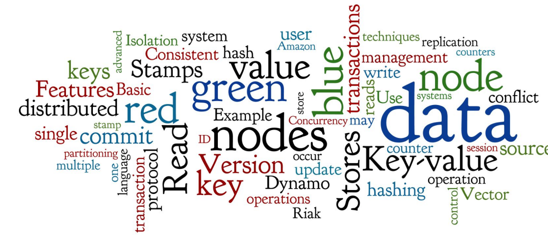
(only data in *A-B* range need relocation)

Use the same hash function for data and nodes

`hash: Keys ->[0,2^n]`

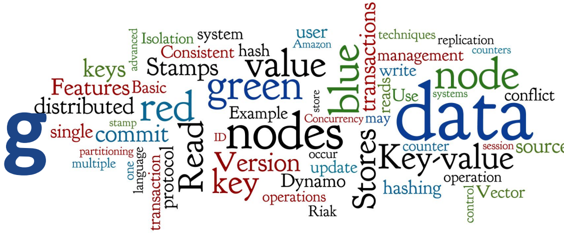
For each hash value, the next clockwise node is "responsible"

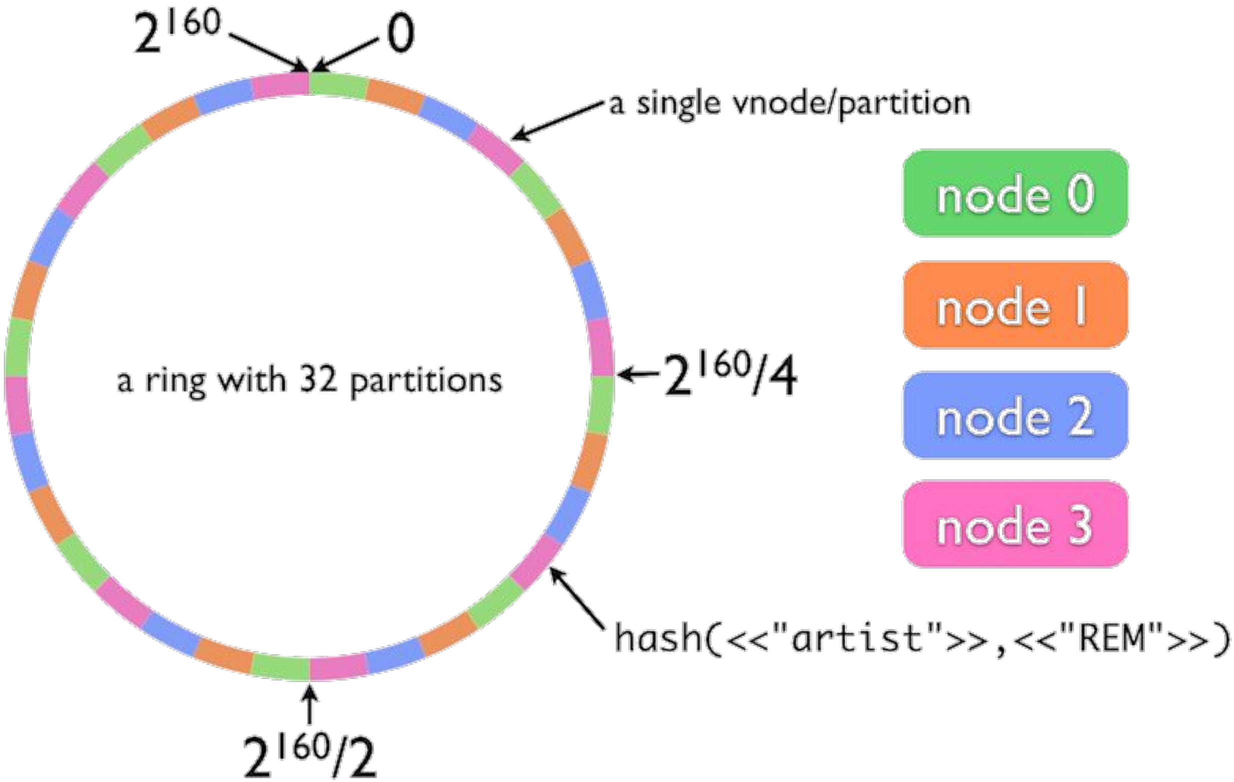
Sharding by Hashing



- Consistent hashing
  - is used in massively distributed systems (like Riak)
- Modulo-based hashing
  - is also used, e.g. in Solr or Lucene
- Modulo hashing is good for keeping data balanced
  - consistent hashing cannot guarantee balanced data
    - especially for low number of nodes
  - it must use different techniques to achieve balancing

Consistent Hash: Data Balancing

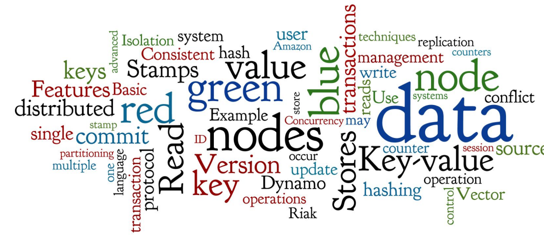




Virtual nodes:

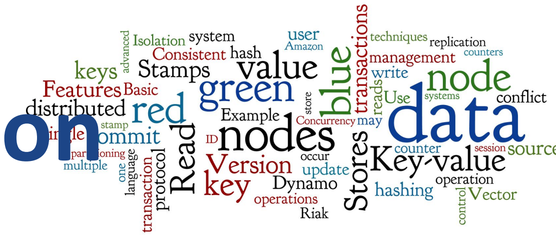
- Q equal-sized partitions (virtual nodes)
- S physical nodes
- Q/S partitions per node
- assumed: $Q >> S$
- result: balanced distribution of data to physical nodes

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

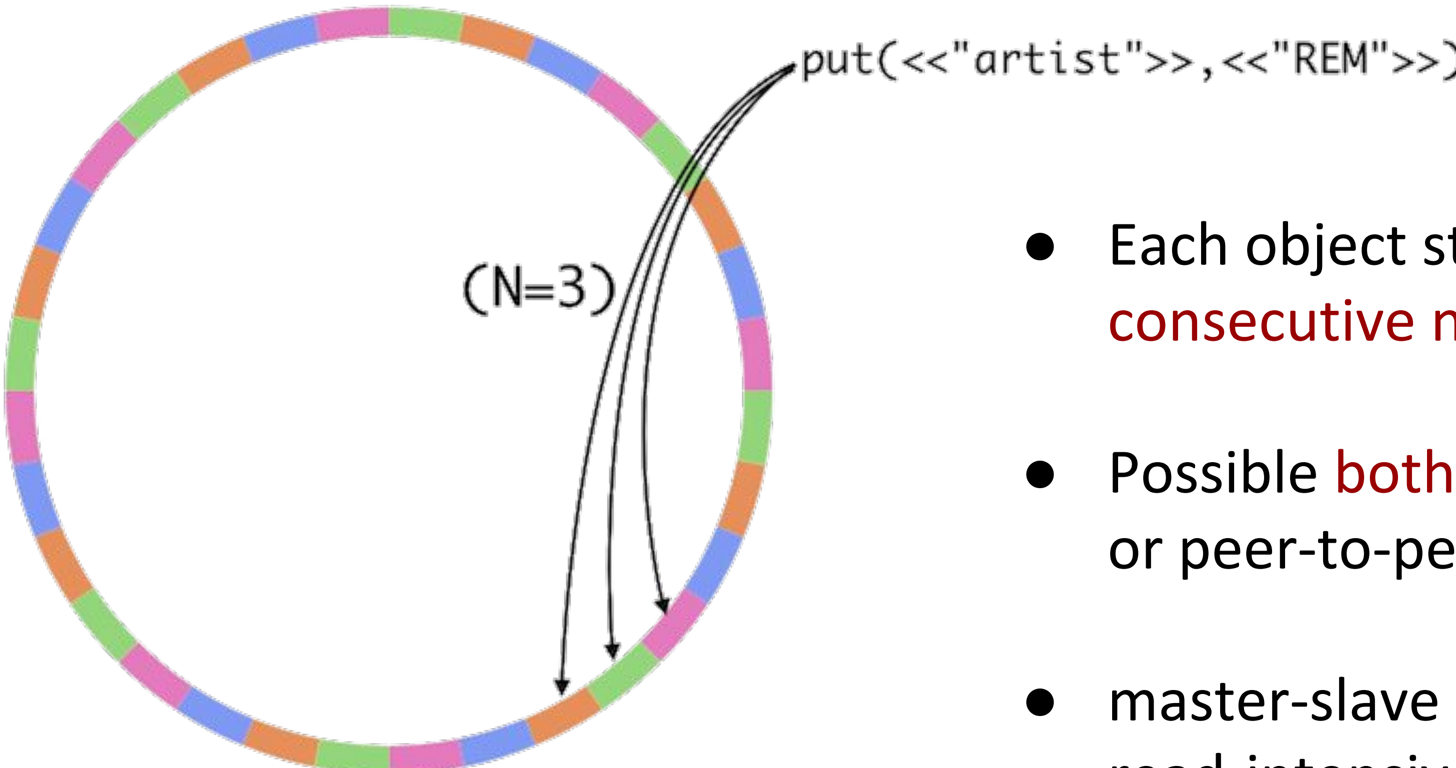


Key Techniques of Key-value Stores

Replica management & consistency



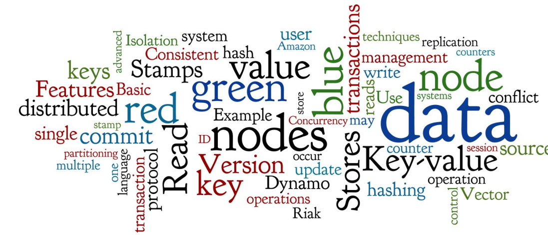
Consistent Hash: Data Replication



- Each object stored at *N* consecutive nodes
- Possible both master-slave or peer-to-peer replication
- master-slave is OK for read-intensive applications

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

P2P Replication: Consistency



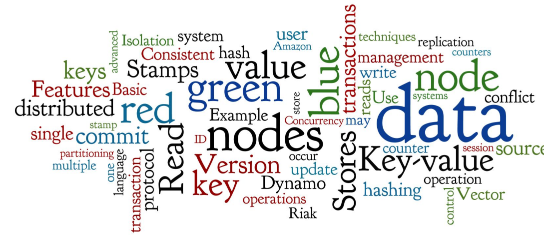
- Recall the concept of quorum
  - **N** = replication factor (typical default: N = 3)
  - **W** = data must be written at least at W nodes
  - **R** = data must be read at least from R nodes

$$W > N/2$$

$$R + W > N$$

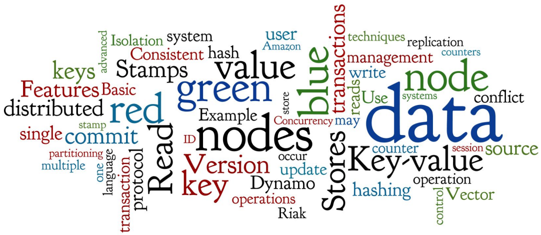
- Example: replication factor **N** = 5, quora **W** = 3
  - Write is reported as successful only when reported as a successful on >= 3 nodes
  - Tolerate **N** – **W** = 2 nodes being down for write operations

Quora per Operation



- The **R**/**W** values can be often set per operation
  - Riak: all / one / quorum / an integer value
  - it is a way to tune efficiency/availability vs. consistency
- example: **N** = 3, quora **W** = 2, **R** = 2
  - value for `key1` is stored on `nodeA, nodeB, nodeC`
  - at least two of them always have the newest value
    - and operation `get(key1)` will always get the newest `value`
- we can set **R** = 1 for operation `value:= get(key1)`
  - meaning: get the value the from any replica, e.g. `nodeB`
  - even though `nodeA` and `nodeC` may have a newer value

A Problem to Solve



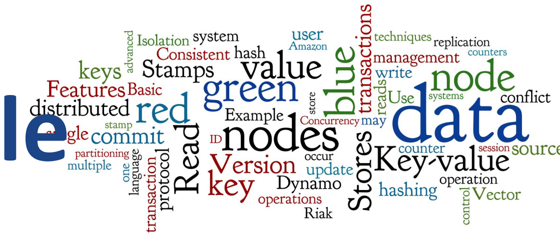
Let’s assume the peer-to-peer replication...

- with write/read quora

...we need to have a mechanisms to:

1. recognize which value for a single record is newer
2. find out that two write operations are concurrent and causing a write-write conflict

P2P Replication Conflict Example



three nodes, initial state: `(key, value1)`

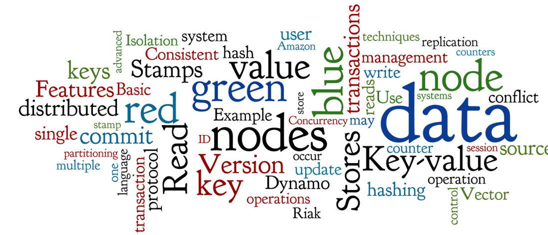
- single update, then sync
  - how to find out which value is newer?
- next update on green
  - how to find out which value is newer?
- two simultaneous updates
  - how to find out that it is a conflict?

Version Stamps

Family of techniques: avoid/detect update conflicts

- Version stamp in general:
  - A field created for each record
  - The stamp changes every time the data record changes
- Basic usage (also in centralized system):
  - A client reads the stamp together with the record
  - When later updating the record, the stamp is sent back together with the new value and checked
  - If the stamp differs from the actual stamp => conflict

Constructing Version Stamps



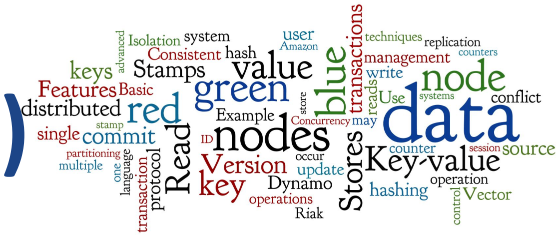
There are several ways to construct the stamps:

1. **counter** - incremented after each record update
   - **pros**: it is clear, which version is newer
   - **cons**: duplications must be avoided (single master?)

2. **GUID** - a large unique random number
   - **pros**: anybody can generate them (client)
   - **cons**: cannot be checked for recentness

3. **Hash** from the data
   - **pros**: anybody can generate it, is deterministic
   - **cons**: cannot be checked for recentness

Constructing Version Stamps (2)



4. **Timestamps**
    - **pros**: recentness like counters, a single master not needed
    - **cons**: clock synchronization, sufficient granularity needed

Combination is worth:

- **counter + hash**:
    - **counter** = recentness comparison
    - **hash** = if two updates appear concurrently on two servers (with the same counter), the hash identifies the conflict

Version Stamps on Multiple Nodes

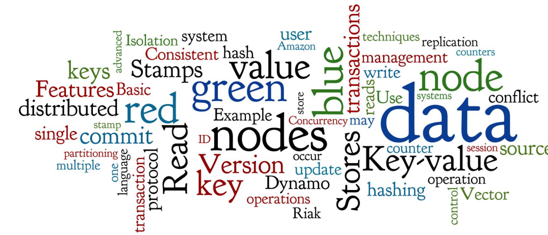
- **If** there is a **single master**, everything works well
- **Peer-to-peer** replication:
  - **Any** peer can process **update**
  - **When** contacted for **update**, the peer must **reply** to the client **immediately** after storing the new value
    - it **cannot wait** until all peers **commit** the update (2-phase commit protocol)
- Objective: A distributed **algorithm** that would
  - reliably **detect** write-write **conflict**
  - **balance** between write **performance** and conflict **prevention**
    - or allow the user to balance it

Vector Stamps Algorithms

Vector stamps

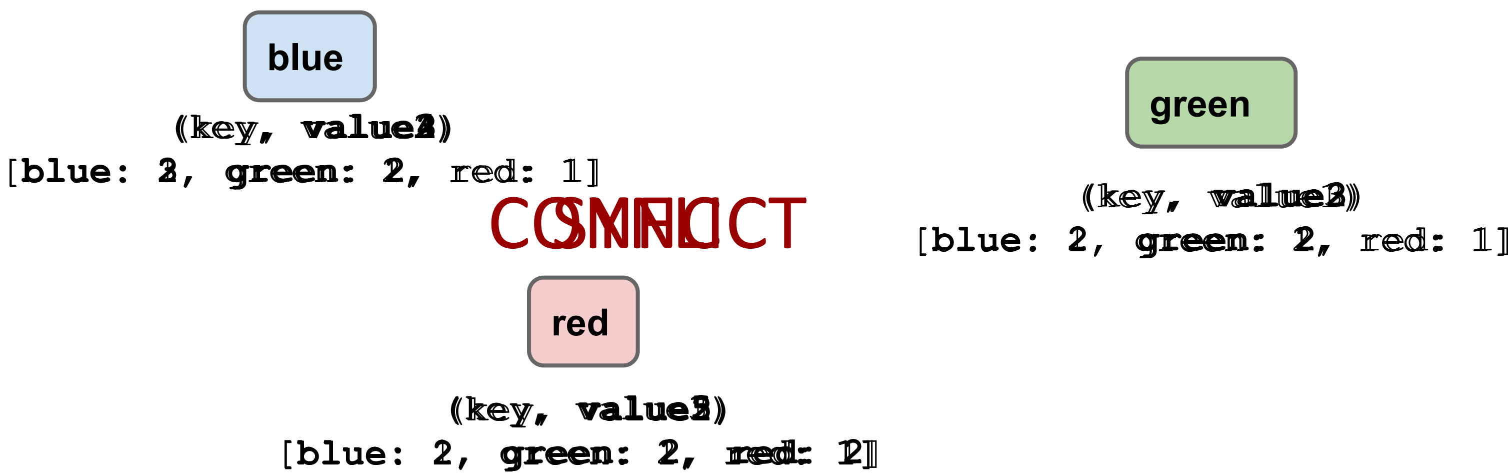
- Family of algorithms for generating a partial ordering of events in a distributed system and detecting “conflicts”.
- Each node has its own counter
  - for each data item
  - The node’s counter increments when its value is updated
- Each node keeps a counter vector with counters of all nodes
  - The nodes exchange their values
- Each node uses the counter vectors to determine
  - which value is new
  - if there is a conflict

Vector Stamps: Example

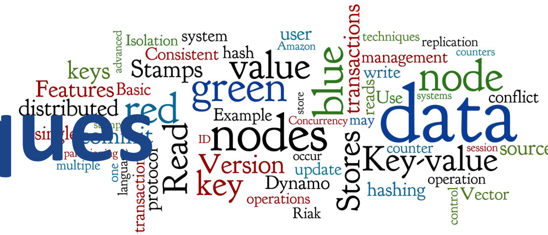


three nodes, initial state: (key, value1) [blue: 1, green: 1, red: 1]

- single update, then sync - the stamp order is clear
  - [blue: 1, green: 1, red: 1] is older than [blue: 2, green: 1, red: 1]
- next update on green
- two simultaneous updates
  - [blue: 3, green: 2, red: 1] cannot be compared to [blue: 2, green: 2, red: 2]



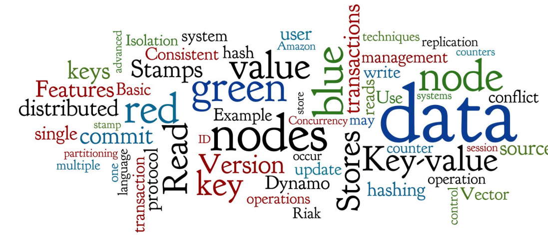
Vector Stamps: Specific Techniques



- Specific techniques differ in the way they communicate

- Widely used techniques (and their variants)
  - **Lamport timestamps** (1987)
  - **Vector clocks** (used by Dynamo, etc.)
    - counters updated whenever nodes communicate
    - last value can be retrieved only during reads
  - **Version vectors**
  - **Matrix clocks**
  - ...

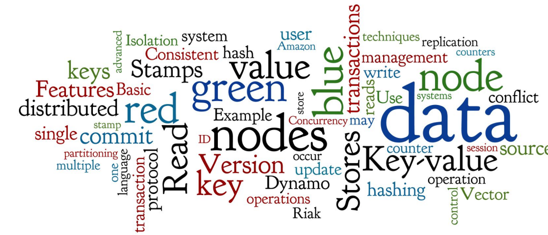
Conflict Resolution



- There are three general ways to resolve conflicts
  - (reconcile differences between copies of distributed data)
  - this process is often known as anti-entropy

1. Write repair
   - The correction takes place during a write operation
2. Read repair
   - The correction is done when a read finds an inconsistency
     - Optimistic strategy, read operation is slowed down
3. Asynchronous repair
   - The correction is done as separate operations
   - AKA active "anti-entropy"

Gossip Protocols

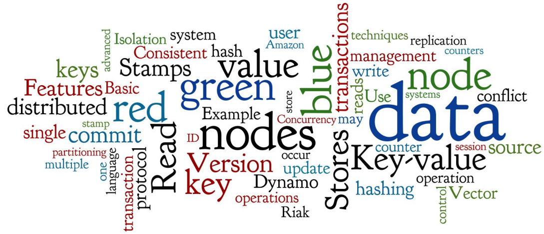


A set of distributed protocols

- Each node periodically sends its current info
  - To a randomly-selected peer
  - The peers keep the newer info

In distributed NoSQL databases, gossip is used for

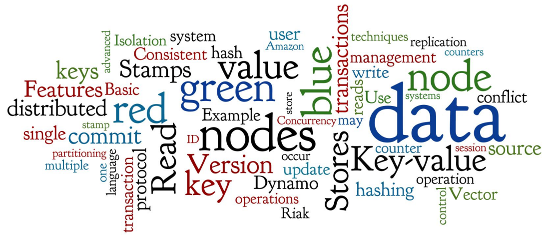
- Spreading information about current state
  - of the entering/leaving/failing nodes
  - asynchronous reconciling of conflicts (anti-entropy)
  - other properties, ...



Key Techniques of Key-value Stores

Distributed Transactions

Transactions

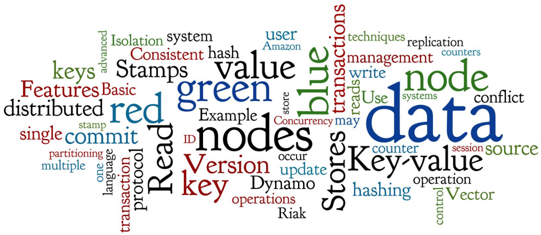


Transaction = a sequence of atomic operations that form one logical operation on the database.

- Some of the distributed key-value stores enable full transactional processing

- The following techniques are key:
  - Two-phase commit protocol (2PC)
    - Atomicity of transaction: either all operations (commit) or none (rollback)
  - Multi-version concurrency control (MVCC)
    - Levels of isolation of transactions

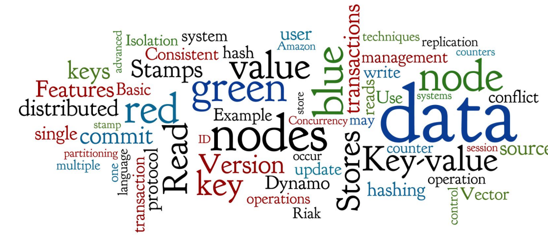
Levels of Isolation (1)



- The transactions should be “isolated”
  - Isolation: property that defines how/when results of one operation become visible to other concurrent operations.

- We recognize four levels of isolation
  - We talk about different "read phenomena" (see below)

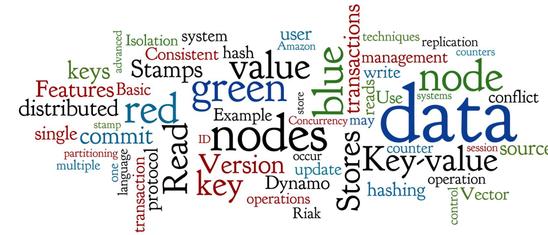
Levels of Isolation (2)



1. **READ UNCOMMITTED:** Operation can access uncommitted changes made by other transactions.
   - It suffers phenomenon "**dirty read**" - a transaction reads uncommitted values that is later rolled back.

2. **READ COMMITTED**: If one transaction commits a value, other transactions will read it immediately.
   - It suffers phenomenon "**non-repeatable read**" - if a transaction reads the same record twice, the second read has a different result.

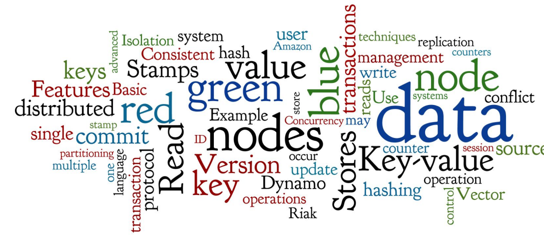
Levels of Isolation (3)



3. **REPEATABLE READS**: Multiple reads of the same record/key issued within the same transaction will always return the same value.
   - It suffers phenomenon "**phantom read**" - if a transaction does two identical "range queries", and the collection of rows returned by the second query differs from the first.

4. **SERIALIZABLE**: All transactions occur in a completely isolated fashion as if executed serially.
   - None of the read phenomena may occur.

Levels of Isolation (4)

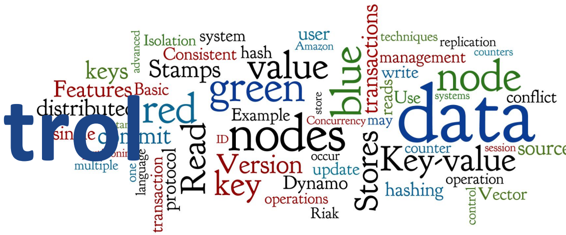


Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur
Read Committed	-	may occur	may occur
Repeatable Read	-	-	may occur
Serializable	-	-	-

Snapshot isolation is still not the serializable level! Assume two records in a table - black and white. T1 changes all blacks into whites and T2 vice versa. If you run them simultanously, in snapshot isolation you will end up with swapped colors. In serializable, you will have either all blacks or all whites!

source: http://en.wikipedia.org/wiki/Isolation_(database_systems)

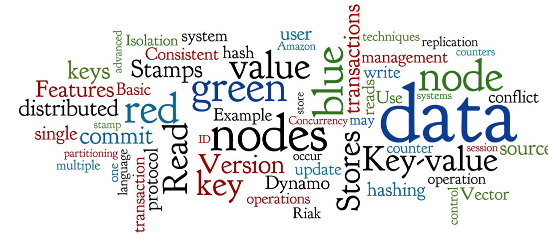
Multi-version Concurrency Control



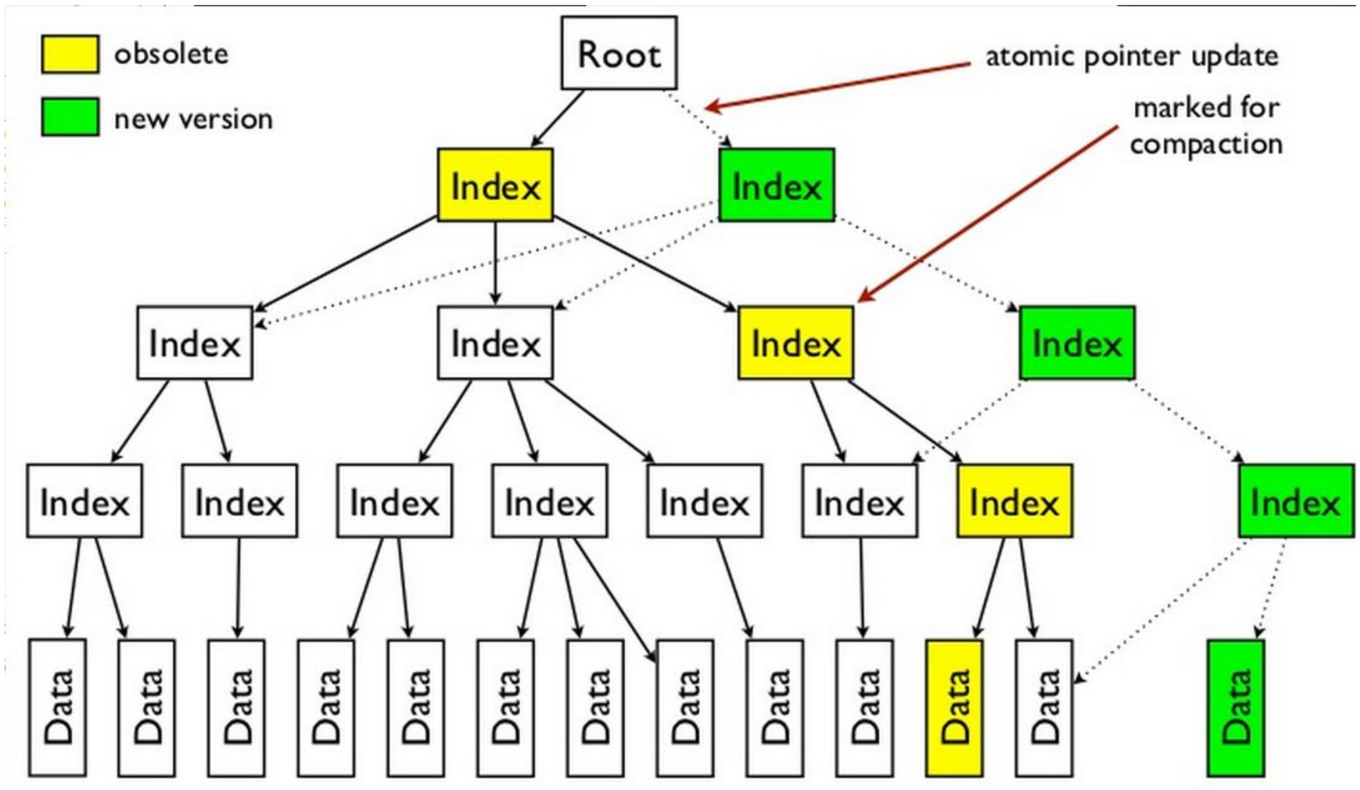
- Multi-version Concurrency Control (MVCC)
  - a technique to solve concurrent access to data
  - faster than strict use of r/w locks
  - popular in many (RDBMS) databases

  - if one transaction is writing and the other is reading, the system can create another version of the data
    - each transaction sees a snapshot of the data at a particular instant in time

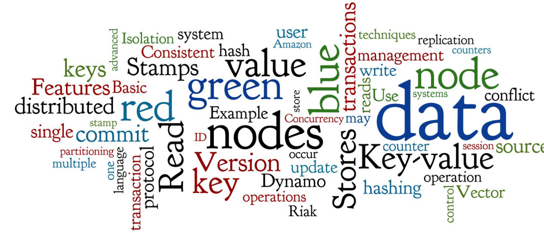
source: http://en.wikipedia.org/wiki/Multiversion_concurrency_control



MVCC: Example



MVCC and Isolation Levels

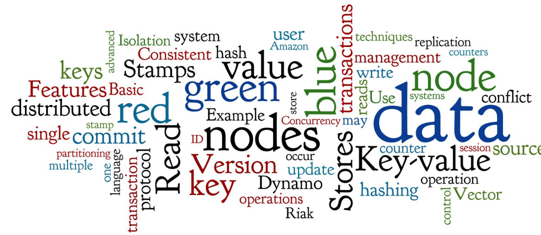


- MVCC cannot ensure full SERIALIZABILITY
  - skew write (write skew anomaly)
    - two transactions concurrently read an overlapping data set, concurrently make disjoint updates and finally concurrently commit, neither having seen the update performed by the other

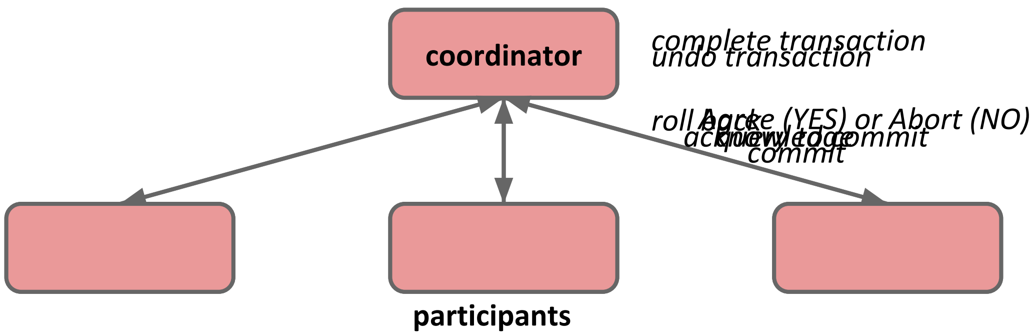
- For instance, in Infinispan user can choose
  - READ_UNCOMMITED
    - don't use transactions at all
  - READ_COMMITED (default)
  - REPEATABLE_READS
    - using some version of JBoss MVCCEntry

source: http://en.wikipedia.org/wiki/Snapshot_isolation http://infinispan.org/docs/7.0.x/user_guide/user_guide.html

Two-phase Commit Protocol



- 2PC: Distributed algorithm
  - coordinating all participants in a distributed transaction
  - on whether to commit or abort (roll back) the transaction
    - it's a special type of consensus protocol

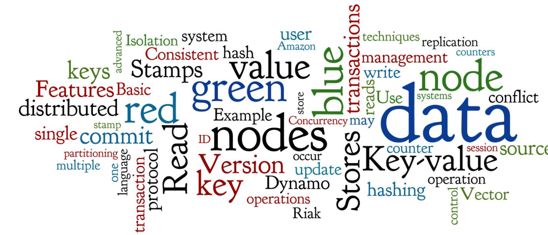


1. Execute transaction up to COMMIT phase / 1. Complete operation / 1. Undo operation
2. Write entry to UNDO and REDO logs / 2. Release locks

1. Commit request phase (voting phase)
2. Commit phase
   a. SUCCESS (agreement from all)
   b. FAILURE (abort from any)

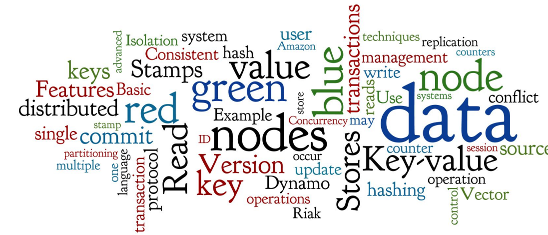
Comparison of K-V Stores & Applicability

K-V Stores: Suitable Use Cases



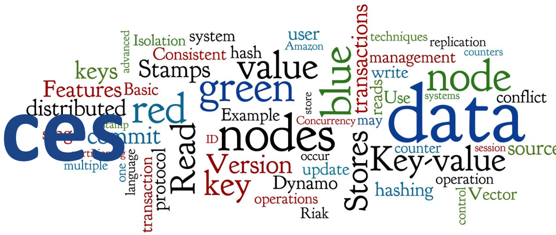
- Storing Web Session Information
  - Every web session is assigned a unique session_id value
  - Everything about the session can be stored by a single PUT request or retrieved using a single GET
  - Fast, everything is stored in a single object
- User Profiles, Preferences
  - Every user has a unique user_id/user_name + preferences (language, time zone, design, access rights, …)
  - As in the previous case: Fast, single object, single GET/PUT
- Shopping Cart Data
  - Similar to the previous cases

K-V Stores: When Not to Use



- Relationships among Data
  - Relationships between different sets of data
    - Some key-value stores provide link-walking features
- Multi-operation Transactions
  - Saving multiple keys
    - Failure to save any of them → revert or roll back the rest of the operations
- Query by Data
  - Search the keys based on something found in the value part
    - Additional indexes needed (some stores provide them)
- Operations by Key Sets
  - Operations are limited to one key at a time
    - No way to operate upon multiple keys at the same time

K-V Stores: Features & Differences



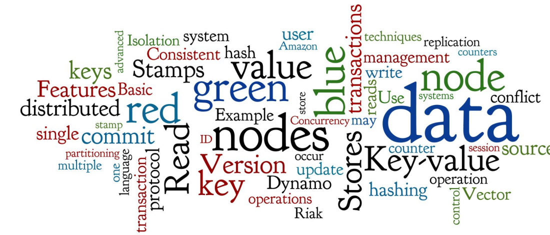
Dozens of key-value stores - how to choose?

1. **Basic** information
   - programming language, license etc.

2. **Internal** Features
   - how are certain principles implemented
   - which influences performance/security/reliability/etc.

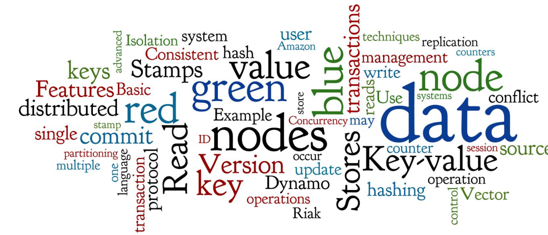
3. **Advanced** (User-visible) Features
   - what “advanced” features does the store provide
     - besides store/get/delete operations

Basic Informations



- **Developer**
  - important information, if there is a company behind
- Initial release **date**
  - is it a hot new (shaky) technology or more stable one?
- **License**
  - Open Source - GNU GPL, Apache, supported version?
- Implementation **language**
  - most often: C/C++, Java, Erlang
- Server **operating** systems
  - usually: Linux, BSD (OS X, MS Windows)

Internal Features (1)

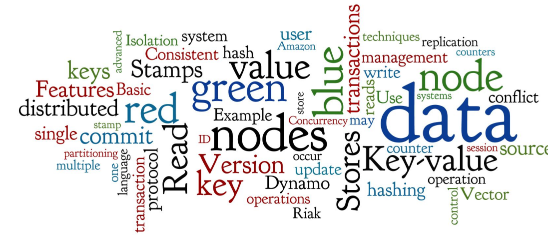


- **Durability**
  - if the system supports data persistence
  - and how is it done (storage models)

- Data Partitioning (**Sharding**)
  - if the system supports (semi)-automatic data sharding

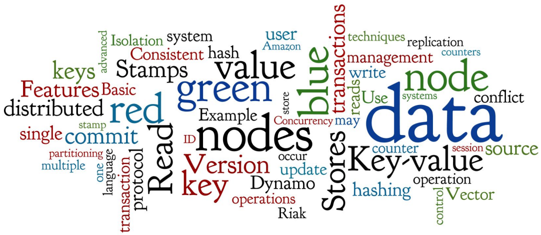
- Data **Replication**
  - replication can speed up read/write and provide reliability
  - master-slave, P2P, R/W quora, version control, etc.

Internal Features (2)



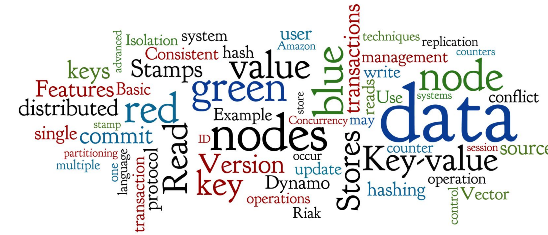
- **Concurrency** control
  - does the system allow concurrent accesses
  - and how is it managed (solved conflicts)
- **Cluster** topology **management**
  - is there a centralized repository of participating nodes
  - or some Gossip protocol
- Node/communication **failure management**
  - how fault-tolerant is the system
  - permanent failure recovery
- **User** concepts
  - any support for user-based access control

Advanced Features (1)



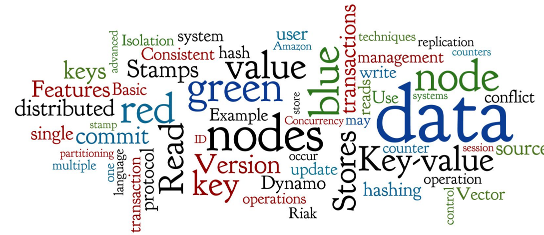
- **Data model**
  - Is the “value” really an unstructured BLOB
  - or are some advanced structures supported
    - e.g. Redis: strings, hashes, lists, sets and sorted sets
- Secondary **Indexes**
  - for efficient access of the data by the values
  - e.g. interval indexes, Lucene-like indexes for full-text search
- **Foreign** keys
  - or other links between data (keys and values)

Advanced Features (2)



- Distributed **Transactions** Management
  - is implemented any concept of transaction management
  - like X/Open XA   (eXtended Architecture)
- **Map-Reduce** processing
  - is available some distributed operation execution like M-R
- **Triggers**
  - procedures started automatically when something happens

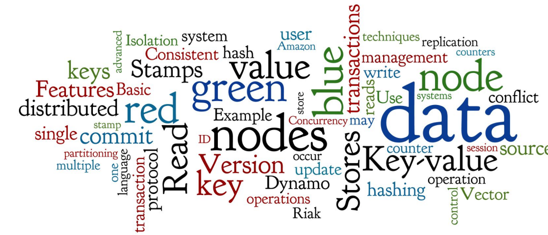
Communication Modes



There are three basic communication modes

1. Via some **web-service interface**
   - HTTP REST service, SOAP
   - usually fast, callable from many clients/libraries/languages
2. Specific language **connector**
   - library in the language of my application
   - may be slower but comfortable
3. **Embedded** to my application
   - the database system runs within the application process
   - requires compatible (the same) programming language

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.
- RNDr. Irena Holubova, Ph.D. MMF UK course NDBI040: Big Data Management and NoSQL Databases