NOSQL

Lê Hồng Hải
UET-VNUH

Overview

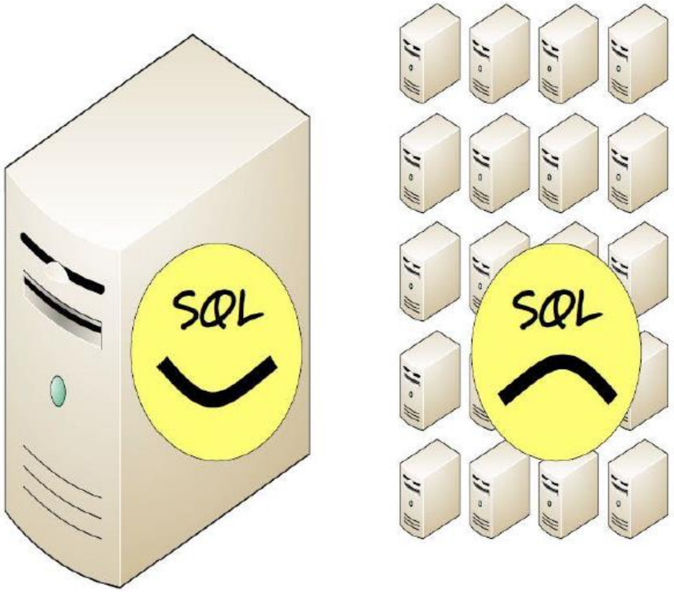
Relational databases

- Very good background
- Standard Query Language (SQL)
- ACID
- Strong consistency, concurrency, recovery
- Lots of tools to use i.e: Reporting services, entity frameworks, ...

SQL Databases

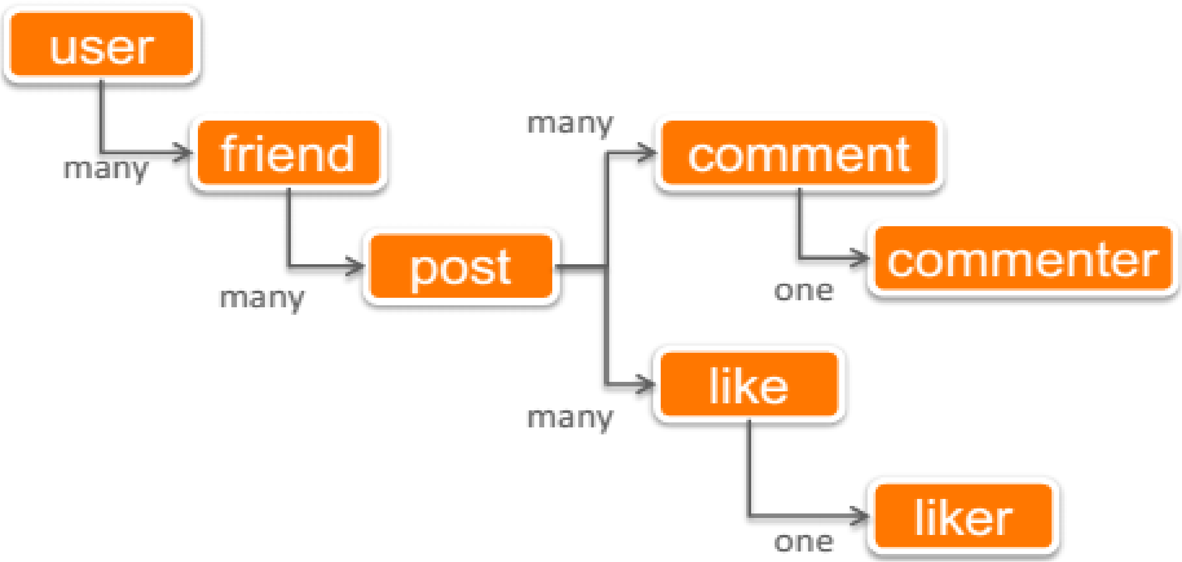
Relational databases

- Relational databases were not built for **distributed applications.**
- Joins are expensive
- Hard to scale horizontally
- Expensive (product cost, hardware, Maintenance)



Relational databases

- In the relational database model, it is needed to join a large number of data tables



Dealing with Big Data and Scalability

- Issues with scaling up when the dataset is just too big
- RDBMS were not designed to be distributed
- Traditional DBMSs are best designed to run well on a “single” machine
  - Larger volumes of data/operations requires to upgrade the server with faster CPUs or more memory known as ‘Scaling up’ or ‘Vertical scaling’

No SQL

- NoSQL stands for:
  - No Relational
  - No RDBMS
  - Not Only SQL
- NoSQL is an umbrella term for all databases and data stores that don't follow the RDBMS principles

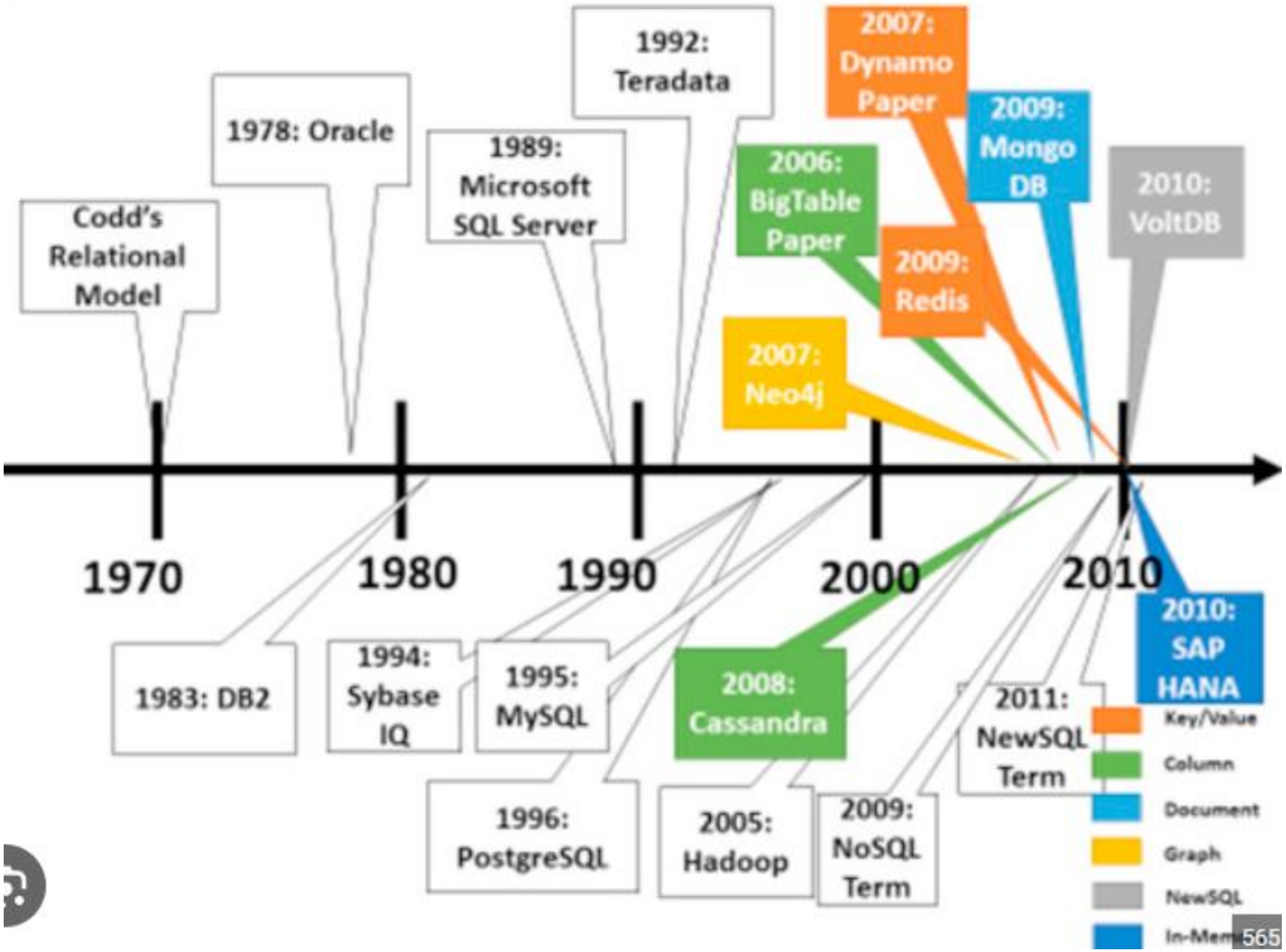
NoSQL Definition

From www.nosql-database.org:

Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable. The original intention has been modern web-scale databases. The movement began early 2009 and is growing rapidly.

Often more characteristics apply as: schema-free, easy replication support, simple API, eventually consistent / BASE (not ACID), a huge data amount, and more.

NoSQL History



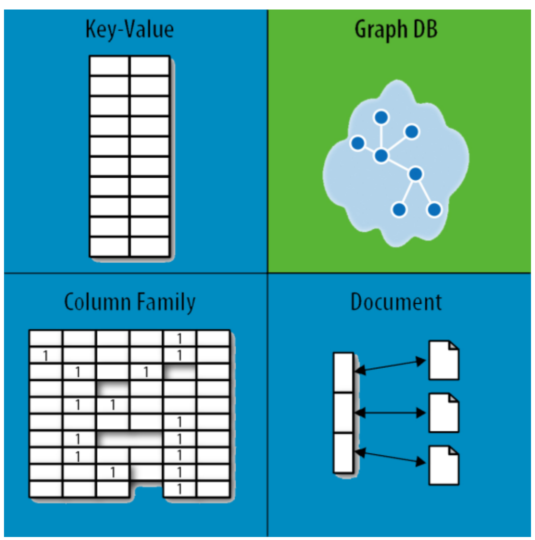
Characteristics of NoSQL databases

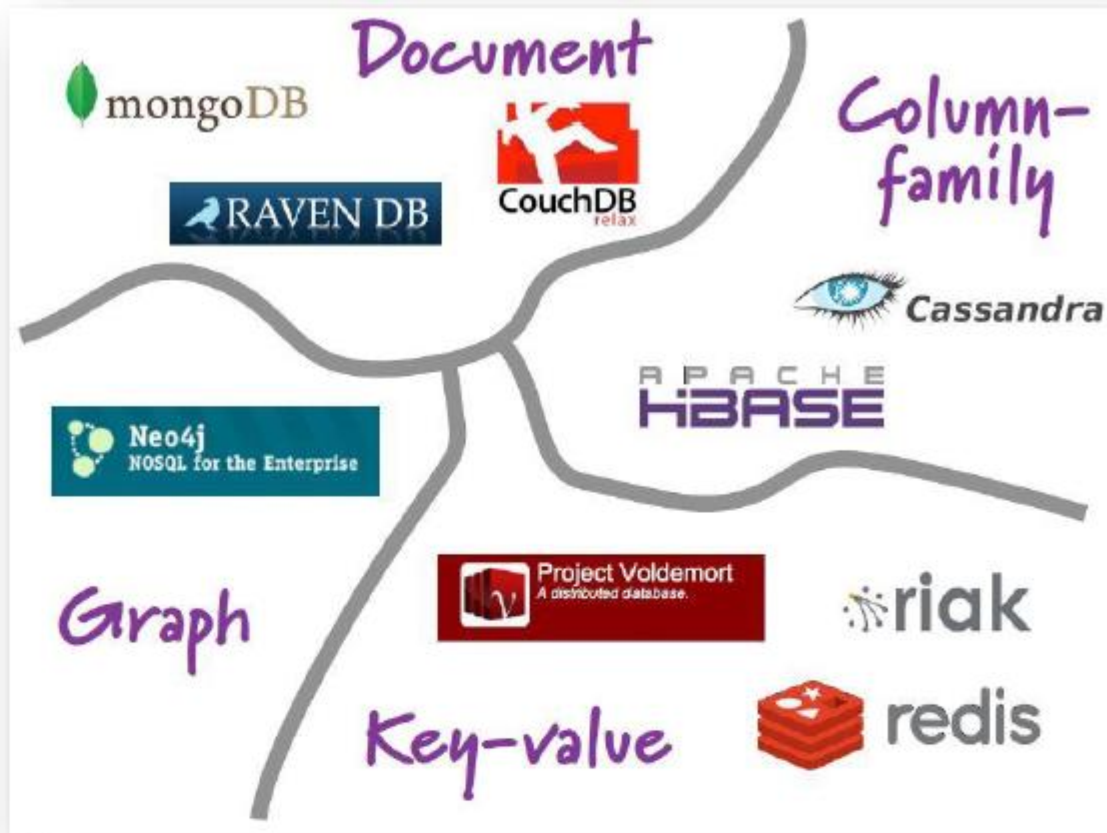
- Easy and frequent changes to DB
  - Fast development
  - Large data volumes (eg. Google)
  - Schema less
- NoSQL solutions are designed to run on clusters or multi-node database solutions
- When not use:
  - Financial Data
  - Data requiring strict ACID compliance
  - Business Critical Data

NoSQL is getting more & more popular

Ashwani Kumar

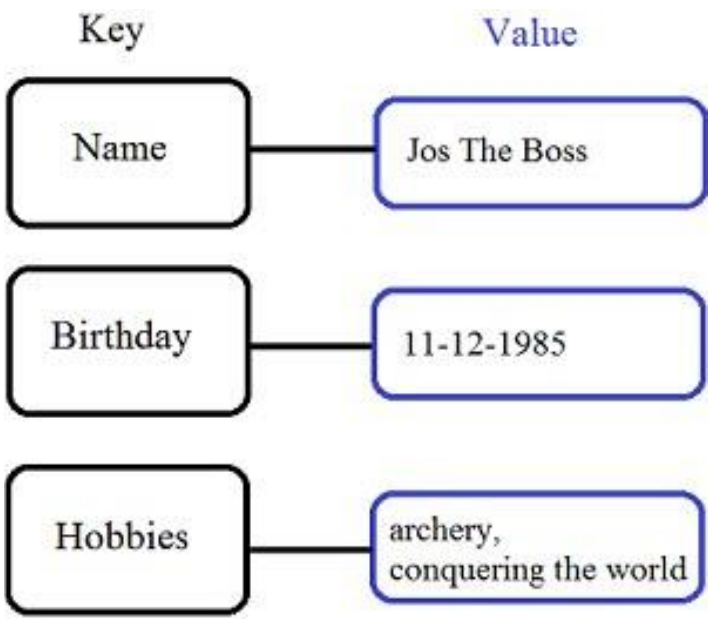
- NoSQL databases are classified in four major data models:
  - Key-value
  - Document
  - Column family
  - Graph





Key-value data

- Simplest NOSQL databases
- The main idea is the use of a hash table
- Access data (values) by strings called keys
- Data has no required format data may have any format



Key/Value stores

- Store data in a schema-less way
- Store data as maps: HashMaps or associative arrays
- Provide a very efficient average running time algorithm for accessing data

Use cases of key-value databases

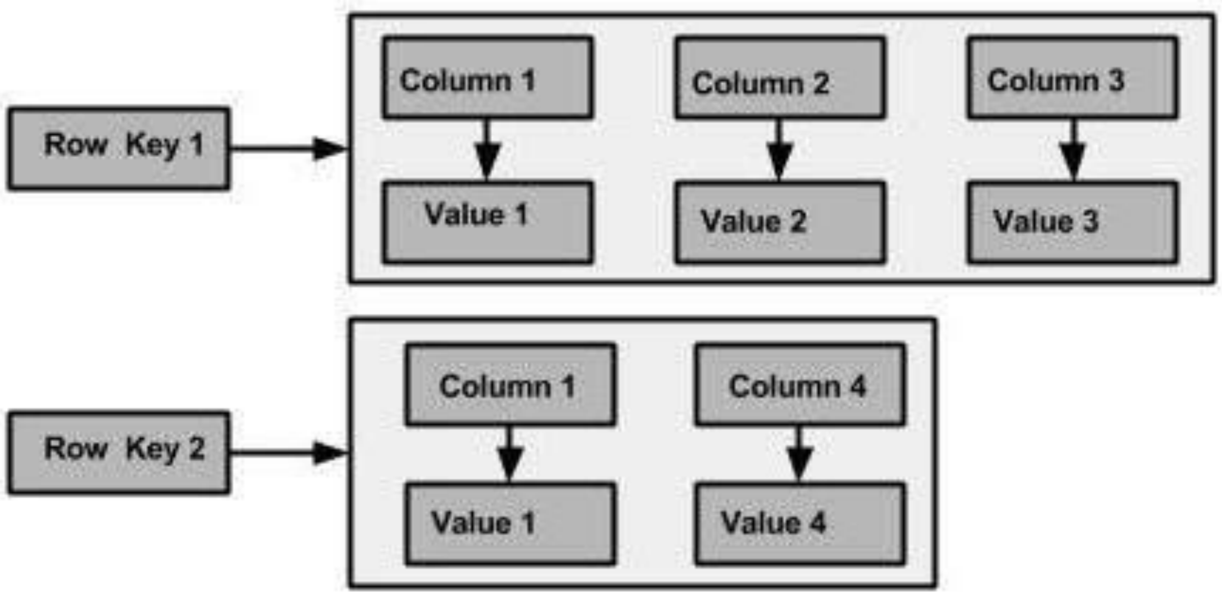
- Session management
  - A session-oriented application, such as a web application, starts a session when a user logs in to an application and is active until the user logs out or the session times out.
- Shopping cart
  - An e-commerce website may receive billions of orders per second during the holiday shopping season
- Caching
  - You can use a key-value database for storing data temporarily for faster retrieval

NoSQL Data Models

1. Key-Value
2. Column Wide
3. Graph
4. Document

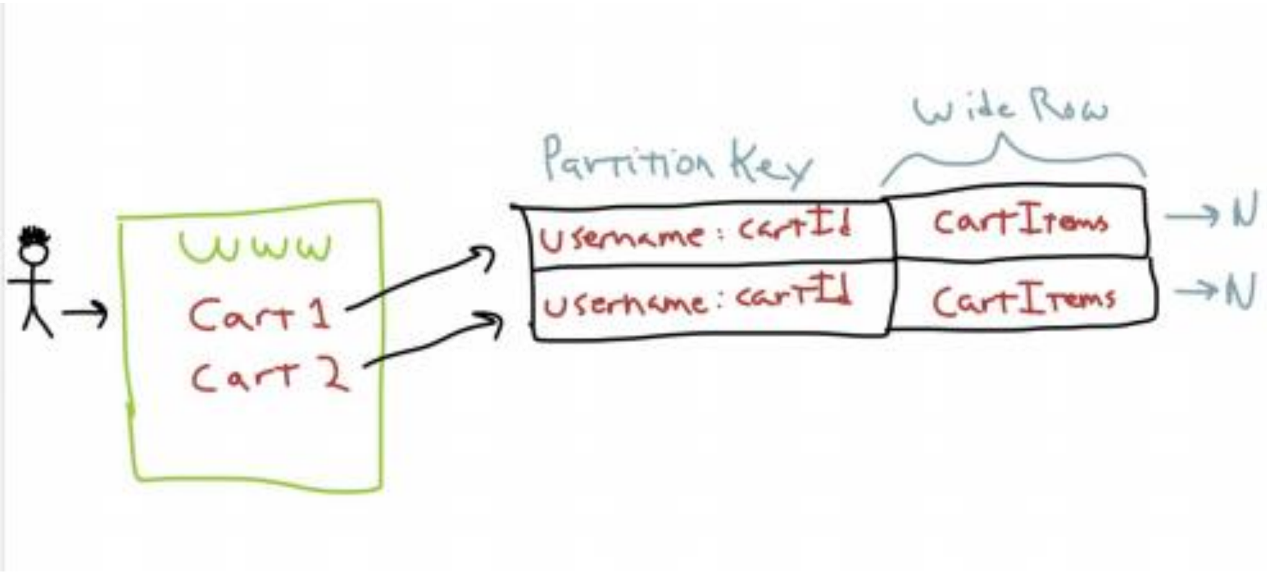
Column wide

- Data are stored in a column-oriented way
  - Data isn’t stored as a single table but is stored by column families
  - Unit of data is a set of key/value pairs
    - **Identified by “row-key”**
    - Ordered and sorted based on row-key



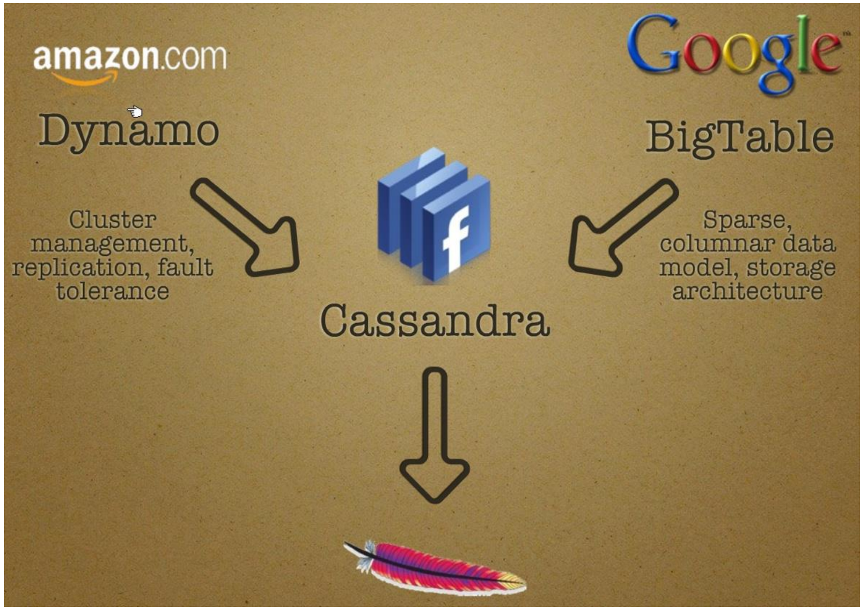
Column Wide

- Can write data with a large number of (dynamic) columns to a data table
- The cartItems part along with the username key and cardId will be written serially to the data stream
- Therefore, it helps to quickly retrieve data during customer purchases



Cassandra Column wide

- Cassandra stands out with the advantage of being able to write and read at any computer node in the cluster, especially writing speed



Data on Cluster

- Determine the location of the data access node based on the partition key

```
SELECT name, description, added_date
FROM videos
WHERE videoid = 06049cbb-dfed-421f-b889-5f649a0de1ed;
```

Cassandra Column wide

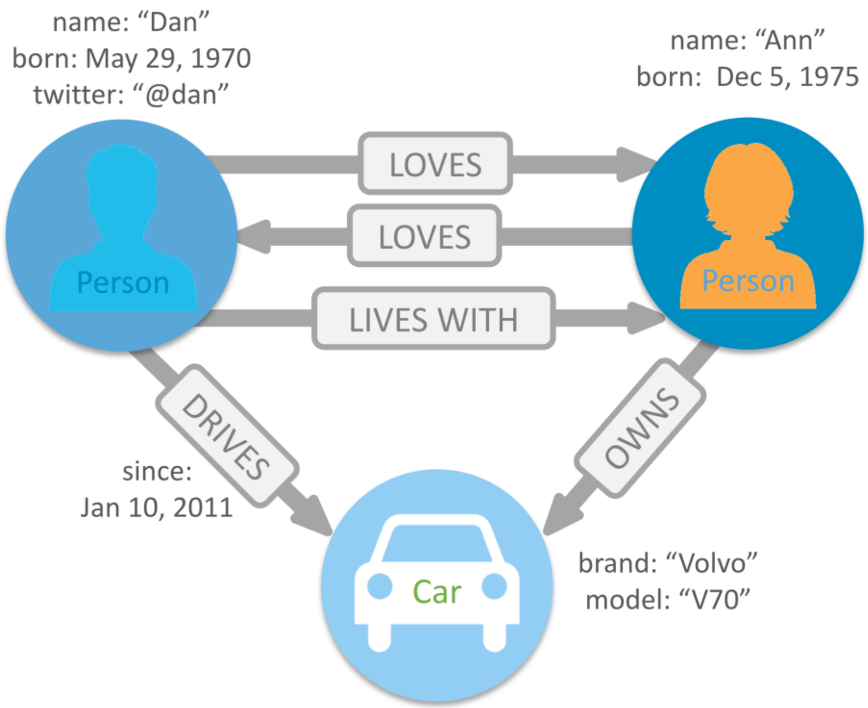
Cassandra Column wide

- **Some statistics about Facebook Search** (using Cassandra)
- MySQL > 50 GB Data
  - Writes Average : ~300 ms
  - Reads Average : ~350 ms
- Rewritten with Cassandra > 50 GB Data
  - Writes Average : 0.12 ms
  - Reads Average : 15 ms

NoSQL Data Models

1	Key-Value
2	Column Wide
3	Graph
4	Document

- **Nodes**: These are the instances of data that represent objects which is to be tracked
- **Edges**: As we already know edges represent relationships between nodes
- **Properties**: It represents information associated with nodes.

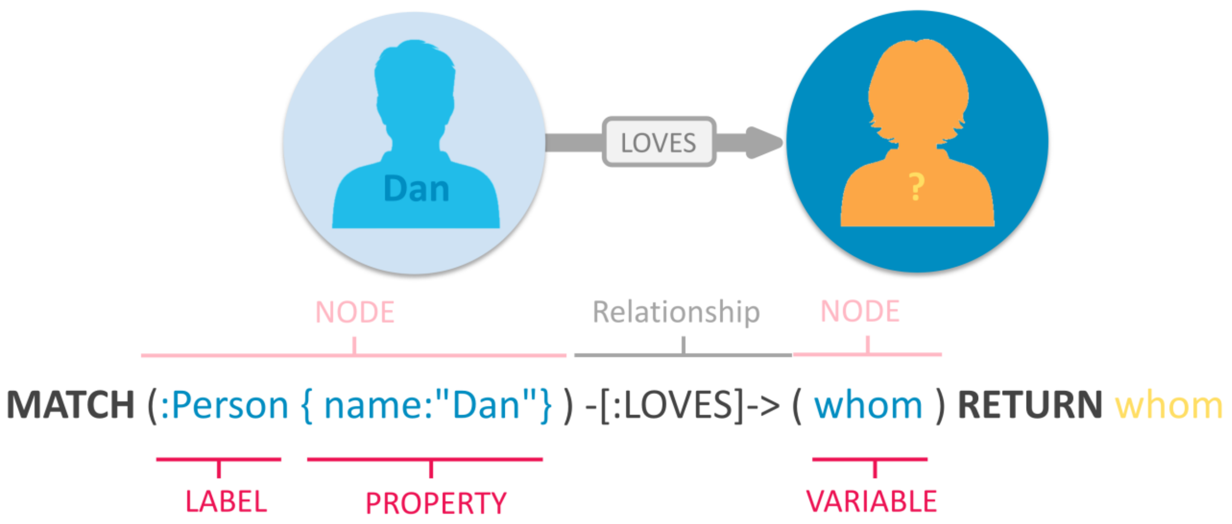


Graph Databases

- While existing relational databases can store these relationships, they navigate them with expensive JOIN operations or cross-lookups, often tied to a rigid schema
- It turns out that "relational" databases handle relationships poorly

Graph Databases

- In a graph database, there are no JOINs or lookups. Relationships are stored natively alongside the data elements (the nodes)
- Everything about the system is optimized for traversing through data quickly



Graph databases

- Graph databases address big challenges many of us tackle daily. Modern data problems often involve many-to-many relationships with heterogeneous data that set up needs to:
  - Navigate deep hierarchies
  - Find hidden connections between distant object
  - Discover inter-relationships between objects

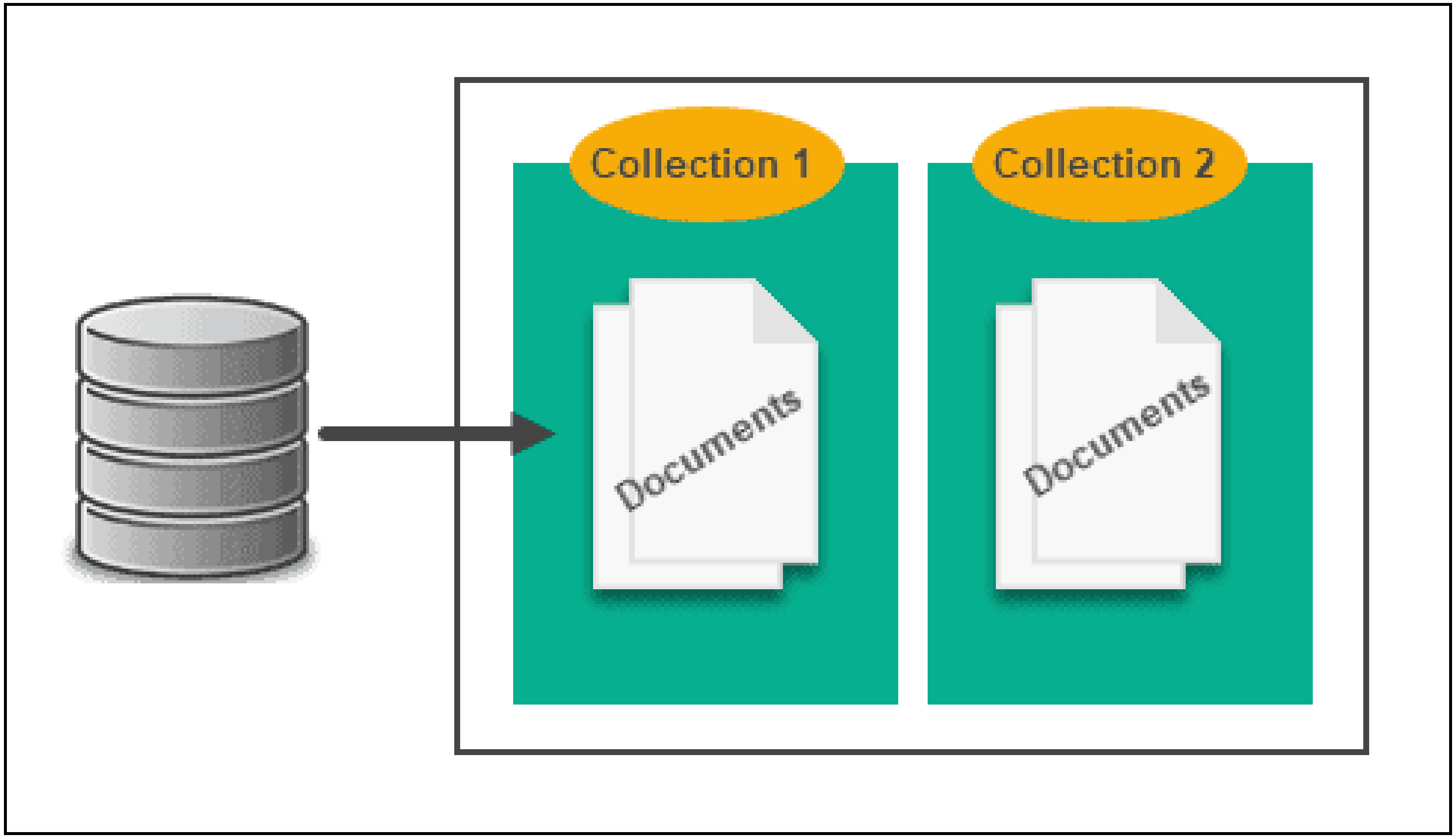
NoSQL Data Models

1	Key-Value
2	Column Wide
3	Graph
4	Document

Document Databases (Document Store)

- Documents
  - Loosely structured sets of key/value pairs in documents, e.g., XML, JSON, BSON
  - Are addressed in the database via a unique key
  - Documents are treated as a whole, avoiding splitting a document into its constituent name/value pairs
- Notable for:
  - **MongoDB** (used in FourSquare, Github, and more)
  - **CouchDB** (used in Apple, BBC, Canonical, Cern, and more)

Document Data



JSON document

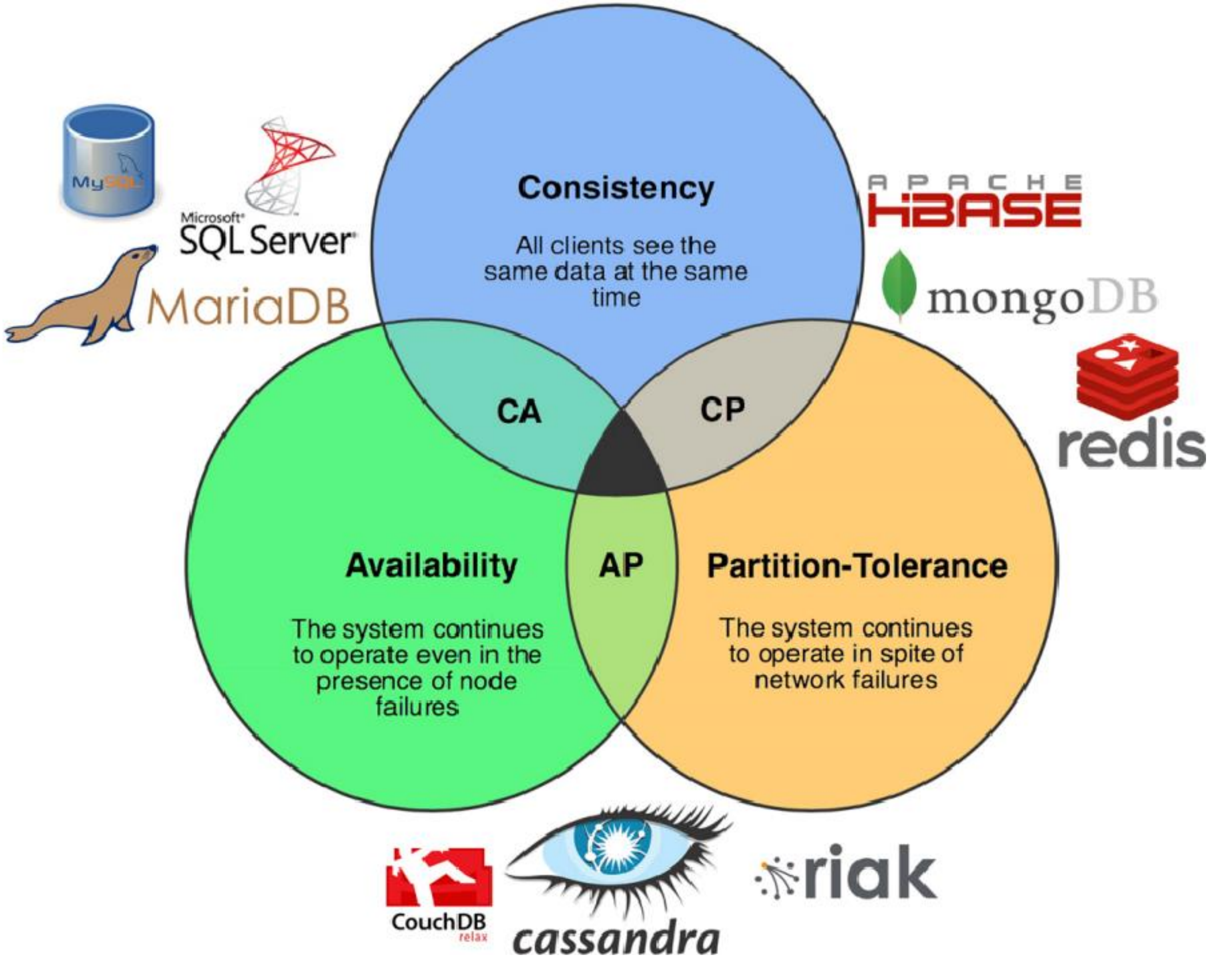
- Field names allow you to understand what kind of data is held within a document with just a glance
Documents in document databases are *self-describing*

Document Features

- **Flexible Schema:** Overall schema is very much flexible to support this statement one must know that not all documents in a collection need to have the same fields
- **Distributed:** Document data models are very much dispersed which is the reason behind horizontal scaling and distribution of data

CAP Theorem: Two out of Three

- CAP theorem – At most two properties on three can be addressed



Performance

- Every database has its advantages and disadvantages
- NoSQL is a set of concepts, ideas, technologies, and software dealing with
  - Big data
  - Sparse un/semi-structured data
  - High horizontal scalability
  - Massive parallel processing
- Different applications, goals, targets, and approaches need different NoSQL solutions

MongoDB

Terminology

Relational (SQL)	MongoDB
Database	Database
Table	Collection
Index	Index
Row	Document
Column	Field

Dynamic Typing

B-tree (range-based)

Think JSON

Primitive types + arrays, documents

Document Database

- MongoDB documents are similar to JSON objects

```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

field: value
field: value
field: value
field: value

MongoDB Document

```
{
   _id: ObjectId("5099803df3f4948bd2f98391"),
   name: { first: "Alan", last: "Turing" },
   birth: new Date('Jun 23, 1912'),
   death: new Date('Jun 07, 1954'),
   contribs: [ "Turing machine", "Turing test", "Turingery" ],
   views : NumberLong(1250000)
}
```

- _id holds an ObjectId
- name holds an embedded document that contains the fields first and last
- birth and death hold values of the Date type
- contribs holds an array of strings.
- views holds a value of the NumberLong type.

The _id Field

- In MongoDB, each document stored in a collection requires a unique _id field that acts as a primary key
- If an inserted document omits the _id field, the MongoDB driver automatically generates an ObjectId for the _id field

Data Types

- **Null**
  - The null type can be used to represent both a null value and a nonexistent field:
  - {"x" : null}
- **Boolean**
  - There is a boolean type, which can be used for the values true and false:
  - {"x" : true}
- **Number**
  - The shell defaults to using 64-bit floating-point numbers. Thus, these numbers

Data Types

String

- Any string of UTF-8 characters can be represented using the string type:
- {"x" : "foobar"}

Date

- MongoDB stores dates as 64-bit integers representing milliseconds since the Unix epoch (January 1, 1970). The time zone is not stored:
- {"x" : new Date()}

- **Array**
  - Sets or lists of values can be represented as arrays:
  - {"x" : ["a", "b", "c"]}
- **Embedded document**
  - Documents can contain entire documents embedded as values in a parent document:
  - {"x" : {"foo" : "bar"}}
- **Object ID**
  - An object ID is a 12-byte ID for documents:
  - {"x" : ObjectId()}

The advantages of using documents

- Embedded documents and arrays reduce the need for expensive joins
- Support dynamic schema supports
- MongoDB stores data records as documents (specifically BSON documents) which are gathered together in collections
- The maximum BSON document size is 16 MB

Inserting Documents

- To insert a single document, use the collection’s insertOne method:

```
db.movies.insertOne({"title" : "Stand by Me"})
```

- insertOne will add an "_id" key to the document (if you do not supply one) and store the document in MongoDB

InsertMany

- This method enables you to pass an array of documents to the database
  - db.movies.insertMany([{"title" : "Ghostbusters"},{"title" : "E.T."},{"title" : "Blade Runner"}]);

Removing Documents

- The CRUD API provides deleteOne and deleteMany for this purpose. Both of these methods take a filter document as their first parameter
  - db.movies.deleteOne({"_id" : 4})
- To delete all the documents that match a filter, use deleteMany:
  - db.movies.deleteMany({"year" : 1984})

Updating Documents

- Once a document is stored in the database, it can be changed using one of several update methods: **updateOne, updateMany, and replaceOne**
  - updateOne and updateMany each take a filter document as their first parameter and a modifier document as the second parameter
  - replaceOne also takes a filter as the first parameter, but as the second parameter replaceOne expects a document with which it will replace the document matching the filter

Update Operators

- "$set" sets the value of a field. If the field does not yet exist, it will be created
- For example: If the user wanted to store his favorite book in his profile, he could add it using "$set":
  - db.users.updateOne({"name" : "joe"}, {"$set" : {"favorite book" : "Green Eggs and Ham"}})

Update Operators

- You can remove the key altogether with "$unset“
  - db.users.updateOne({"name" : "joe"}, {"$unset" : {"favorite book" : 1}})

Introduction to find

- The find method is used to perform queries in MongoDB. Querying returns a subset of documents in a collection
  - db.users.find({"age" : 27})
- Multiple conditions can be strung together by adding more key/value pairs to the query document
  - db.users.find({"username" : "joe", "age" : 27})

Query Criteria

- Queries can go beyond the exact matching
- "$lt", "$lte", "$gt", and "$gte" are all comparison operators, corresponding to <,<=, >, and >=, respectively.
- They can be combined to look for a range of values.
  - `db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})`

OR query

- There are two ways to do an OR query in MongoDB. "$in" can be used to query for a variety of values for a single key
- "$or" is more general; it can be used to query for any of the given values across multiple keys
  - `db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )`

$not

- "$not" is a meta conditional: it can be applied on top of any other criteria

Querying Arrays

- Querying for elements of an array is designed to behave the way querying for scalars does. For example, if the array is a list of fruits, like this:

```
db.food.insertOne({"fruit" : ["apple", "banana", "peach"]})
```

- The following query will successfully match the document:

```
db.food.find({"fruit" : "banana"})
```

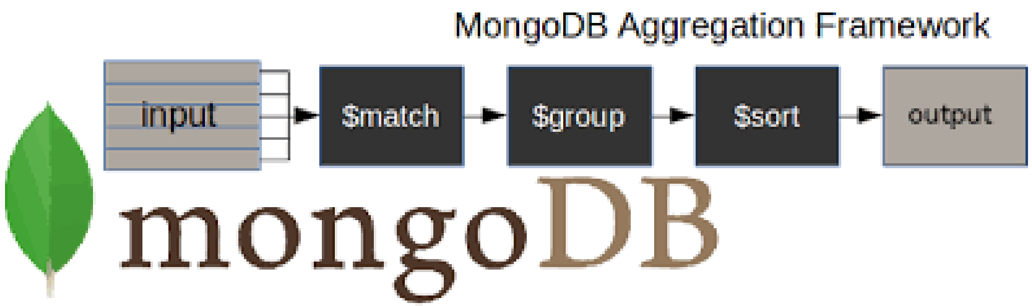
Querying on Embedded Documents

```
{
    "name" : {
        "first" : "Joe",
        "last" : "Schmoe"
    },
    "age" : 45
}
```

```
db.people.find({"name.first" : "Joe", "name.last" :
"Schmoe"})
```

aggregate() Method

- The aggregate() method uses the aggregation pipeline to process documents into aggregated results



Example

```
db.train.aggregate( [
                    {$match:{class:"first-class"}},
                    {$group:{_id:"id",total:{$sum:"$fare"}}}
                  ])
```

} pipeline stages

```
{
    id:"181",
    class:"first-class",
    fare: 1200
}
{
    id:"181",
    class:"first-class",
    fare: 1000
}
{
    id:"181",
    class:"second-class",
    fare: 1000
}
{
    id:"167",
    class:"first-class",
    fare: 1200
}
{
    id:"167",
    class:"second-class",
    fare: 1500
}
```

⇒ $match

```
{
    id:"181",
    class:"first-class",
    fare: 1200
}
{
    id:"181",
    class:"first-class",
    fare: 1000
}
{
    id:"167",
    class:"first-class",
    fare: 1200
}
```

⇒ $group

```
{
    _id:"181",
    total: 2200
}
{
    _id:"167",
    total: 1200
}
```

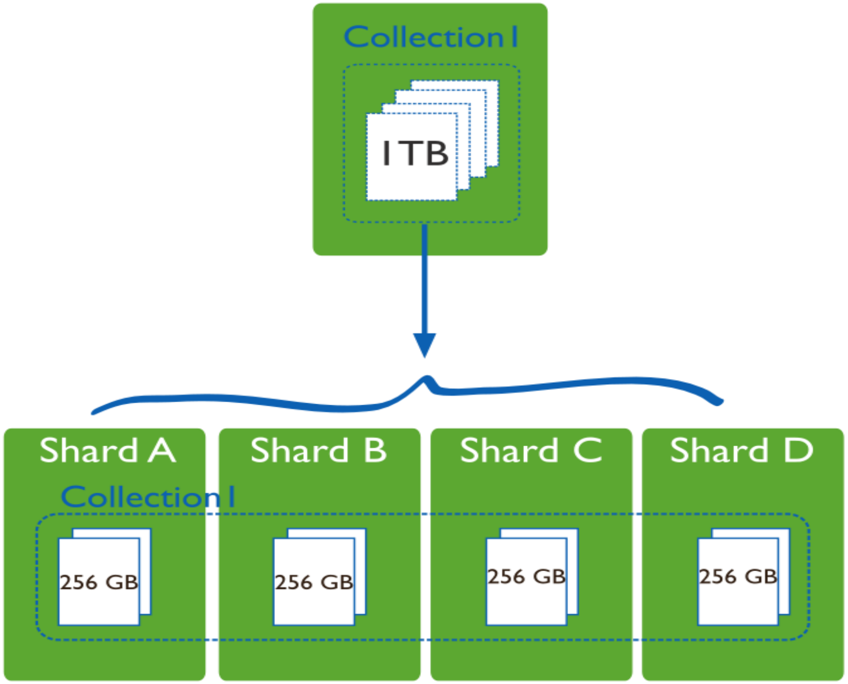
https://www.geeksforgeeks.org/aggregation-in-mongodb/

Accumulators

- **sum:** It sums numeric values for the documents in each group
- **count:** It counts total numbers of documents
- **avg:** It calculates the average of all given values from all documents
- **min:** It gets the minimum value from all the documents
- **max:** It gets the maximum value from all the documents
- **first:** It gets the first document from the grouping
- **last:** It gets the last document from the grouping

Sharding

- *Sharding* refers to the process of splitting data up **across machines**; the term *partitioning* is also sometimes used to describe this concept
- It becomes possible to store more data and handle more load



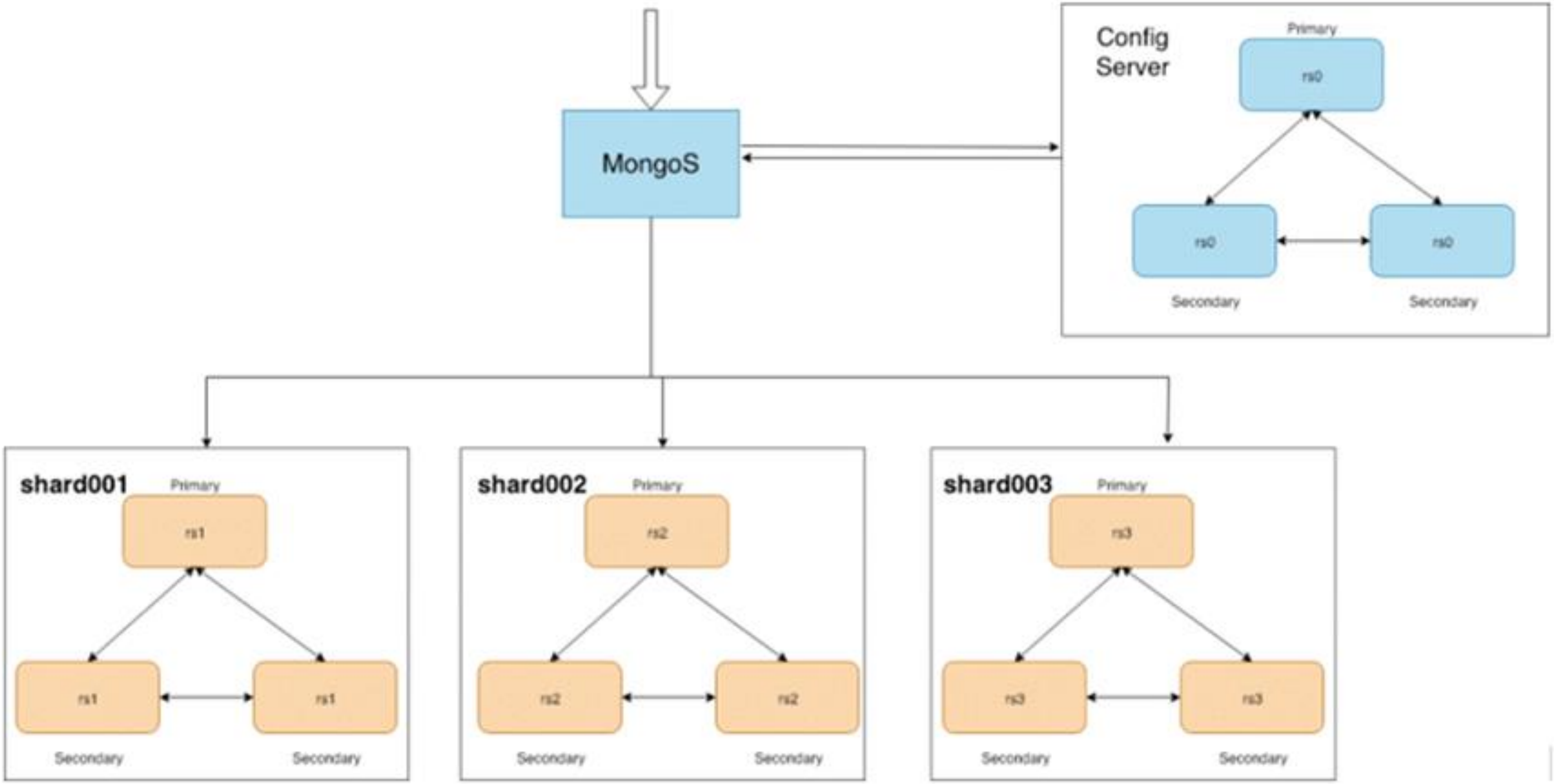
When to Shard

- Increase available RAM
- Increase available disk space
- Reduce load on a server
- Read or write data with greater throughput than a single *mongod* can handle

MongoDB Sharding

- MongoDB supports autosharding, which tries to both abstract the architecture away from the application and simplify the administration of such a system
- MongoDB automates balancing data across shards and makes it easier to add and remove capacity

MongoDB Sharding



Sharding on a Single-Machine Cluster

- We'll start by setting up a quick cluster on a single machine. First, start a *mongo* shell with the --nodb and --norc options: $ mongo --nodb –norc
- Run the following in the *mongo* shell you just launched

```
st = ShardingTest({
  name:"one-min-shards",
  chunkSize:1,
  shards:2,
  rs:{
    nodes:3,
    oplogSize:10
  },
  other:{
    enableBalancer:true
  }
});
```

Connect to Mongos

- Next, you'll connect to the *mongos* to play around with the cluster. Your entire cluster

  $ mongo –nodb

- Use this shell to connect to your cluster's *mongos*.
- Again, your *mongos* should be running on port 20009:
- `db = (new Mongo("localhost:20009")).getDB("accounts")`

Sharding on a Single-Machine Cluster

- Start by inserting some data:

```
> for (var i=0; i<10000; i++) {
db.users.insert({"username" : "user"+i, "created_at" :
new Date()});}
> db.users.count()
10000
```

- As you can see, interacting with *mongos* works the same way as interacting with standalone server does
- You can get an overall view of your cluster by running sh.status(). It will give you a summary of your shards, databases, and collections:

Enable Sharding

- To shard a particular collection, first enable sharding on the collection's database:

  ```
  sh.enableSharding("accounts")
  ```

- When you shard a collection, you choose a shard key. For example, if you chose to shard on "username", MongoDB would break up the data into ranges of usernames

Sharding Collection

- To even create a shard key, the field(s) must be indexed. You have to create an index on the key you want to shard by:

```
db.users.createIndex({"username" : 1})
```

- Now you can shard the collection by "username":

```
sh.shardCollection("accounts.users",
{"username" : 1})
```

The collection has been split up into 13 chunks, where each chunk is a subset of your data.

Sharding key

- Sharding is per-collection and range-based
- The highest-impact choice you make is the shard key:
  - Random keys: good for writes, bad for reads
  - Right-aligned index: bad for writes
  - Small # of discrete keys: *very* bad

  Ideal: balance writes, make reads *routable* by mongos. *Optimal shard key selection is hard*

Choosing a Shard Key

- The most common ways people choose to split their data are via:
  - Ascending
  - Random
  - Location-based keys

Ascending Shard Keys

- Ascending shard keys are generally something like a "date" field or ObjectId—anything that *steadily increases over time*

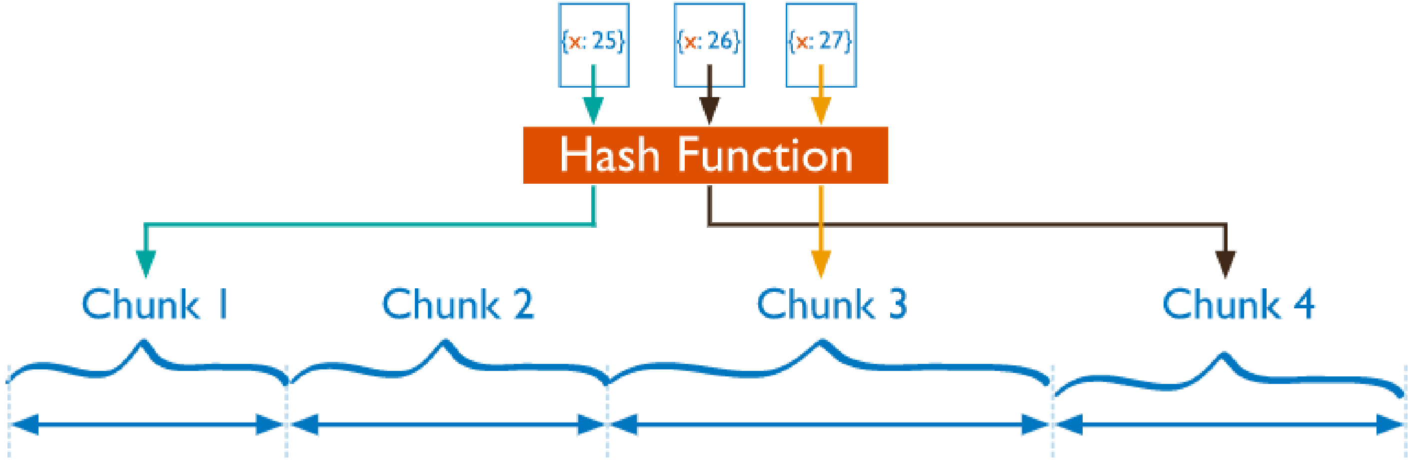
$minKey -> ObjectId("5112fa61b4a4b396ff960262")
ObjectId("5112fa61b4a4b396ff960262") -> ObjectId("5112fa9bb4a4b396ff96671b")
ObjectId("5112fa9bb4a4b396ff96671b") -> ObjectId("5112faa0b4a4b396ff9732db")
ObjectId("5112faa0b4a4b396ff9732db") -> ObjectId("5112fabbb4a4b396ff97fb40")
ObjectId("5112fabbb4a4b396ff97fb40") -> ObjectId("5112fac0b4a4b396ff98c6f8")
ObjectId("5112fac0b4a4b396ff98c6f8") -> ObjectId("5112fac5b4a4b396ff998b59")
ObjectId("5112fac5b4a4b396ff998b59") -> ObjectId("5112facab4a4b396ff9a56c5")
ObjectId("5112facab4a4b396ff9a56c5") -> ObjectId("5112facfb4a4b396ff9b1b55")
ObjectId("5112facfb4a4b396ff9b1b55") -> ObjectId("5112fad4b4a4b396ff9bd69b")
ObjectId("5112fad4b4a4b396ff9bd69b") -> ObjectId("5112fae0b4a4b396ff9d0ee5")

Randomly Distributed Shard Keys

- Randomly distributed keys could be usernames, email addresses, UUIDs, MD5 hashes, or any other key that has no identifiable pattern in your dataset
- As writes are randomly distributed, the shards should grow at roughly the same rate, limiting the number of migrates that need to occur.

Hashed Shard Key

- A hashed shard key can make any field randomly distributed, so it is a good choice
- *The trade-off is that you can never do a targeted range query with a hashed shard key*. If you will not be doing range queries, though, hashed shard keys are a good option.



Hashed Shard Key

- To create a hashed shard key, first create a hashed index:

```
> db.users.createIndex({"username" : "hashed"})
```

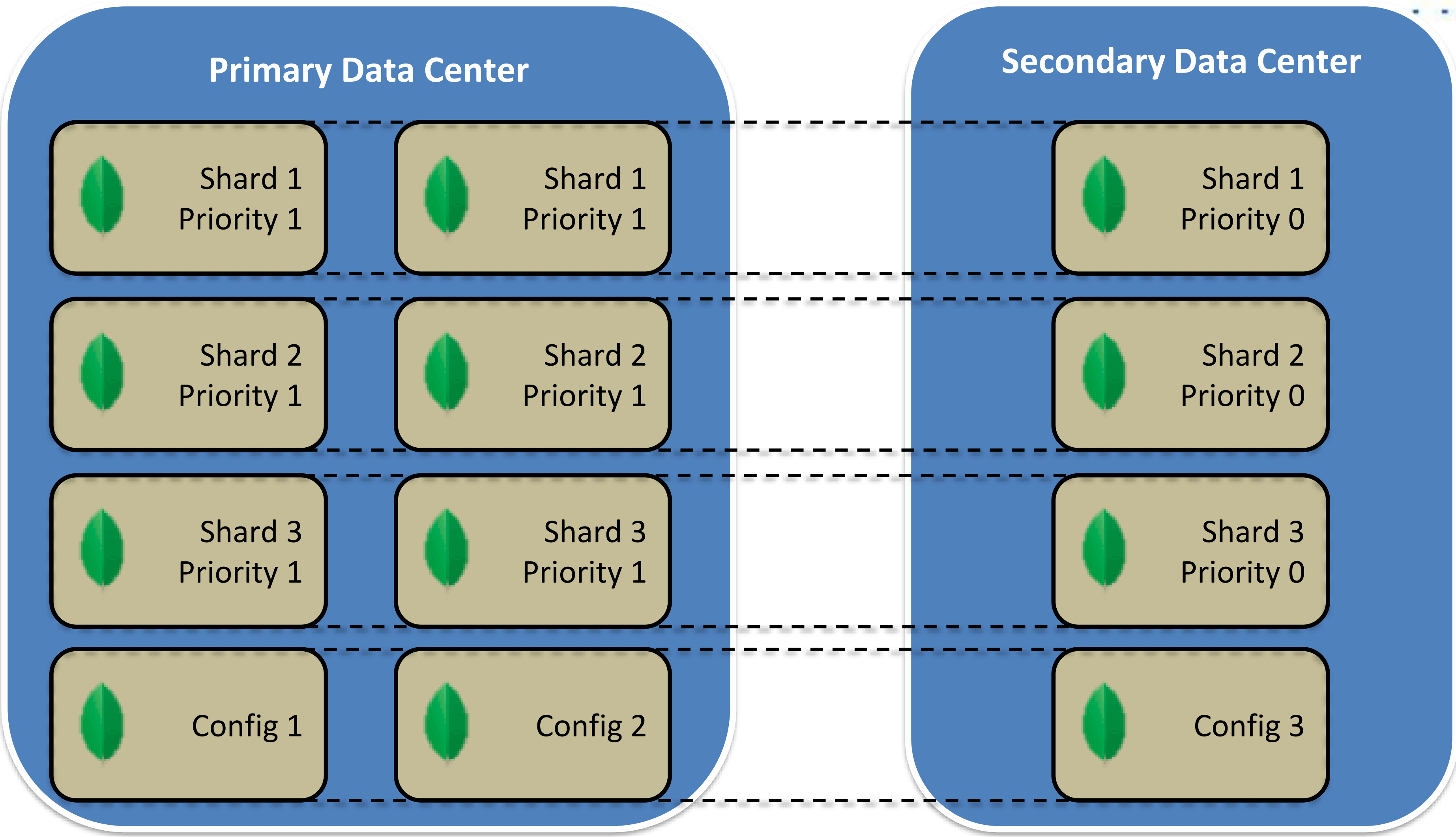
- Next, shard the collection with:

```
> sh.shardCollection("app.users", {"username" : "hashed"})
```

Location-Based Shard Keys

- A location-based key is a key where documents with *some similarity fall into a range based on this field*.
- This can be handy for both putting data close to its users and keeping related data together on disk

Sharding setup example



THANKS YOU