Northeastern University - Seattle

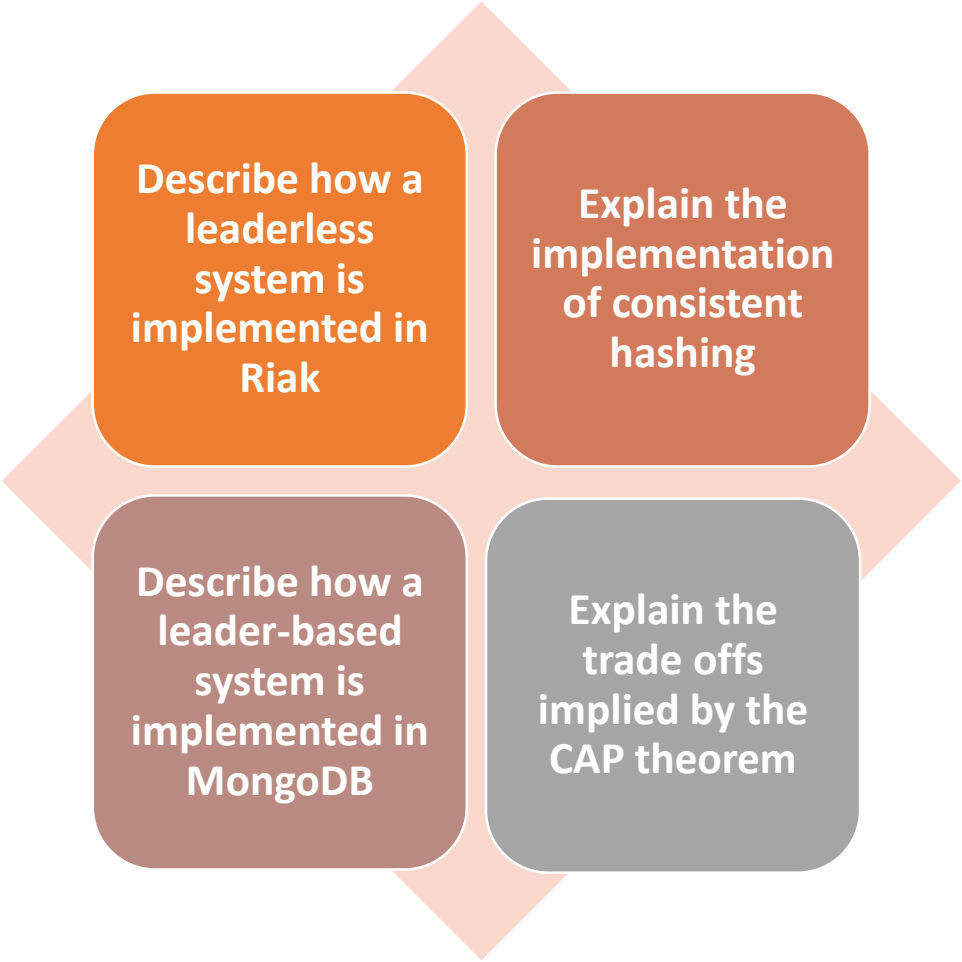
CS6650 Building Scalable Distributed Systems

Professor Ian Gorton

Building Scalable Distributed Systems

Week 6 – NoSQL Databases

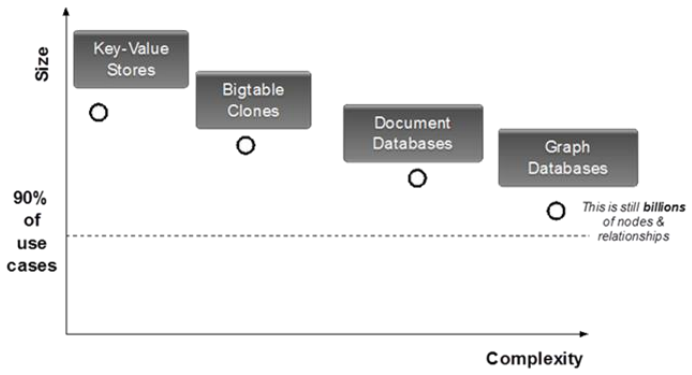
Learning Objectives



Outline

- Riak architecture
- Consistent hashing
- MongoDB architecture
- CAP Theorem

Scalability and NoSQL Databases

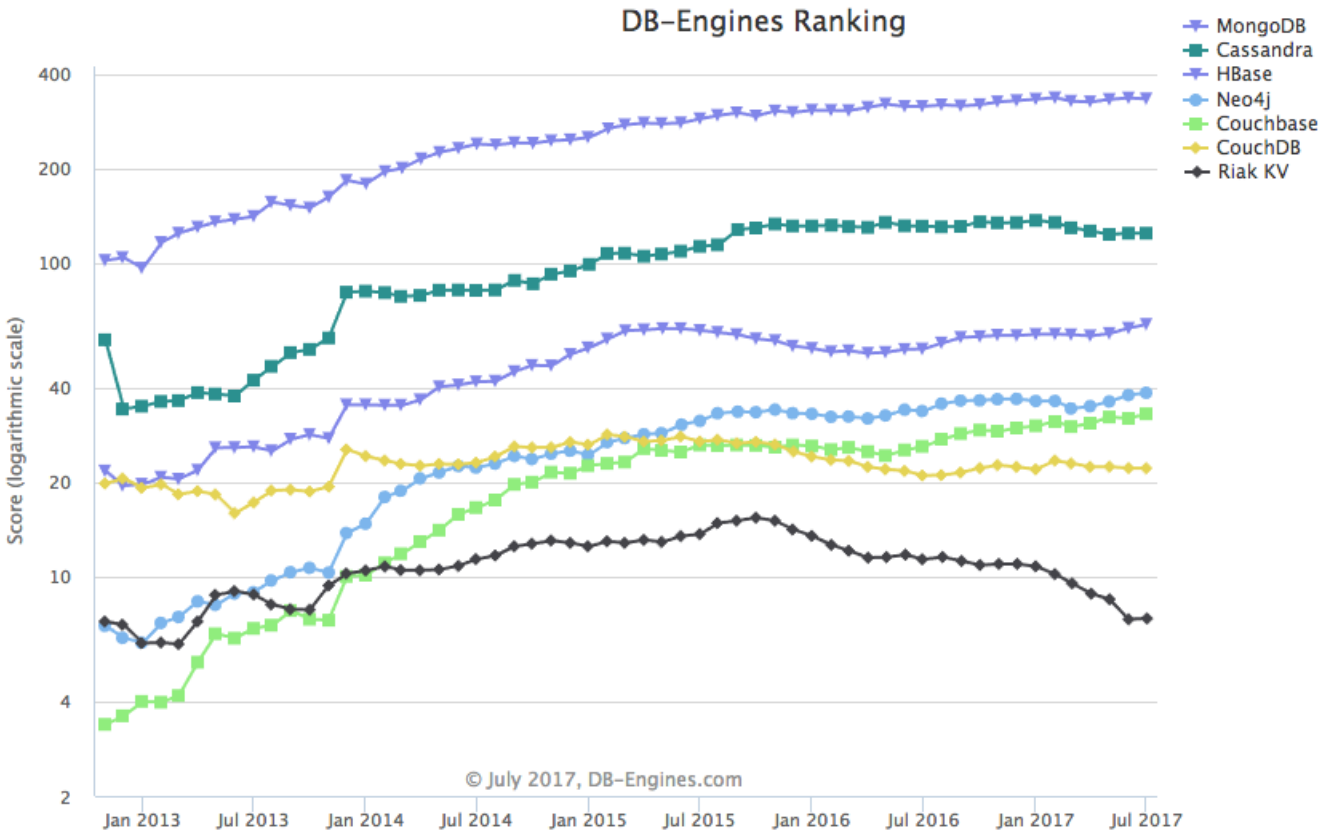


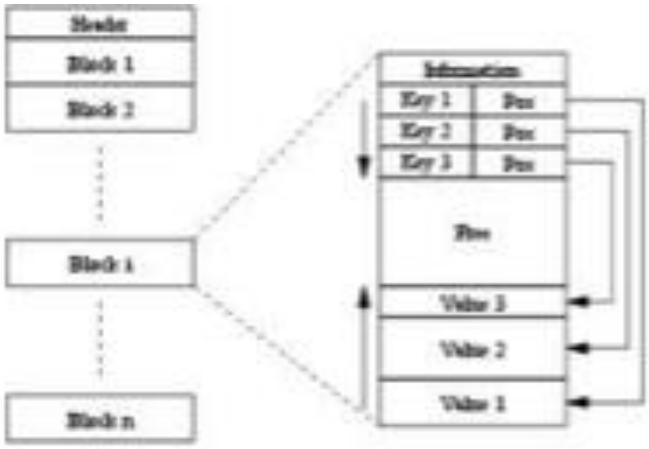
- Each model has strengths and weaknesses that directly effect scalability
  - Data model
  - Query model
  - Consistency
  - Scalability

Riak

Key Value Store

NoSQL Database Rankings 2017





Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, et.al., SOSP '07

Key-Value Databases

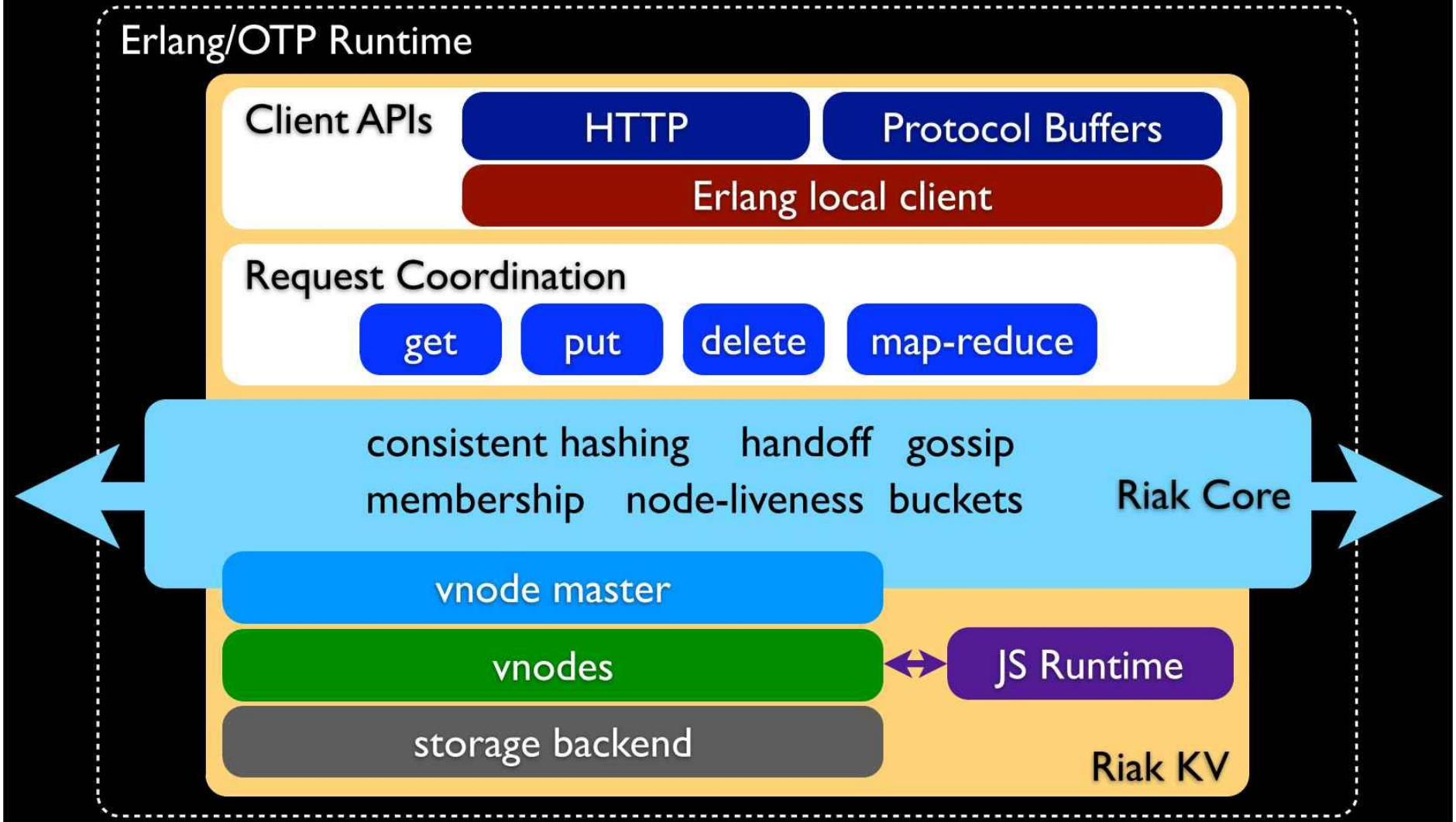
- Simple hash table
  - Associates keys with opaque data items
  - Some implementations support associated metadata to facilitate non-key queries
- Easy to scale horizontally
  - Distributed hash table
  - Consistent hashing
- Object versioning
  - Handle conflict resolution

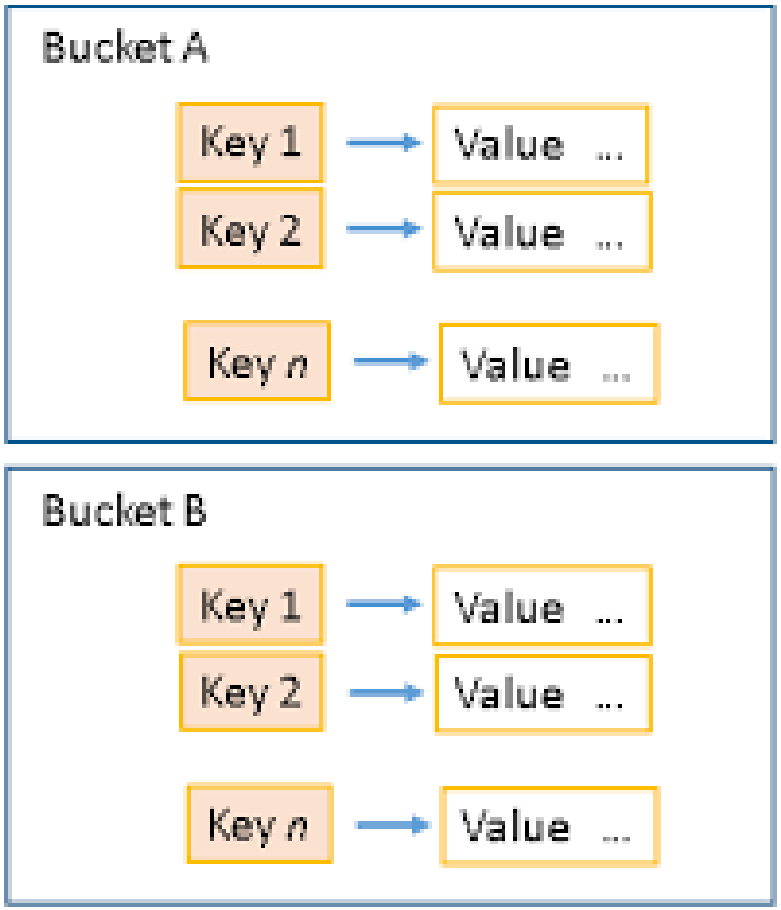
Riak KV (v2.9.0)

- Implementation based on Amazon dynamo paper
- Distributed NoSQL key-value data store
  - high availability,
  - fault tolerance,
  - operational simplicity
  - Scalability
- First released by Basho in 2009
- 2017 Basho went bust, moved to community model (v2.2.5)
- Latest version is 2.9 (2019)

https://www.infoq.com/presentations/riak-dynamo

Riak Architecture





Key Value Store

- Schema-less data model
- Buckets provide a namespace for keys
  - Analogous to a relational table
- Lookup by key
- Values can be anything and don't need to be homogenous – application defined
  - Text
  - Image
  - Data structure
  - JSON
  - etc

Creating Objects

```
// create econnection
RiakCluster cluster = setUpCluster();
RiakClient client = new RiakClient(cluster);
// create plain text object
RiakObject quoteObject = new RiakObject()
    .setContentType("text/plain")
    .setValue(BinaryValue.create("You're dangerous, Maverick"));
Namespace quotesBucket = new Namespace("quotes");
Location quoteObjectLocation = new Location(quotesBucket,
"Icemand");
StoreValue storeOp = new StoreValue.Builder(quoteObject)
    .withLocation(quoteObjectLocation)
    .build();
// write object to Riak
StoreValue.Response response = client.execute(storeOp);
```

Reading Objects

```
FetchValue fetchOp = new FetchValue.Builder(quoteObjectLocation)
        .build();
RiakObject fetchedObject =
client.execute(fetchOp).getValue(RiakObject.class);
assert(fetchedObject.getValue.equals(quoteObject.getValue()));
```

Updating Objects

```
fetchedObject.setValue(BinaryValue.create("You can be my wingman any time."));
StoreValue updateOp = new StoreValue.Builder(fetchedObject)
        .withLocation(quoteObjectLocation)
        .build();
StoreValue.Response updateOpResp = client.execute(updateOp);
```

Deleting Objects

```
DeleteValue deleteOp = new
DeleteValue.Builder(quoteObjectLocation)
        .build();
client.execute(deleteOp);
```

Complex Objects

```
Book mobyDick = new Book();
modyDick.title = "Moby Dick";
mobyDick.author = "Herman Melville";
mobyDick.body = "Call me Ishmael. Some years ago...";
mobyDick.isbn = "1111799723";
mobyDick.copiesOwned = 3;
```

Complex Objects

```
Namespace booksBucket = new
Namespace("books");
Location mobyDickLocation = new
Location(booksBucket, "moby_dick");
StoreValue storeBookOp = new
StoreValue.Builder(mobyDick)
        .withLocation(mobyDickLocation)
        .build();
client.execute(storeBookOp);

// if we read the value back, we get:
{
  "title": "Moby Dick",
  "author": "Herman Melville",
  "body": "Call me Ishmael. Some years
ago...",
  "isbn": "1111979723",
  "copiesOwned": 3
}
```

Riak Replication

- Riak uses leaderless replication of objects:
  - Favors availability over consistency
- Can be configured on a per bucket basis
  - Using bucket properties
- Replication controlled by n_val
  - default = 3
  - Every object in a bucket replicated to three different nodes, creating three replicas of the object.
  - Objects *hashed* by key onto different nodes

```
riak-admin bucket-type create custom_props '{"props":{"n_val":5,"r":3,"w":3}}'
riak-admin bucket-type activate custom_props
```

Bucket Properties

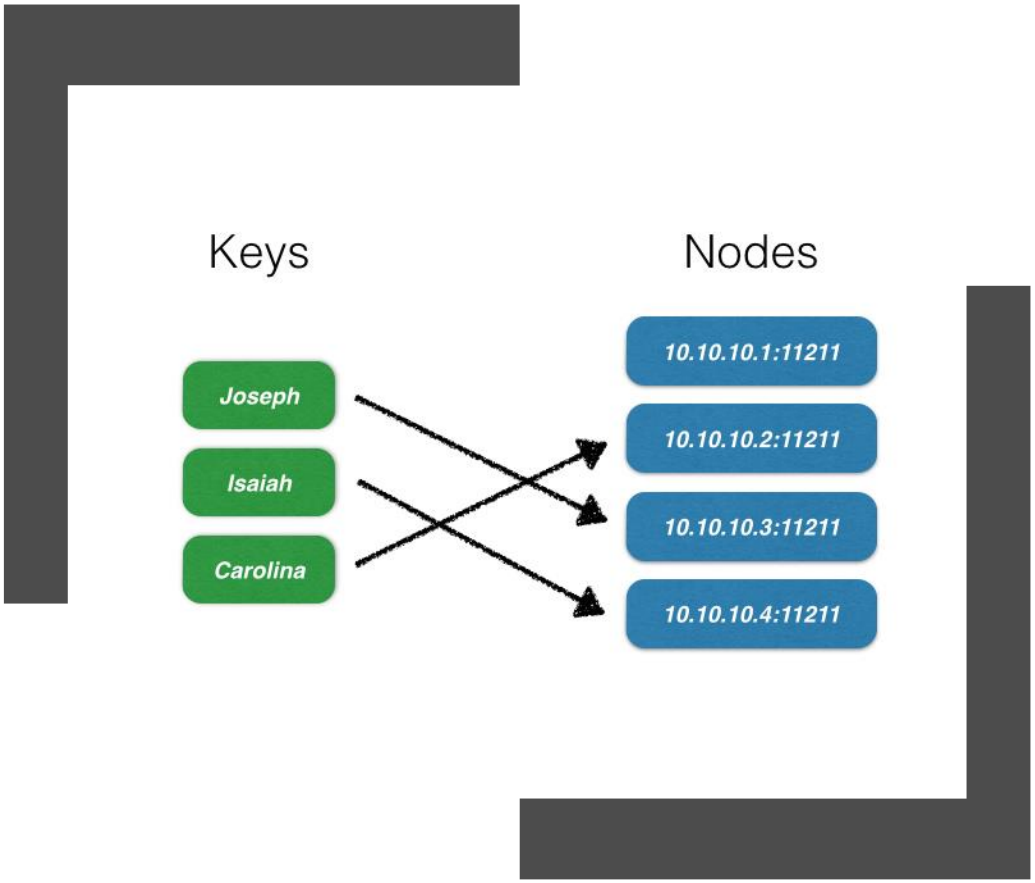
Parameter	Common name	Default value	Description
n_val	N	3	Replication factor, i.e. the number of nodes in the cluster on which an object is to be stored
r	R	quorum	The number of servers that must respond to a read request
w	W	quorum	Number of servers that must respond to a write request

Does this Sound Familiar?

Riak Replication

- trade-off between **data accuracy** and **client responsiveness**.
  - i.e. consistency and latency
- Reads sent to all replicas, need r to respond
- Writes sent to all replicas, completes when w nodes respond to a write
- Also a DW value, specifies how many nodes must write to disk (not just respond)

Hashing in Distributed Databases

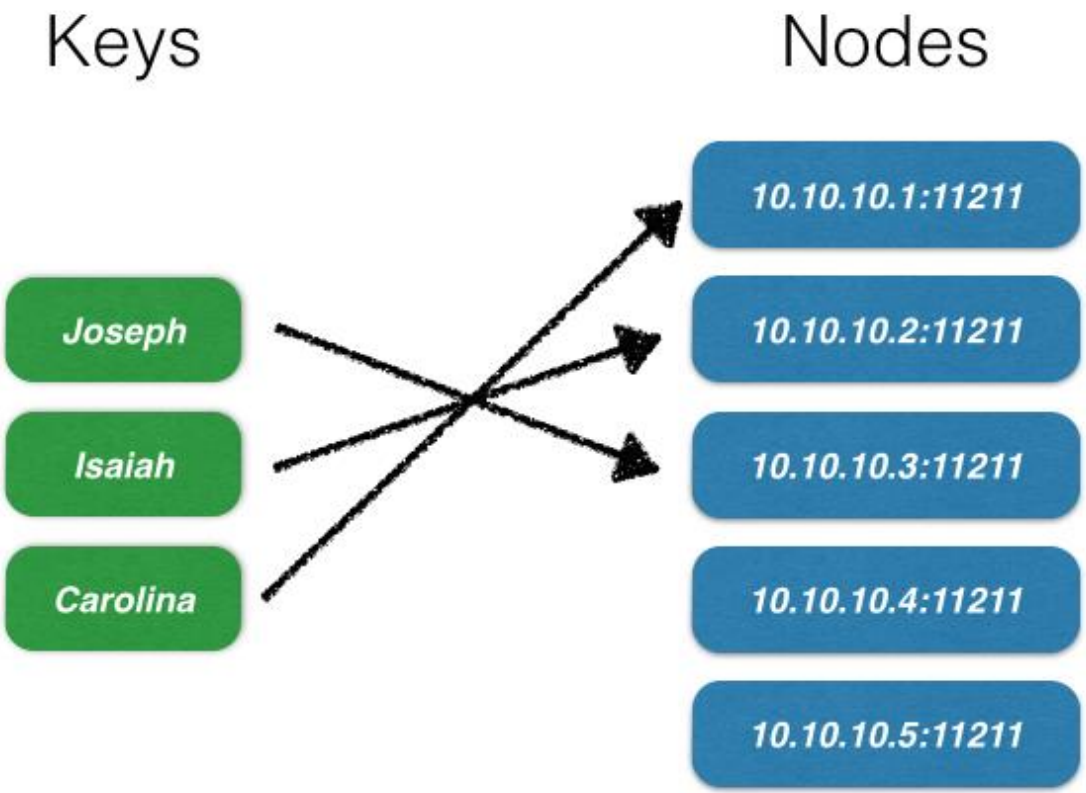


- Hashing effectively distributes keys across nodes
- If we have N nodes:
  - Server = hash(key) mod N

Hashing in Distributed Databases

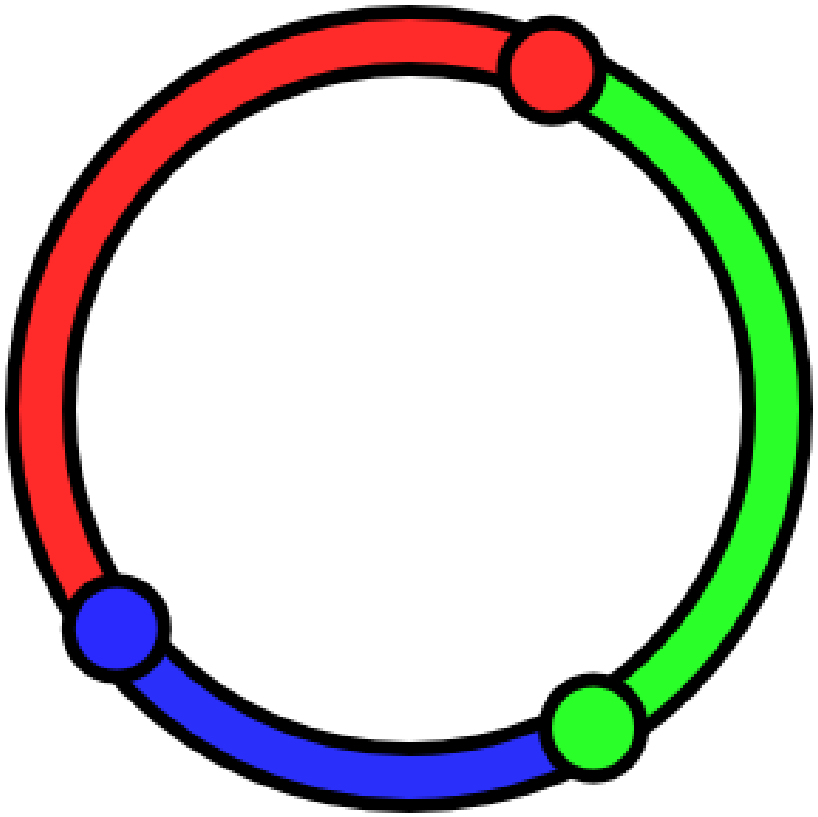
- In a scalable system, we will need to add more nodes as data size grows
- What happens?

Hashing in Distributed Databases



Hashing in Distributed Databases

- Adding servers causes nearly every key to be relocated
  - Bad for caches
  - Catastrophic for databases
  - 1/n objects remain in same location, rest are invalidated
  - Gets worse as n grows
- Can cause a ‘cascading failure’ as database gets overwhelmed moving objects to new nodes



Consistent Hashing

- Essential mechanism for ease of scalability
- Avoids problems of rebalancing with ‘key mod N’ style hashing
  - Causing costly rebalances as N changes
- Hash keys conceptually organized as a ring
  - when a node is added, it takes its share of objects from all the other nodes
  - when a node fails, its objects are shared between the remaining machines.
  - Only 1/n keys are invalidated

David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97). ACM, New York, NY, USA, 654-663.

Let’s look at an example

Describes caches, but databases work just the same

Consistent Hashing

Now consider four objects: `object1~object4`. We use a hash function to get their key values and map them into the circle, as illustrated in figure 2.

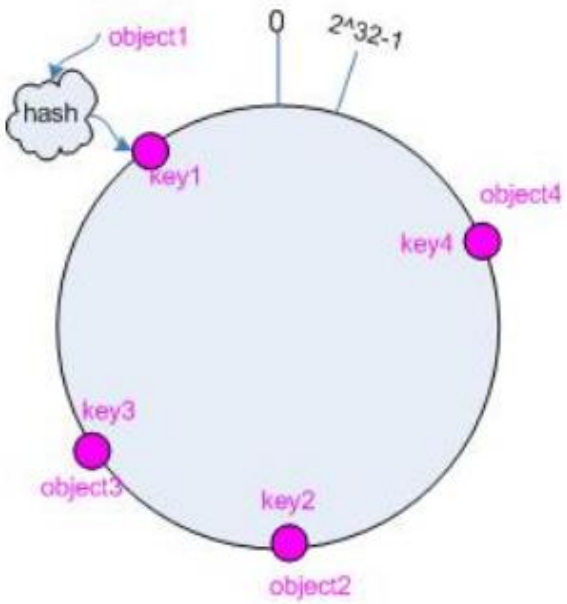


Figure 2

https://www.codeproject.com/Articles/56138/Consistent-hashing

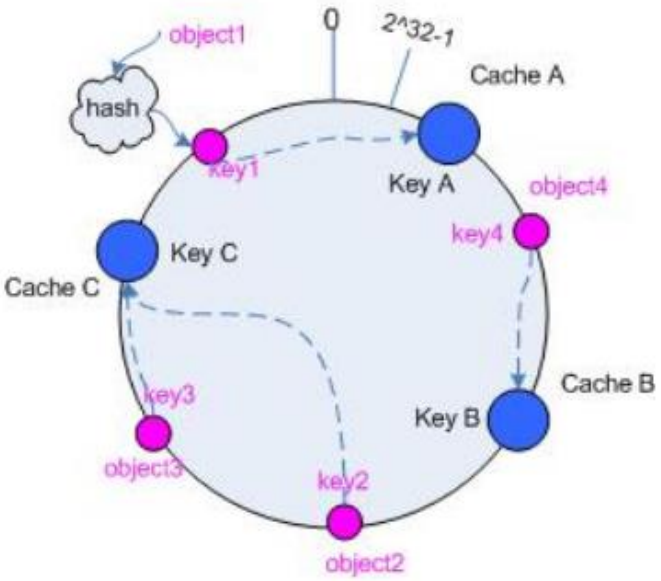
Map the cache into hash space

The basic idea of consistent hashing is to map the cache and objects into the same hash space using the same hash function.

Now consider we have three caches, A, B and C, and then the mapping result will look like in figure 3.

Hide Copy Code

```
hash(cache A) = key A;
....
hash(cache C) = key C;
```



Add or remove cache

Now consider the two scenarios, a cache is down and removed; and a new cache is added.

If cache B is removed, then only the objects that cached in B will be rehashed and moved to C; in the example, see `object4` illustrated in figure 4.

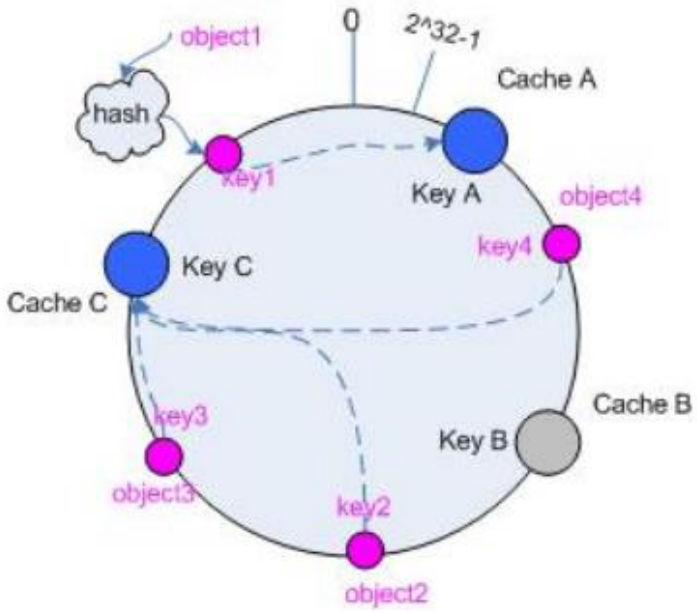


Figure 4

If a new cache D is added, and D is hashed between `object2` and `object3` in the ring, then only the objects that are between D and B will be rehashed; in the example, see `object2`, illustrated in figure 5.

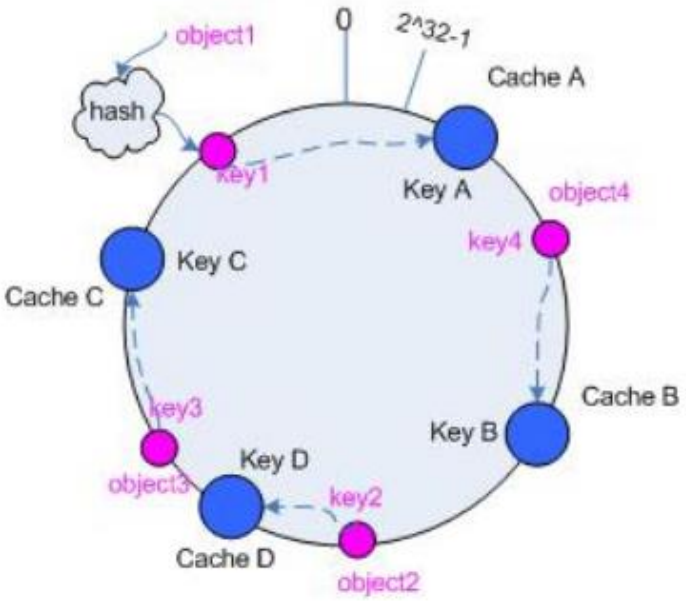


Figure 5

Virtual nodes

It is possible to have a very non-uniform distribution of objects between caches if you don't deploy enough caches. The solution is to introduce the idea of "virtual nodes".

Virtual nodes are replicas of cache points in the circle, each real cache corresponds to several virtual nodes in the circle; whenever we add a cache, actually, we create a number of virtual nodes in the circle for it; and when a cache is removed, we remove all its virtual nodes from the circle.

Consider the above example. There are two caches A and C in the system, and now we introduce virtual nodes, and the replica is 2, then three will be 4 virtual nodes. Cache A1 and cache A2 represent cache A; cache C1 and cache C2 represent cache C, illustrated as in figure 6.

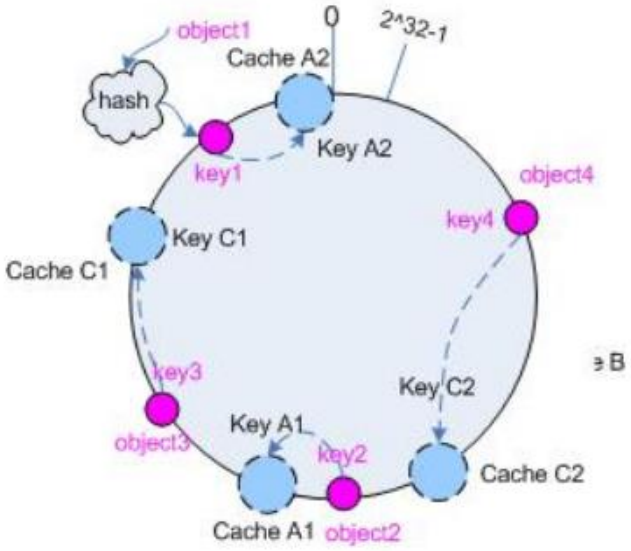
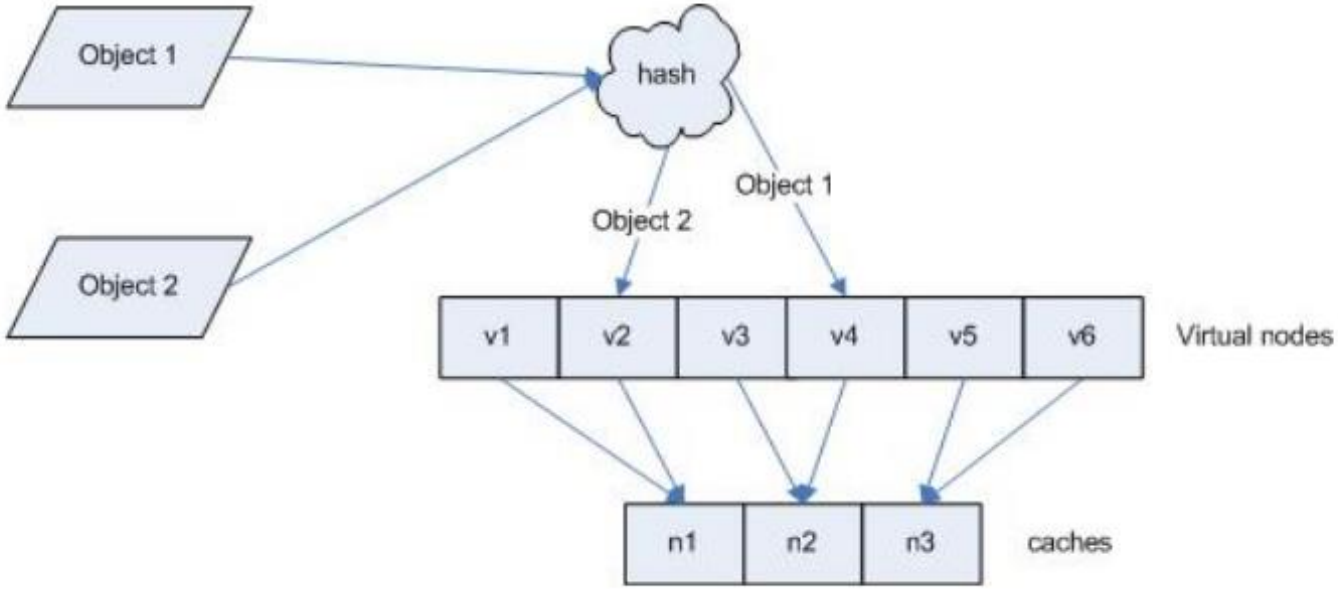


Figure 6

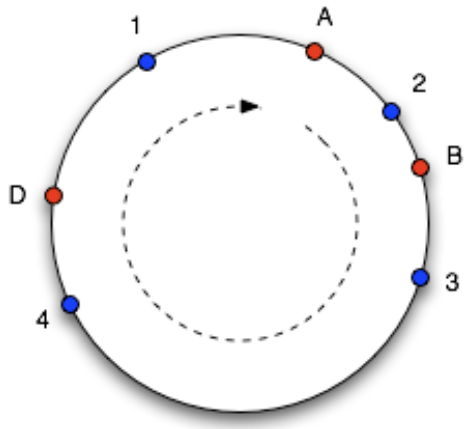
Mapping

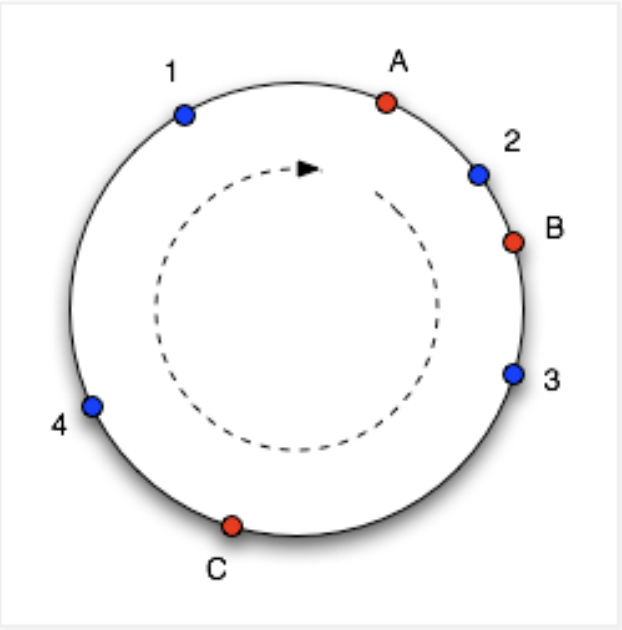
```
objec1->cache A2; objec2->cache A1; objec3->cache C1; objec4->cache C2
```

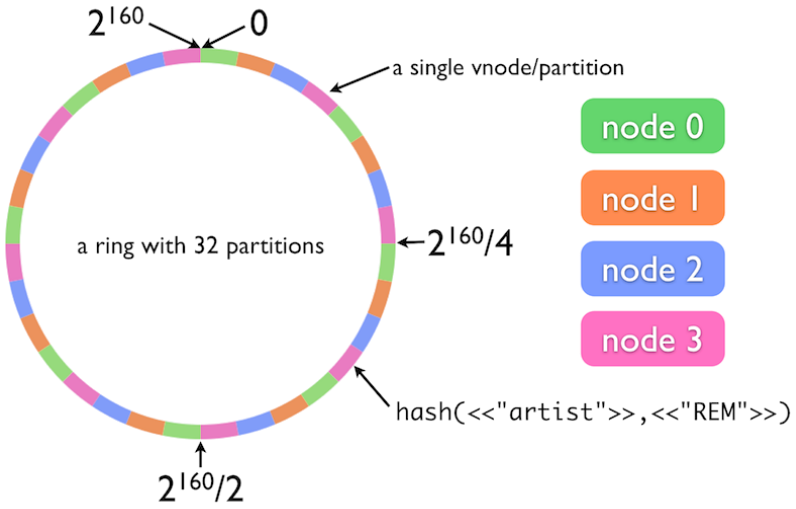


Consistent Hashing

- *Consistently* maps keys to same node
- Hash both nodes and keys into same hash space
- Example: range -2^{31} to $2^{31}-1$
  - Treat as a circle so values wrap around
- We hash nodes (e.g A,B,C) and keys (e.g. 1,2,3,4) into same range (top figure)
- Keys allocated to nodes by moving clockwise around the ring until a node's range is discovered





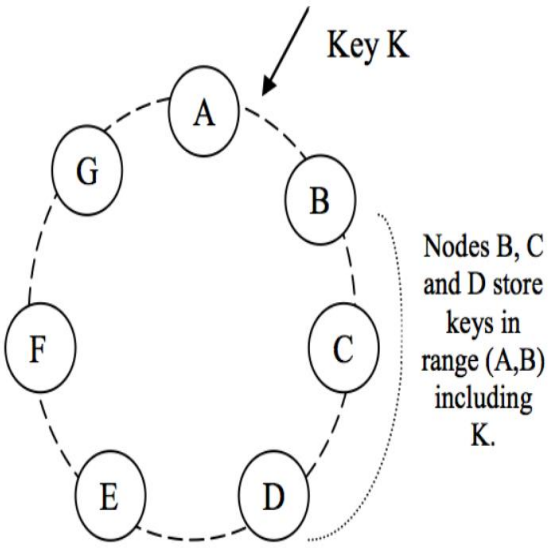


Riak Replication

- Multi-master architecture
  - Each node in a Riak cluster allocated 1/N keys in a 160-bit integer key space
- Each node split into vnodes
  - Responsible for single ring partition
  - Multiple partitions per node
  - Bucket information shared by a gossip protocol
  - Data replicated to separate vnodes

http://docs.basho.com/riak/1.1.4/references/appendices/concepts/

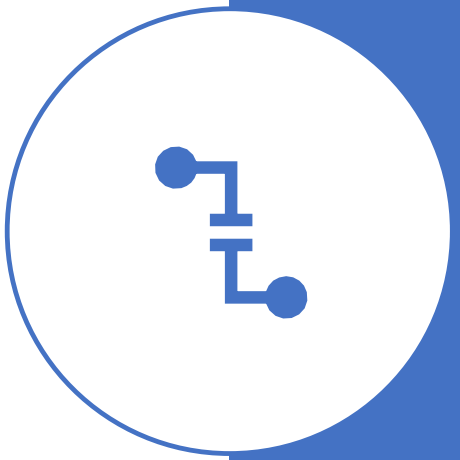
Finding Keys



- Each node maintains a preference list
- Essentially maps keys to vnodes where they are stored
- Any node can handle a client request
  - Looks up key in preference list coordinates the request across replicas
  - Acts as request coordinator

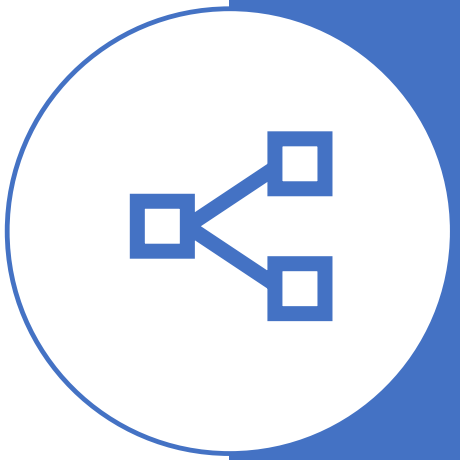
Conflict Resolution

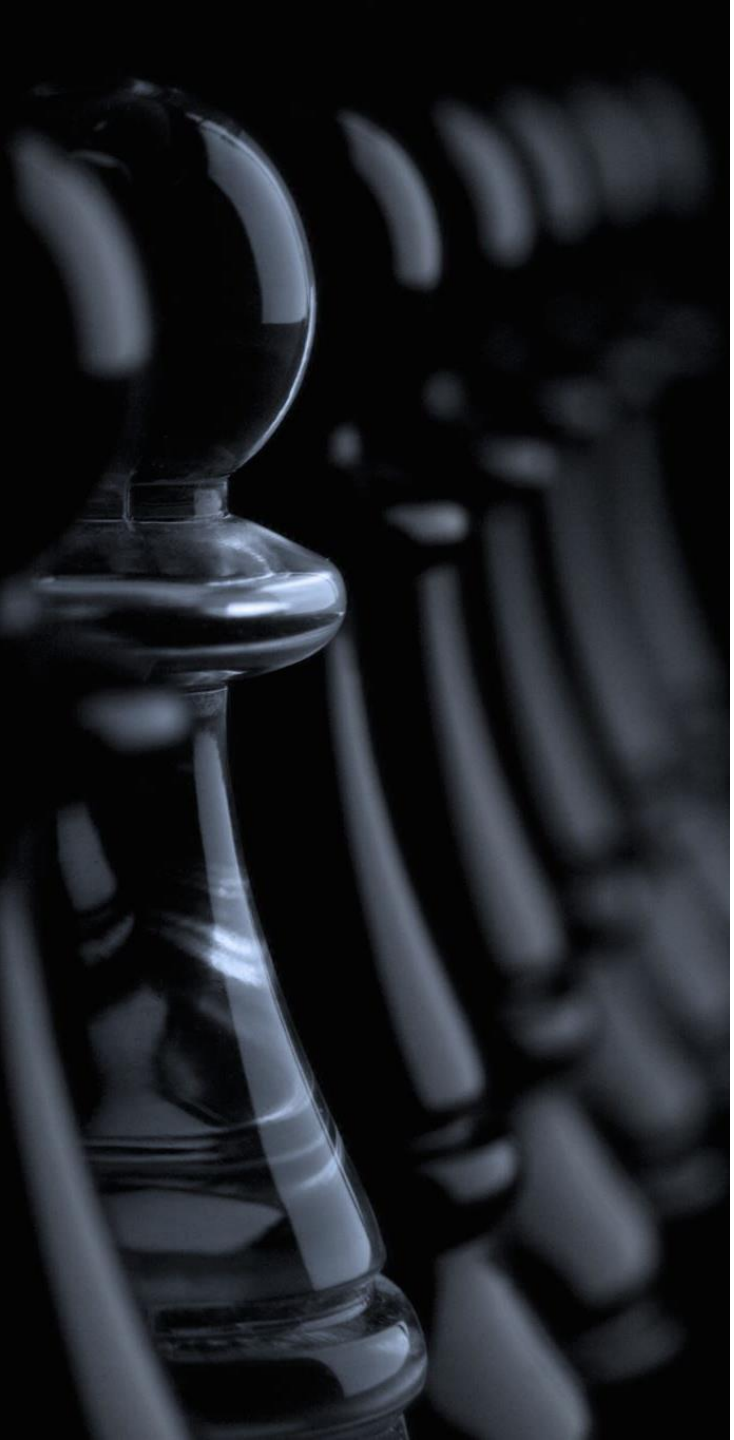
- Riak aims for high availability.
- Any node (leaderless) can process client requests
  - without every replica node participating in every request.
- Eventually consistency means conflicts between different replica object values inevitable
- Riak detects conflicts if you use causal context, i.e. vector clocks or dotted version vectors, when updating objects.



Conflict Resolution

- allow_mult = true
- Riak retains writes when concurrent updates to a key occur
  - siblings
- Application applies own use-case-specific conflict resolution logic for siblings





Conflict Resolution

```
import com.basho.riak.client.api.cap.ConflictResolver;


public class ThingResolver implements
ConflictResolver<Thing> {
    @Override
    public Thing resolve(List<Thing> siblings) {
        // Insert your sibling resolution logic here
    }
}
// register conflict resolver singleton
ConflictResolverFactory factory =
                ConflictResolverFactory.getInstance();
factory.registerConflictResolver(Thing.class, new
                ThingResolver());
```

Conflict Resolution

```
// First, we fetch the object with ‘Key’
FetchValue fetch = new FetchValue.Builder(Key).build();
FetchValue.Response res = client.execute(fetch);

// this object has causal context attached
RiakObject obj = res.getValue(RiakObject.class);


// Then we modify the object's value
obj.setValue(BinaryValue.create(“New Value"));

// Then we store the object, which has the causal context still  attached
StoreValue store = new StoreValue.Builder(obj)
      .withLocation(Key);
client.execute(store);
// Riak can now detect conflicts …
```

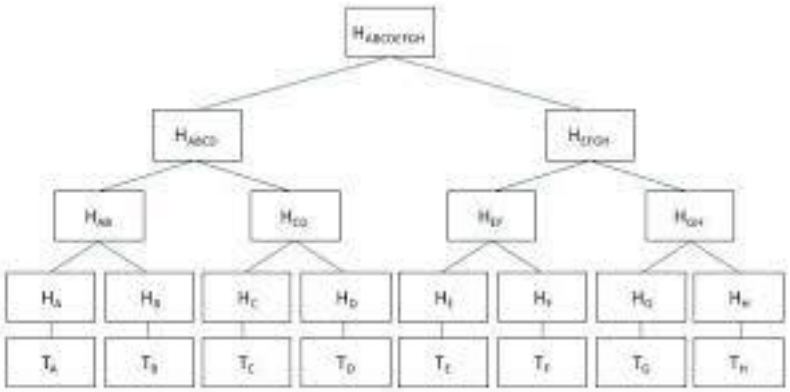
Read Repair

- Read repair:
  - when a successful read occurs—i.e. the R value number of nodes have responded — but not all replicas of the object agree on the value.
- A node may not have the object
  - Returns not_found
- A node respond with a causal context that is an ancestor of the casual context of other replicas
  - Object value out of date
- Riak tells the errant nodes to update the object's value based on the value of other nodes.

Active Anti-Entropy (AAE) Repair

- Aka Replica Synchronization
- AAE is a continuous background process that enables continuous repair of divergent replicas
- Builds a disk-based Merkle tree representation of Riak objects.
- Supports Merkle tree exchange to find divergent replica objects

Merkle Trees



- Aka a Hash Tree
- leaf nodes are labelled with the hash of associated objects
- every non-leaf node is labelled with hash of the labels of its child nodes.

AAE Repair

- Nodes periodically exchange Merkle tree hash values
- If root hash values are same, then all objects are identical
  - ☺
- If root hash not the same, at least one object has divergent replica values
- AAE recursively compares the tree, level by level, until it pinpoints objects with difference values between replica nodes.

Other Riak Features

- Hinted handoffs
- Secondary Indexes
- MapReduce
- Strong consistency
- Convergent Replicated Data Types (CRDTs), e.g.
  - Counters
  - Flags
  - Maps
  - Registers
  - Sets

Lab Exercise: Consistent Hashing

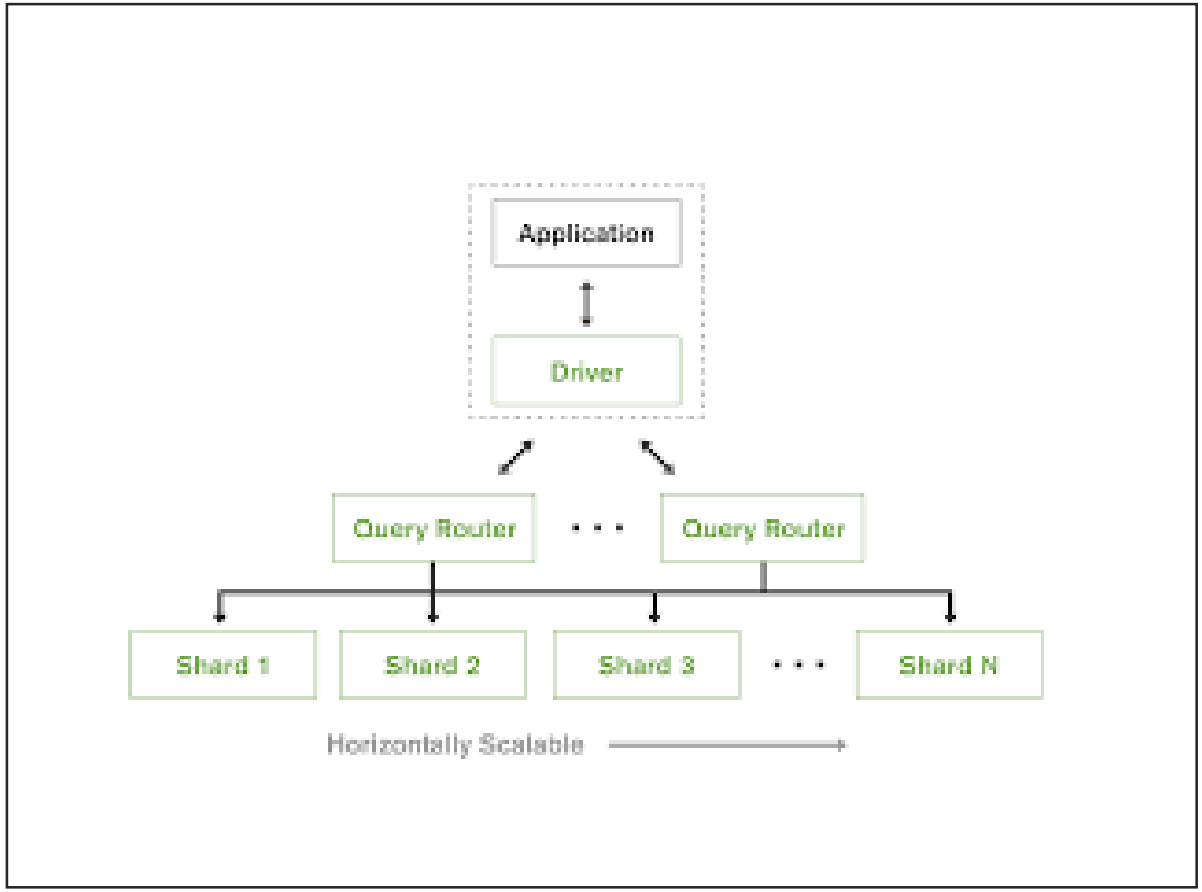
MongoDB

Document Store

MongoDB (v4.2)

- JSON-like document store
- schemaless
- Minimizes impedance mismatch between data store and programming languages
- Ad hoc queries, indexing, and real time aggregation
- Leader-based distributed, horizontal scaling and replication

MongoDB Architecture



Create Documents

- Documents organized with collections
  - Analogous to a relational table
- Each document as a unique _id field that is primary key.
- If an inserted document omits the _id field, the MongoDB driver automatically generates an _id field and returns it

```
db.inventory.insertOne(
   { item: "canvas", qty: 100, tags: ["cotton"], size: { h: 28, w: 35.5, uom: "cm" } }
)

db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w: 21, uom: "cm" } },
   { item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom: "cm" } },
   { item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w: 22.85, uom: "cm" } }
])
```

Read Documents

- Selects documents in a collection or view and returns a cursor to the selected documents
- Can specify fields to search on and conditionals

```
db.inventory.find( { item: "canvas" } )


var myCursor = db.bios.find( );

var myDocument = myCursor.hasNext() ? myCursor.next() : null;

if (myDocument) {
    var myName = myDocument.name;
    print (tojson(myName));
}
```

```
$currentDate: { lastModified: true }
```

Update Documents

- To update a document, MongoDB provides update operators, eg
  - $set, to modify field values.

```
db.inventory.updateOne(
   { item: "paper" },
   {
     $set: { "size.uom": "cm", status: "P" },
     $currentDate: { lastModified: true }
   }
)
```

```
db.inventory.updateMany(
   { "qty": { $lt: 50 } },
   {
     $set: { "size.uom": "in", status: "P" },
     $currentDate: { lastModified: true }
   }
)
```

Delete Documents

- // delete all collection docs
- db.inventory.deleteMany({})

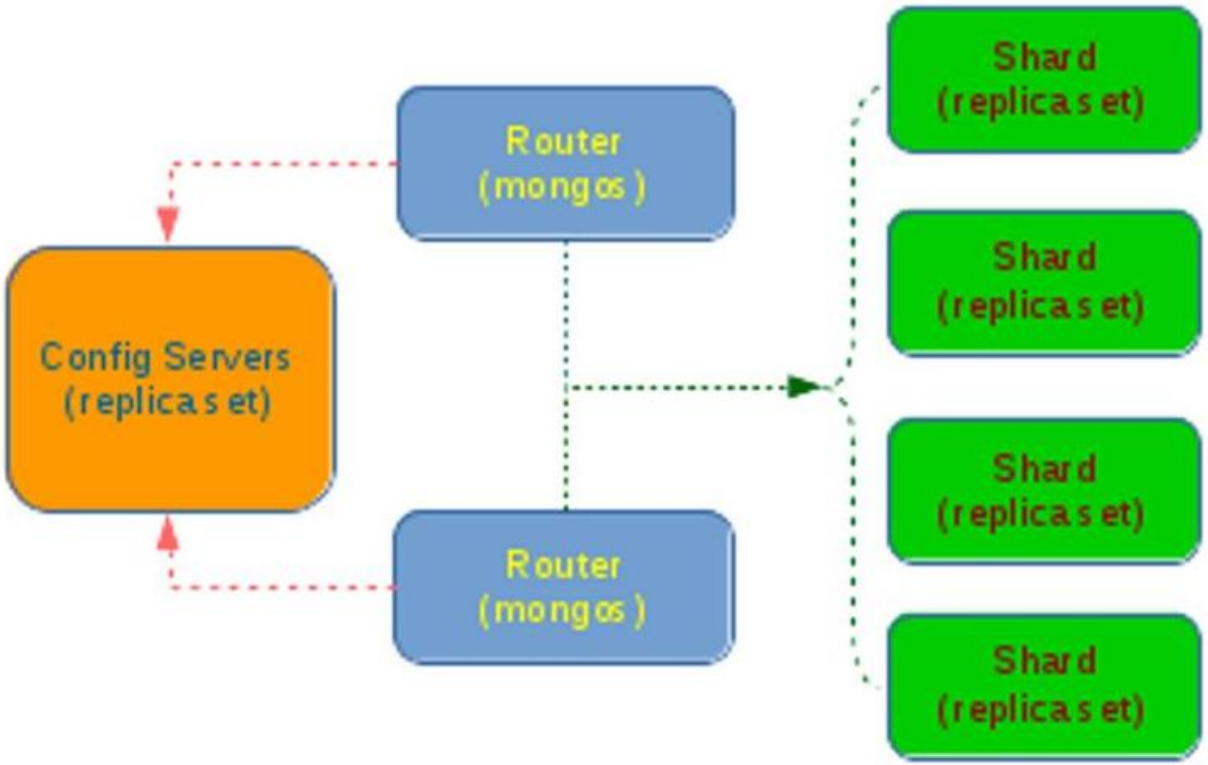
- db.inventory.deleteMany({ status : "A" })

- db.inventory.deleteOne({ status: "D" })

MongoDB Partitioning

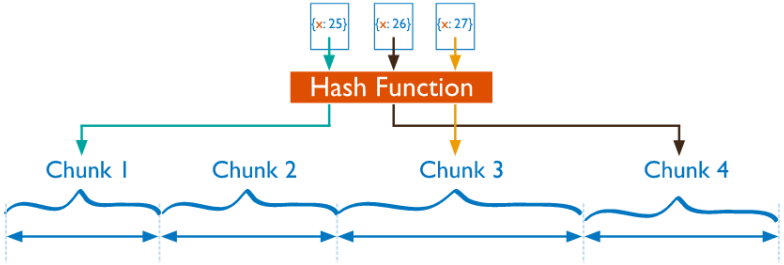
- Aka sharding
- shard: subset of the data set.
- mongos: a query router that provides an interface between client applications and the sharded cluster.
- config servers: Config servers store metadata and configuration settings for the cluster

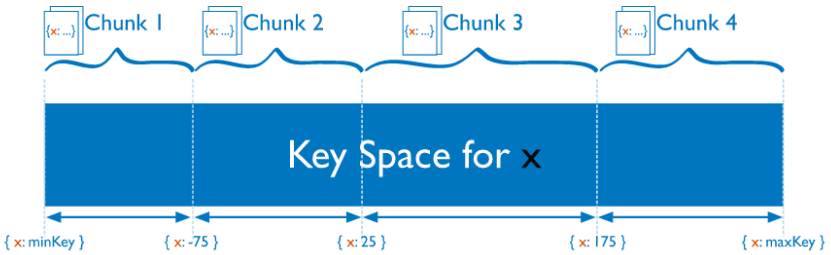
Sharding in MongoDB



Shard Key

- distributes the collection’s documents across shards.
- consists of a field or fields that must exist in every document in the collection.
- The choice of shard key cannot be changed after sharding.
- Shard keys cane be
  - Hashed
  - ranges





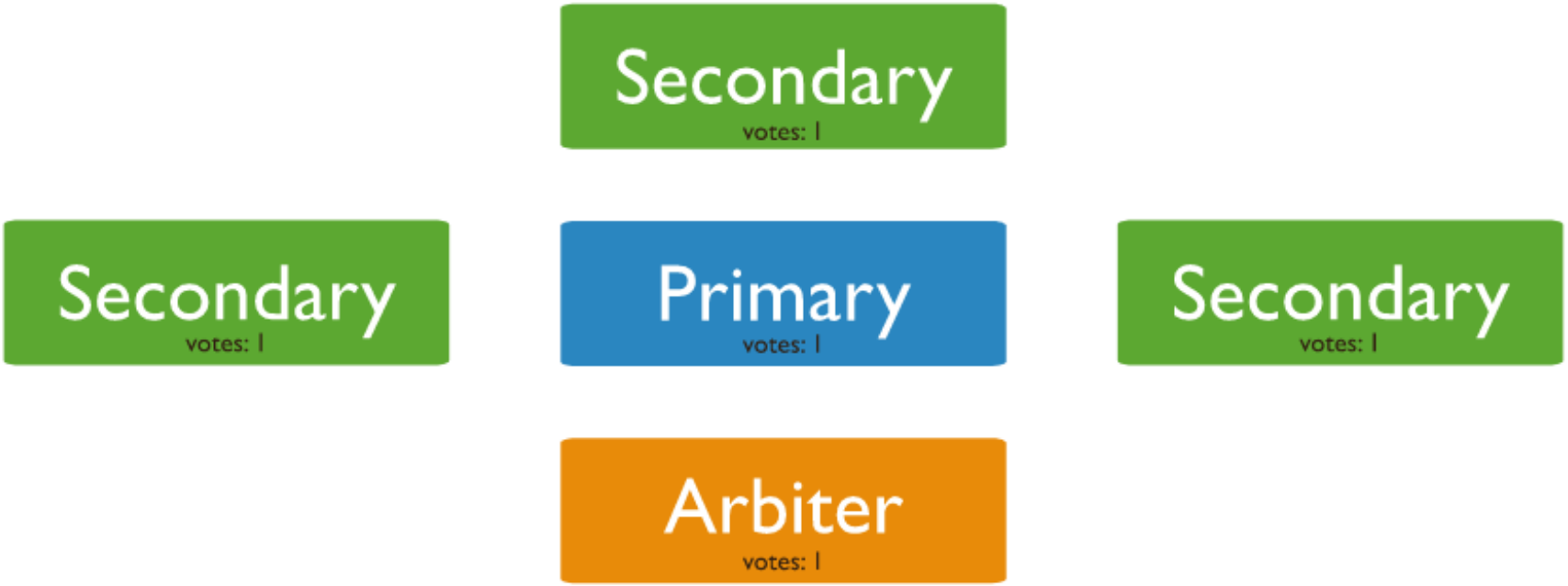
Sharding Options

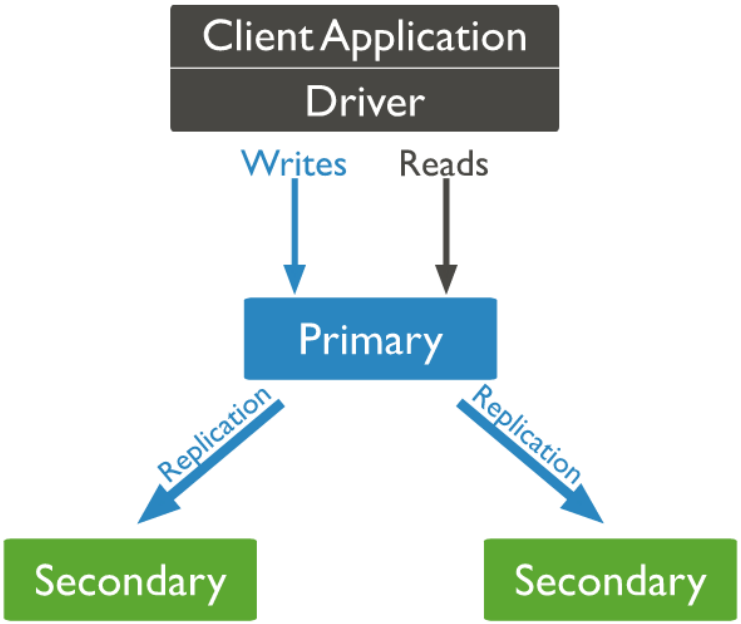
Replication

- Aka *Replica Sets*
- group of mongod instances that store the same data set
- contains several data nodes and optionally one *arbiter* node.
- Data nodes have one primary (leader) and multiple secondaries (followers)
- Arbiter nodes support primary failover/elections

Arbiter

an arbiter allows the set to have an odd number of votes for elections

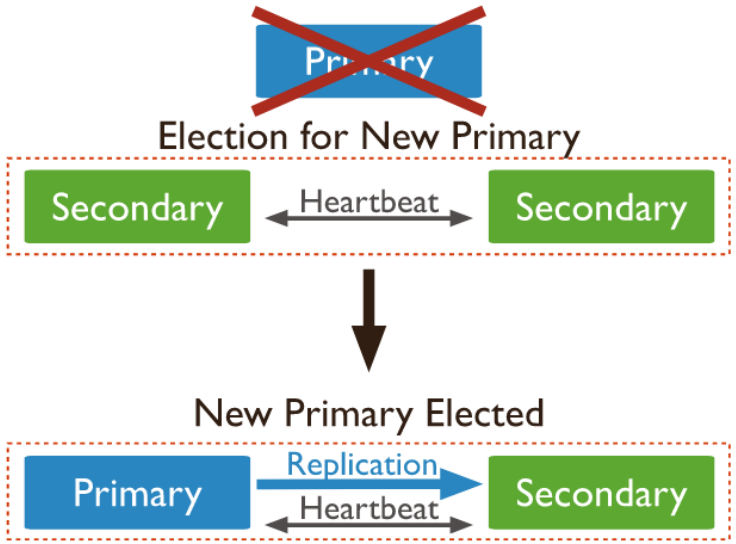




Replica Sets

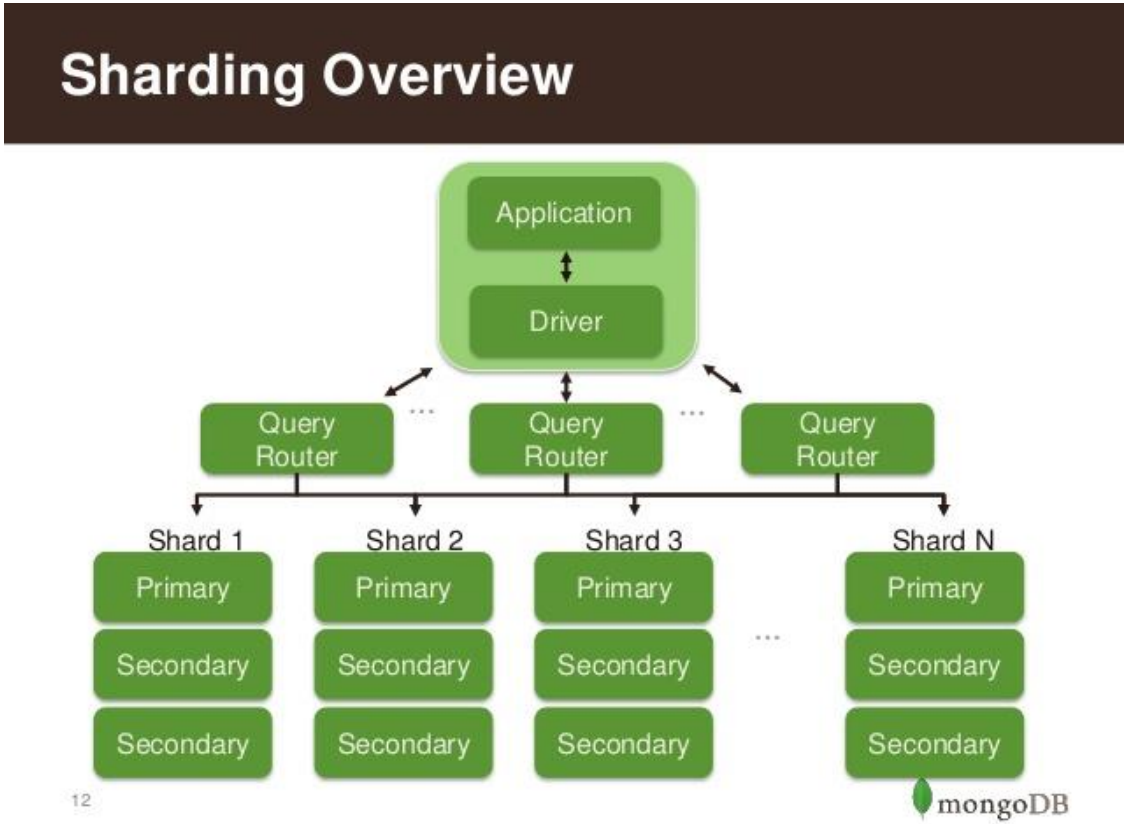
- Writes go to primary
- Secondaries asynchronously replicate the primary transaction log (oplog) to support eventual consistency
- By default reads go to the primary
  - Read Preference settings can send reads to secondaries

Primary Elections



- Primary deemed failed after 10 second (default) timeout
- Secondaries can be assigned priorities
- Highest priority will become leader
- Network partitions can cause a primary to step down if it can't see a majority of nodes
- May need to rollback writes from primary if they were not replicated to secondaries before partition

Sharding and Replication



MongoDB Other features

- Indexes on any document field
- Multi document transactions
- Change streams
- MapReduce operations on collections

Finally – a quick reflection

Riak is eventually consistent by default

Favors availability over consistenty

MongoDB is strongly consistent by default

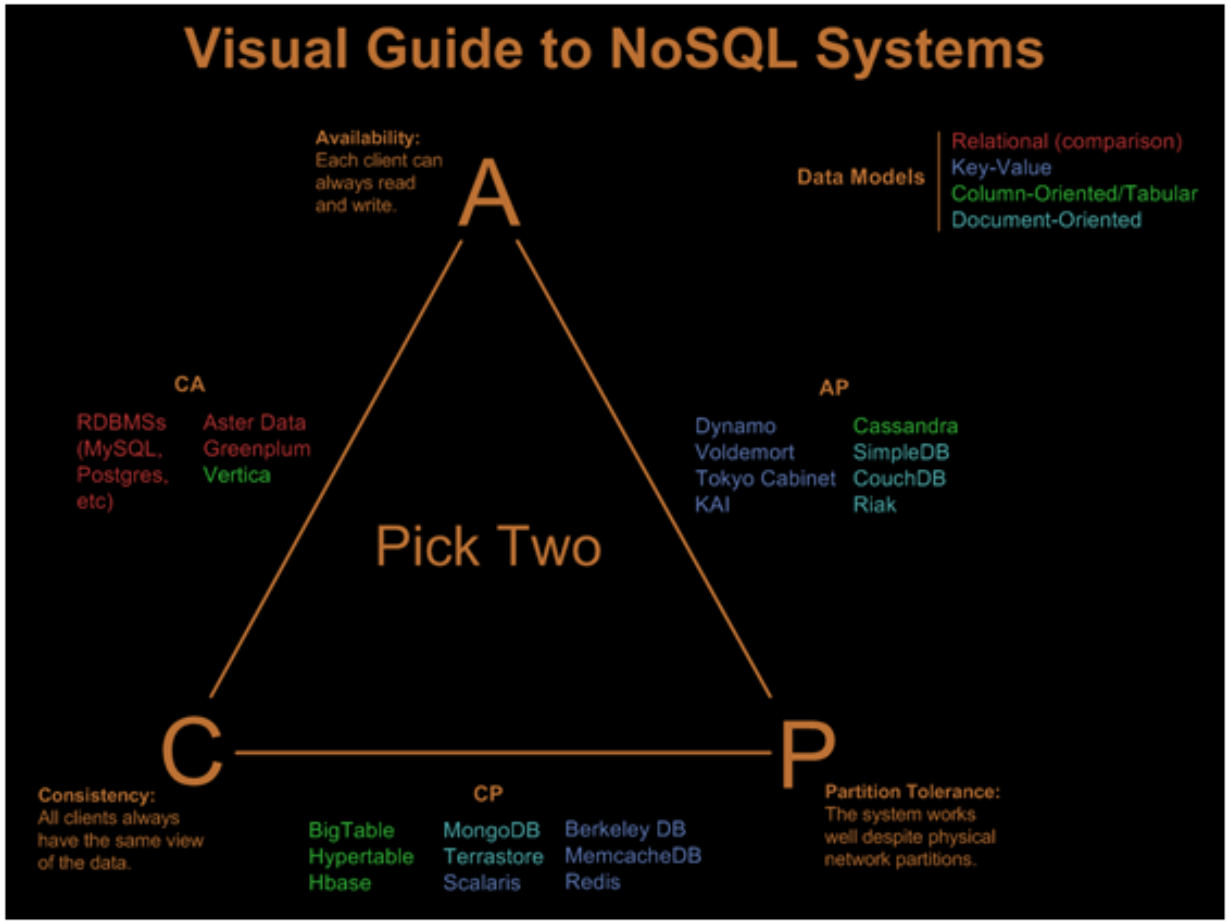
Reads/writes go to master for a hard

CAP Theorem

- In the presence of a partition, two options:
  - consistency
  - Availability
- With no partition, both availability and consistency can be satisfied.
- ***the choice is between consistency and availability only when a network partition or failure happens***

Eric Brewer, "CAP twelve years later: How the 'rules' have changed", Computer, Volume 45, Issue 2 (2012), pg. 23–29. doi:10.1109/MC.2012.37

The CAP Theorem



Summary

- Riak is a leaderless key-value database that uses consistent hashing to replicate and distribute data
- MongoDB is a leader-based system supporting sharding and replication
- CAP Theorem described choice between availability and consistency when a partition occurs