WiXEngineering

MongoDB Database Architecture & Administration Fundamentals

by
Baruch Assif Osoveskiy

baruchao@wix.com

baruch assif osoveskiy

Baruch Assif osoveskiy

- Senior dba @ wix.com
- Linux/UNIX SYSADMIN (from 1999)
- DBA From 2000.
- Talk about databases from 2012.
- DBA – Oracle (Ret), MySQL, MongoDB, Elasticsearch, Cassandra, PostgreSQL

Code will be in github :

https://github.com/barucho/gw2019

You can send me interesting bugs @:

baruchao@wix.com

Find more presentations:

http://www.slideshare.net/baruchosoveskiy

My blog

https://barucho.github.io/blog/

At the end of each section will be Q&A slide you can ask question I will try to answer

AGENDA

- Introduction
- A talk about Data Base Technology's
- JSON/BSON
- Introduction to Mongo DB
- Mongo DB Architecture.
- Quick installation Guide
- CRUD
- Mongo Indexes
- Replication
- Sharding
- Backup And Monitoring
- Aggregations
- Data modeling Introduction
- Administration Best practice

I Will **Not** Talk About

Performance.

Other Databases vs Mongo DB.

Mongo DB as a service like Atlas , RDS.

Whey the database you are using is better.

Windows vs Linux.

For This Talk

MongoDB 4.x+.

Linux Installation.

I will run all demos on centos 7.6

WiXEngineering

Q&A

01

Database Technology

ACID

ACID properties

Atomic – In a transaction with two or move pieces of information, either all the information is committed to save, or none is saved. Essentially, an “all or nothing” rule is followed.

Consistent – The data saved can’t violate any of the database’s integrity. Interrupted changes are rolled back to ensure the database is placed in a state prior to the change.

Isolation – The transaction in question is not affected by any other transactions taking place. This avoids “mid-air collisions.”

Durable – Once the transaction is committed, any failure or system restart, returns the data in a correct state. Stated another way, once a transaction is committed, it will remain so, regardless of a subsequent system failure..

CAP theorem

CAP theorem

In distributed system:

- **Consistency:** This condition states that all data see the same data at the same time
- **Availability:** This condition states that every request gets a response on success/failure.
- **Partition Tolerance:** A system that is partition-tolerant can sustain any amount of network failure that doesn’t result in a failure of the entire network.

CAP theorem

- The CAP theorem implies that in the presence of a network partition, one has to choose between consistency and availability.

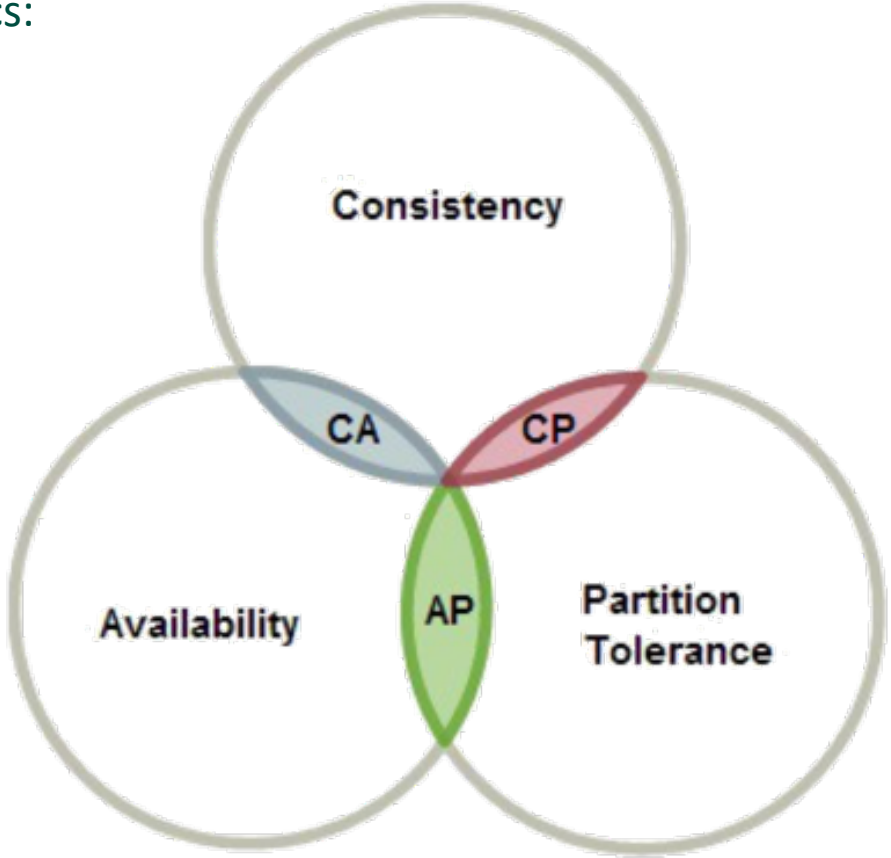
- Note that consistency as defined in the CAP theorem is quite different from the consistency guaranteed in ACID database transactions.

CAP theorem

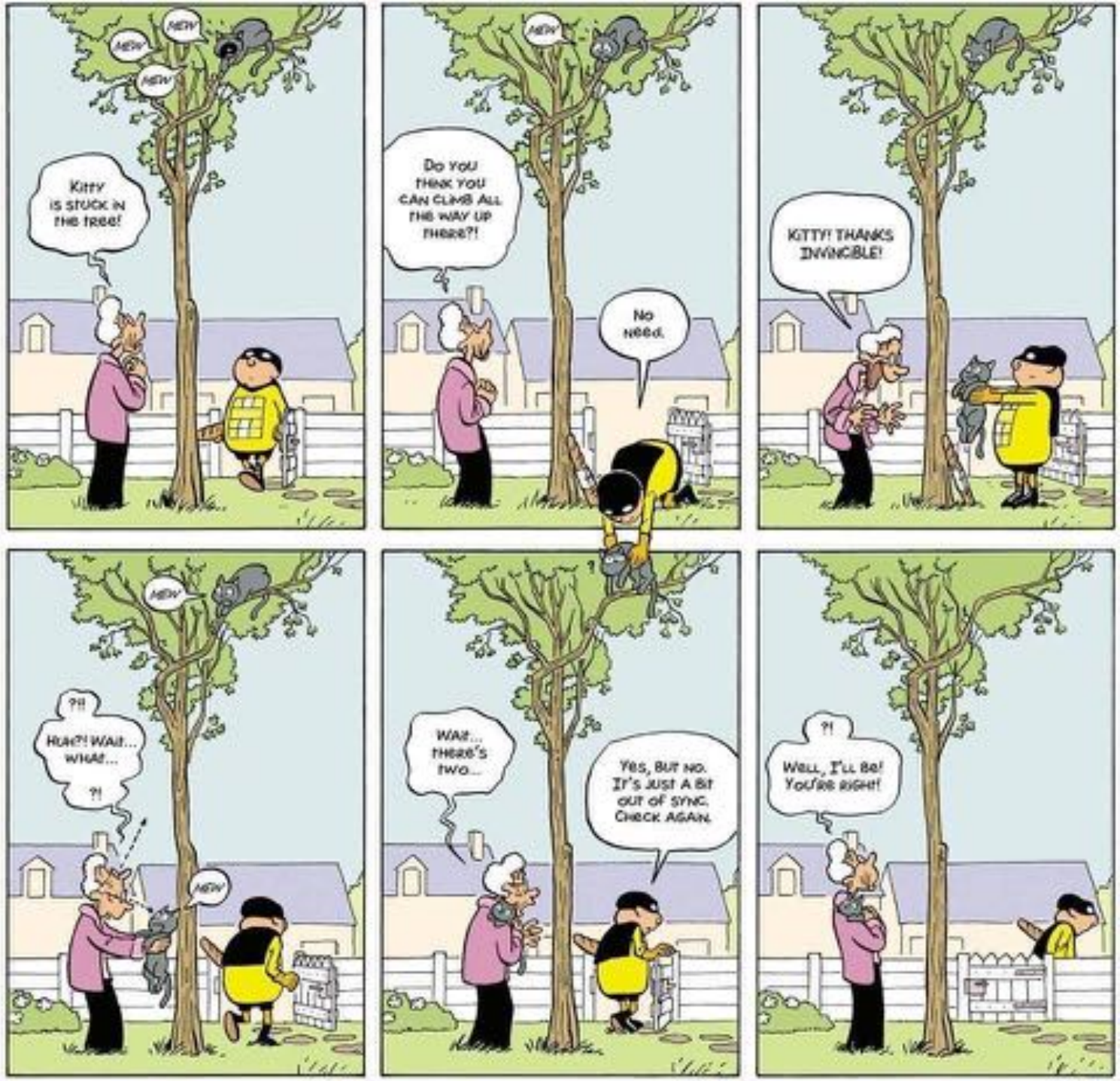
The CAP theorem states that in a distributed/partitioned application,

you can only pick two of the following three characteristics:

- **C**onsistency.
- **A**vailability.
- **P**artition Tolerance.



eventually consistent



The V^s

3Vs concept

Gartner analyst Doug Laney introduced the 3Vs concept in a 2001 MetaGroup research publication

https://www.datasciencecentral.com/forum/topics/the-3vs-that-define-big-data

Volume

The Size/Amount of the data

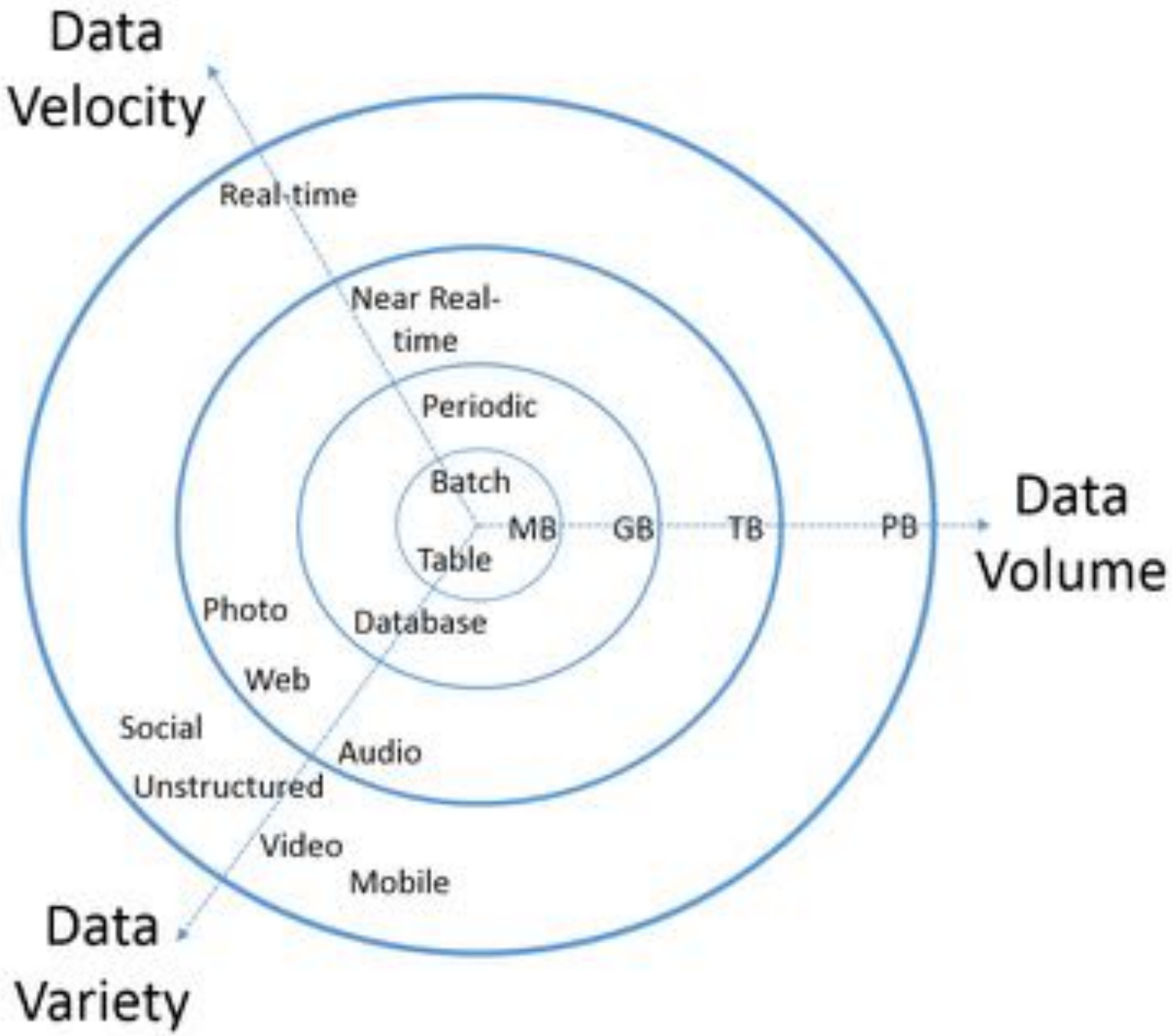
- The RDBMS model provides strong consistency (ACID)
- You can either pick Availability or Partition tolerance.
- Because availability is very important (ACID) , scaling out becomes problematic scale it out can be very expensive.

- Scaling-out reads (selects) is very difficult (replication,clusters).
- Scaling-out writes (between machines) need cluster and shared storage.

Velocity

The rate of data insert/select

- The RDBMS model has scalability problems.
- Due to the ACID property of RDBMS transactions commit process can be slow.


Data Velocity
Real-time
Near Real-time
Periodic
Batch
MB
GB
TB
PB
Table
Data Volume
Photo
Database
Web
Social
Audio
Unstructured
Video
Mobile
Data Variety

VARIETY

The amount of different data types

- The RDBMS model are not very good with different data types
- Lets Try to add different data types to table

ID	NAME
1	bill
2	larry

SELECT ID,NAME FROM TABLE WHERE ID = 1

ID	NAME
1	bill

Now I need to add EMAIL to the DB:

ALTER TABLE ‘TABLE’ ADD COLUMN EMAIL VARCHAR(200);

ID	NAME	EMAIL
1	bill	bill@hotmail.com
2	larry	Larry@gmail.com

SELECT ID,NAME,EMAIL FROM TABLE WHERE ID = 1

ID	NAME	EMAIL
1	bill	bill@hotmail.com

JSON “Alter table”

In NoSQL like databases we can just modify the document without changing the schema

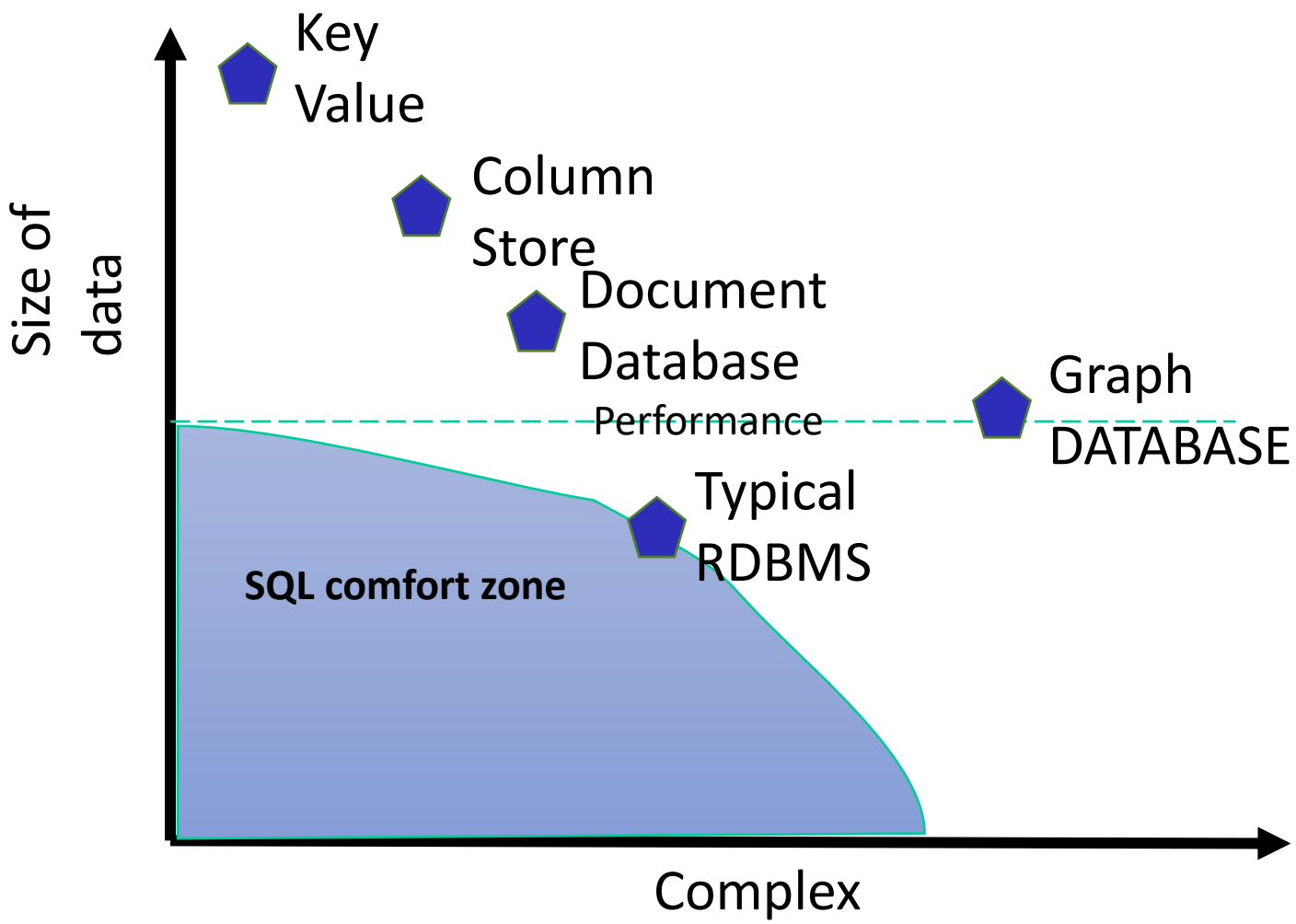
```
db.users.insertOne({_id:0,name:"bill"})

db.users.updateOne(
{_id:{$eq:0}},
{$set :{email :"bill@gmail.com"}}
)
```

NoSQL Database Types

Elasticsearch is not DATABASE

https://www.elastic.co/blog/found-elasticsearch-as-nosql

DATABASE TYPES

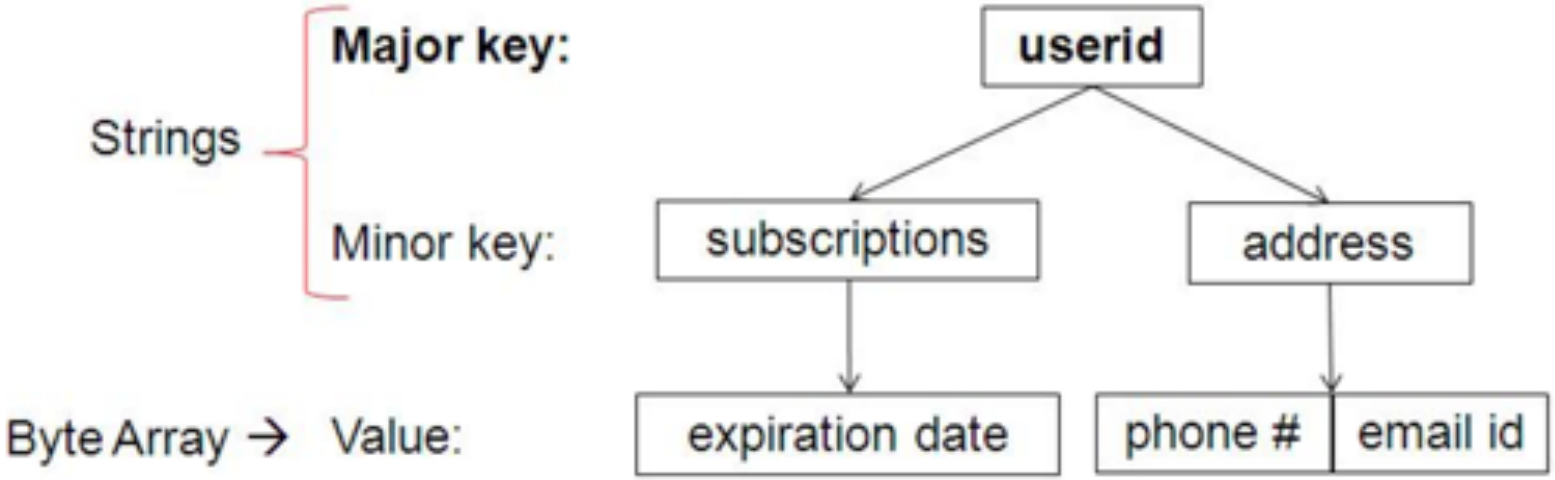


NoSQL Database types

- Key-value store (REDIS,DYNAMODB).
- Column-family store (CASSANDRA).
- Document DATABASE (MONGO).
- Graph DATABASE (not used in WIX).

Key–Value or KV stores

- The key-value stores are usually distributed hash tables.
- They allow to store a unique key with arbitrary object value.
- Usually have linear scalability, can support very large amounts of data and are fault tolerant.



Key–Value or KV stores

Example for implementations:

- MemcacheDB**.**
- **Redis**.
- **Google cloud datastore**
- Oracle KV store Berkeley DB.
- **Amazon Dynamo**
- Project Voldemort
- Tokyo Cabinet (Kyoto)
- **MySQL NDB,INNODB API,MyRocksDB**

Column (row) -Family Stores

RDBMS Table:

EmpId	Lastname	Firstname	Salary
10	Smith	Joe	40000
12	Jones	Mary	50000
11	Johnson	Cathy	44000
22	Jones	Bob	55000

Column Stores:

10:001,**12**:002,**11**:003,**22**:004;**Smith**:001,**Jones**:002,**Johnson**:003,**Jones**:004;**Joe**:001,**Mary**:002,**Cathy**:003,**Bob**:004;**40000**:001,**50000**:002,**44000**:003,**55000**:004;

Column (row) -Family Stores

Example for Column store DATABASES :

- ClickHouse
- Oracle column store
- Mariadb column engine
- **Cassandra (?)**
- Greenplum
- vertica

Document store

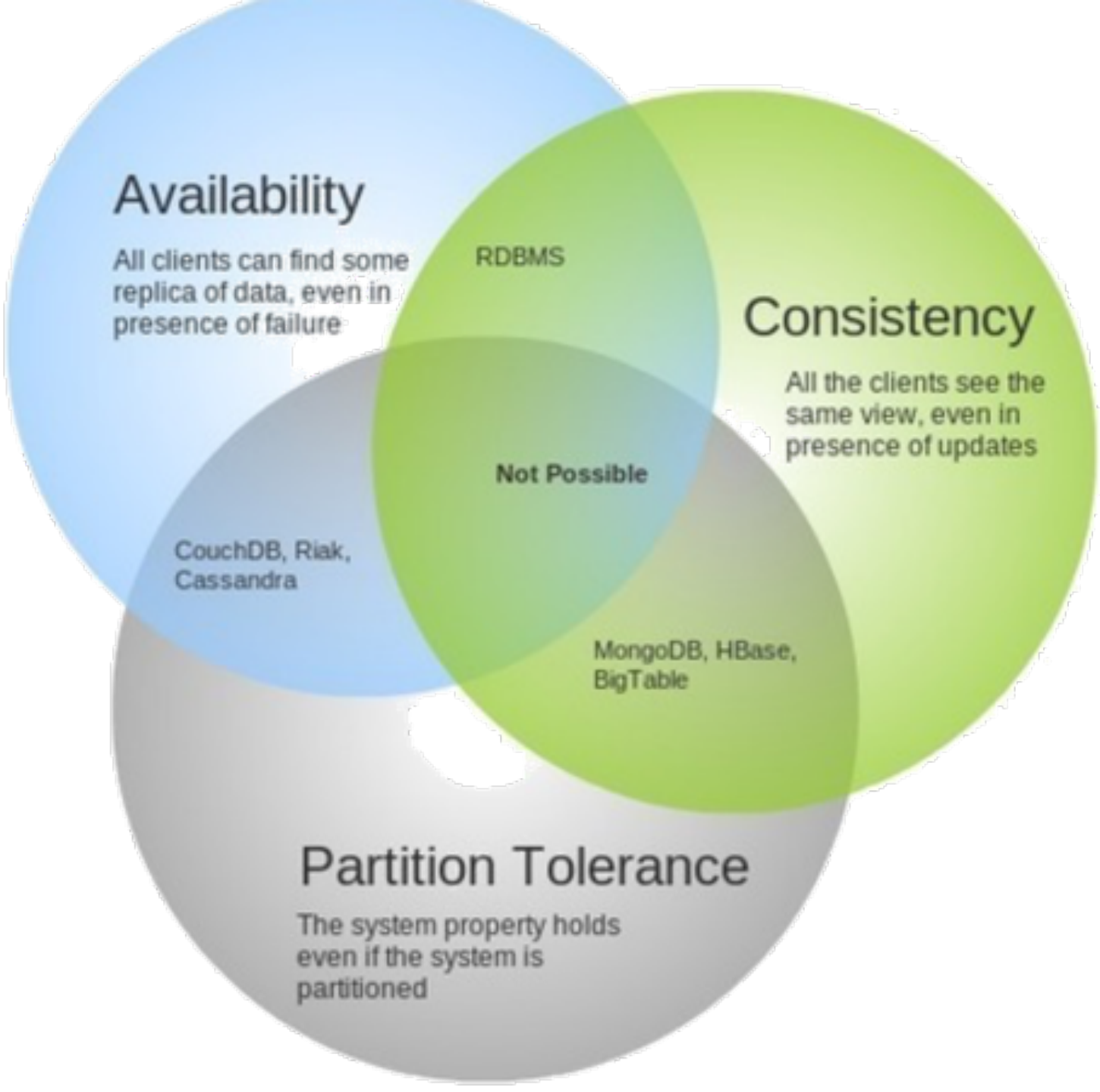
Example for document store DATABASES :

- **MongoDB** (concurrency with Update-in-Place).
- CouchDB (concurrency with MVCC).

Task specific Databases

We also have more types of database that are task specific like:

- Graph databases (Neo4j,ArangoDB)
- Time series database (Graphit , Grafana, Promotions)


Availability
All clients can find some replica of data, even in presence of failure
RDBMS
Consistency
All the clients see the same view, even in presence of updates
Not Possible
CouchDB, Riak, Cassandra
MongoDB, HBase, BigTable
Partition Tolerance
The system property holds even if the system is partitioned

02 — JSON/BSON

Document(JSON) structure

The document has simple structure and very easy to understand the content

JSON is smaller, faster and lightweight compared to XML.

For data delivery between servers and browsers, JSON is a better choice

Easy in parsing, processing, validating in all languages

JSON can be mapped more easily into object oriented system.

BSON

“Binary JSON”

Binary-encoded serialization of JSON-like docs

Also allows “referencing”

Embedded structure reduces need for joins

Goals:

- Lightweight
- Traversable
- Efficient (decoding and encoding)

http://bsonspec.org/

BSON Data Types

Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular Expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

The number can be used with the $type operator to query by type!

JSON example

```
{
"_id": ObjectId(7df78ad8902c),
"title": "MongoDB Overview",
"description": "MongoDB is no sql database",
"by": "tutorials point",
"url": "http://www.tutorialspoint.com",
"tags": ["mongodb", "database", "NoSQL"],
"likes": 100,
"comments": [
{
"user":"user1",
"message": "My first comment",
"dateCreated": new Date(2011,1,20,2,15),
"like": 0
},
{
"user":"user2",
"message": "My second comments",
"dateCreated": new Date(2011,1,25,7,45),
"like": 5
}
]
```

The _id Field

By default, each document contains an _id field. This field has a number of special characteristics:

- _id is a 12 bytes hexadecimal number
- Value serves as primary key for collection.
- Value is unique, immutable, and may be any non-array type.
- Default data type is ObjectId, which is “small, likely unique, fast to generate, and ordered.” Sorting on an ObjectId value is roughly equivalent to sorting on creation time.

The _id Field

You can provide _id while inserting the document

If you didn't provide then MongoDB provide a unique id for every document.

- These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id,
- next 2 bytes for process id of mongodb server
- remaining 3 bytes are simple incremental value.

WiXEngineering

Q&A

03

Introduction to Mongo DB

What is MongoDB ?

MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability.
MongoDB works on concept of collection and document

- MongoDB offers both a Community and an Enterprise version
- Built for Speed
- Rich Document based queries for Easy readability.
- Replication and Failover for High Availability.
- Auto Sharding for Easy Scalability.
- Map / Reduce for Aggregation.

Why should use MongoDB

- Document Oriented Storage : Data is stored in the form of JSON style documents
- Vary Large Datasets (more then 5TB)
- Index on any attribute **
- Replication & High Availability
- Auto-Sharding
- Rich Queries **
- Fast In-Place Updates
- Professional Support By MongoDB

MongoDB API vs SQL

mongoDB	SQL
Document	Tuple/DataBase
Collection	Table/View
PK: _id Field	PK: Any Attribute(s)
Uniformity not Required	*Uniform Relation Schema
Index	Index
Embedded Structure	Joins
Shard	Partition

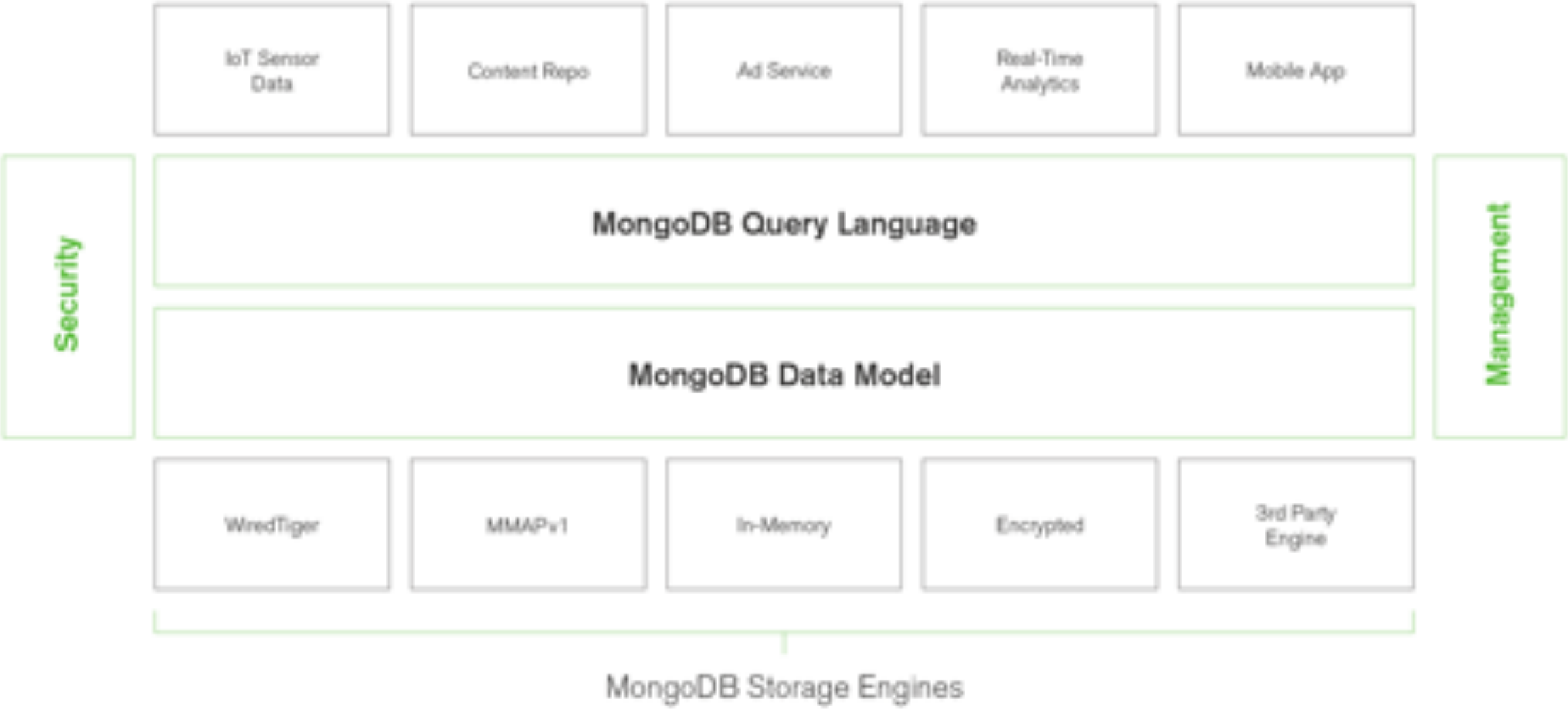
RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)
Database Server and Client	
Mysqld/Oracle	mongod
mysql/sqlplus	mongo

04 — MongoDB ARCHITECTURE

MongoDB Architecture

- MongoDB is a cross-platform, document oriented database.
- Written in CPP and GO.
- Single Process Multi threaded
- Support for Multiple Storage Engines
  - WiredTiger Storage Engine (including support for Encryption at Rest)
  - In-Memory Storage Engine
  - MMAPv1 Storage Engine (Deprecated in MongoDB 4.0)
  - Include storage engine API .

- Listen to Single port (27017 default)


IoT Sensor Data
Content Repo
Ad Service
Real-Time Analytics
Mobile App
Security
MongoDB Query Language
MongoDB Data Model
Management
WiredTiger
MMAPv1
In-Memory
Encrypted
3rd Party Engine
MongoDB Storage Engines

Horizontal Scalable

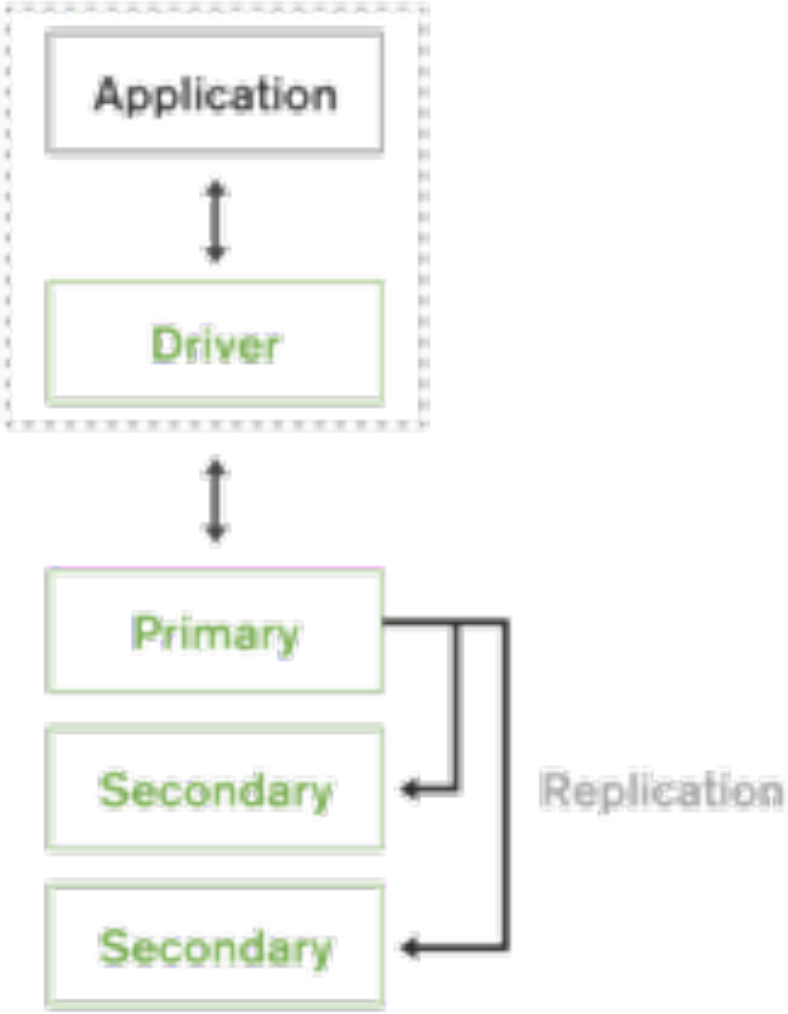
Three types: hash-based, range-based, location-aware

Increase or decrease capacity as you go

Automatic balancing

mongoDB

Replica Set



Replica Set – 2 to 50 copies

Self-healing shard

Data Center Aware

Addresses availability considerations:

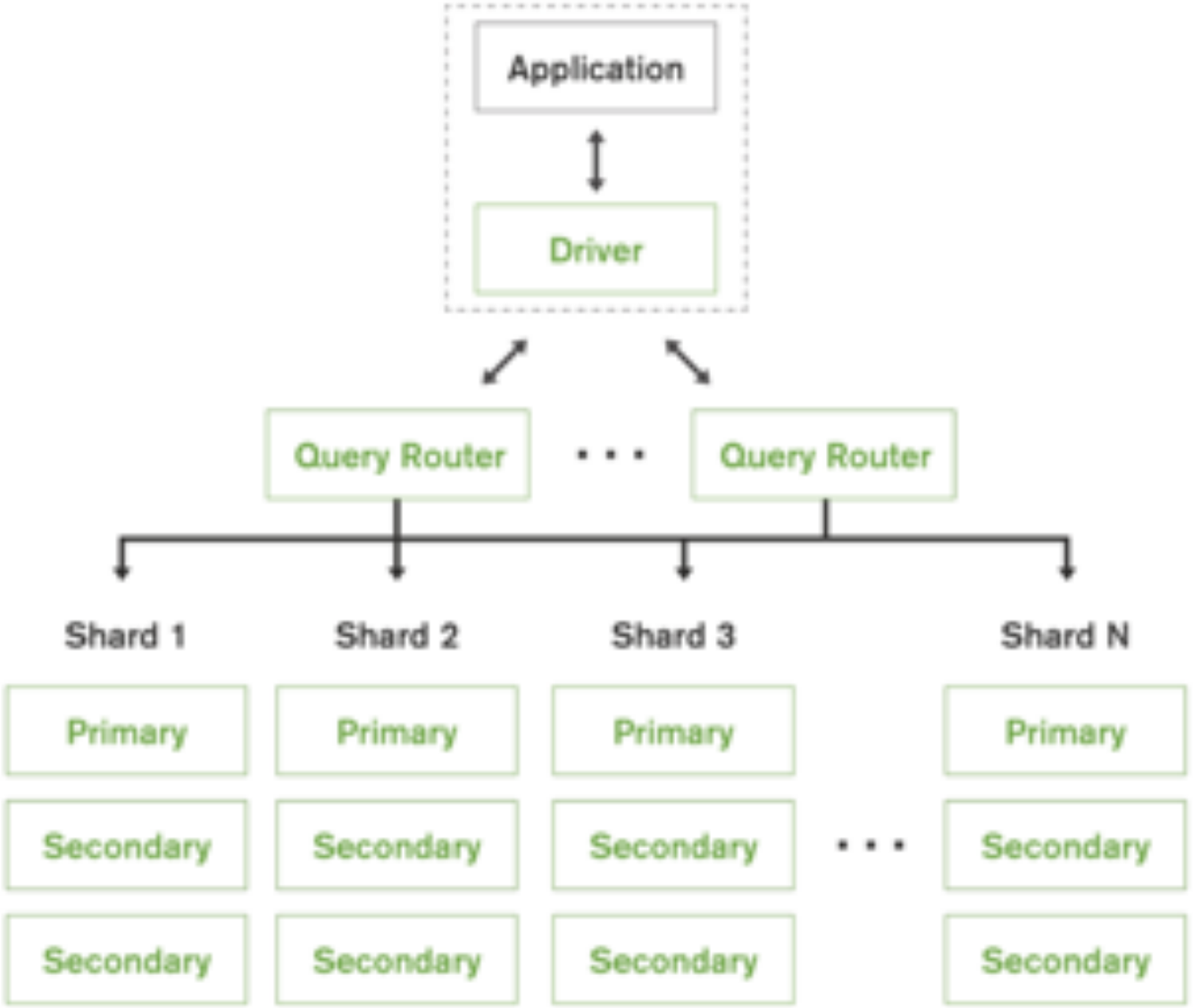
High Availability

Disaster Recovery

Maintenance

Workload Isolation: operational & analytics

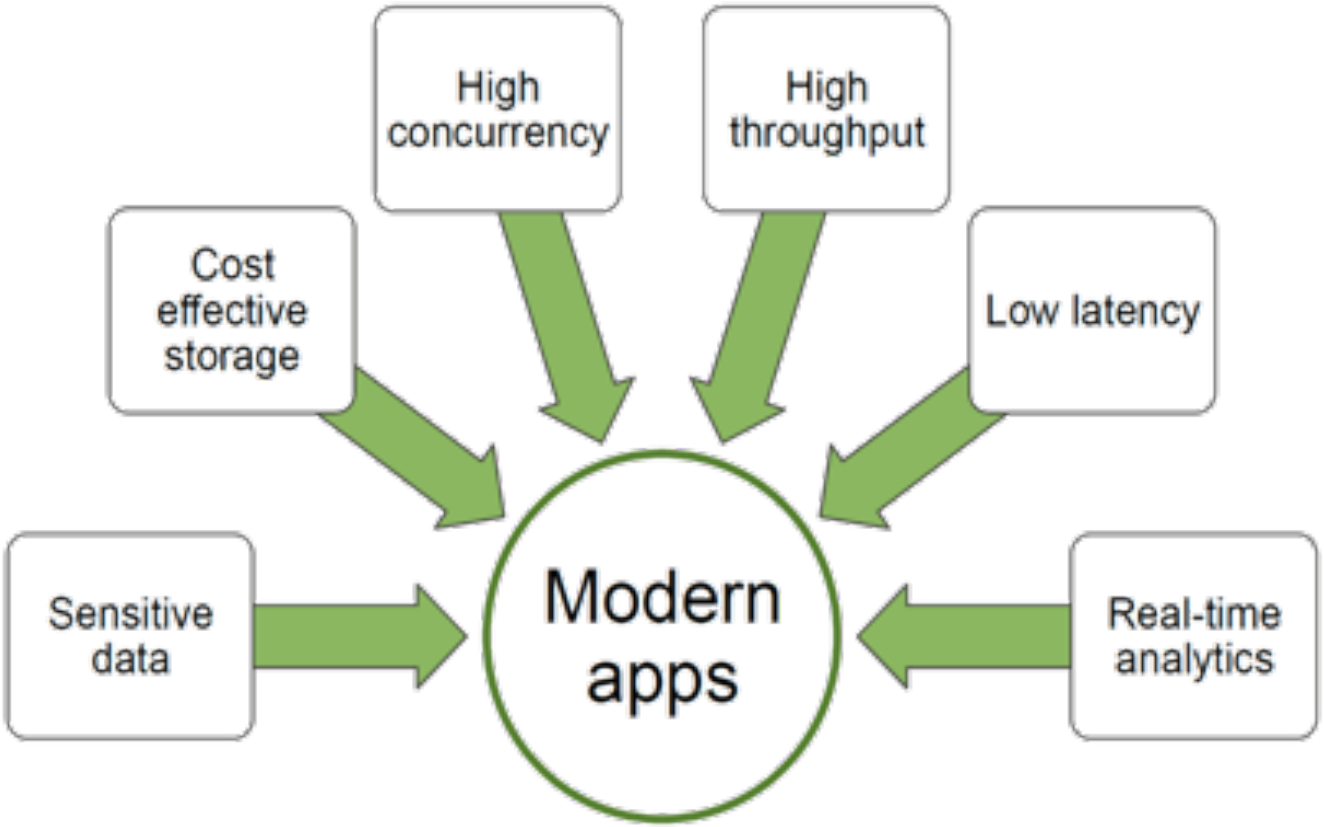
Distributed Database



mongoDB

Storage Engines

- is the primary component of MongoDB responsible for managing data. MongoDB provides a variety of storage engines, allowing you to choose one most suited to your application.



Storage Engines API

- Allows to "plug-in" different storage engines
  - Different use cases require different performance characteristics
  - mmapv1 is not ideal for all workloads
  - More flexibility
- Can mix storage engines on same replica set/sharded cluster
- Opportunity to integrate further (HDFS, native encrypted, hardware optimized …)

In-Memory

- Handle ultra-high throughput with low latency and high availability
- the in-memory storage engine does not maintain any on-disk data, including configuration data, indexes, user credentials, etc.
- Multi-document transactions are not available for deployments with members that use in-memory storage engine.
- In-Memory can be used with other storage engines in sharded cluster
- The in-memory storage engine does not persist data shutdown.

- https://docs.mongodb.com/manual/core/inmemory/

MongoRocks

- MongoRocks is a storage engine for MongoDB based on the RocksDB key-value store optimized for fast storage. It is developed by Facebook and designed to handle write-intensive workloads.

https://www.percona.com/doc/percona-server-for-mongodb/3.4/mongorocks.html#using-mongorocks

In-Memory use in YAML file

```yaml
storage:
engine: inMemory
dbPath: <path>
inMemory:
engineConfig:
inMemorySizeGB: <newSize>
```

Encrypted Storage Engine

- Reduces the management and performance overhead of external encryption mechanisms
- AES-256 Encryption, FIPS 140-2 option available
- Key management: Local key management via keyfile or integration with 3rd party key management
- appliance via KMIP Based on WiredTiger storage engine
- **Requires MongoDB Enterprise Advanced**

WiredTiger

- WiredTiger is the **default storage** engine starting in MongoDB 3.2.
- It is well-suited for most workloads and is recommended for new deployments.
- WiredTiger provides a **document-level** concurrency model, checkpointing, and compression, among other features.
- WiredTiger also supports ** Encryption at Rest.

** **Requires MongoDB Enterprise Advanced**

WiredTiger

- WiredTiger uses Multi Version Concurrency Control (MVCC).
- At the start of an operation, WiredTiger provides a point-in-time snapshot of the data to the operation.
- snapshot presents a consistent view of the in-memory data.
- When writing to disk, WiredTiger writes all the data in a snapshot
- Snapshot is written to Disk every 60 sec
- The checkpoint ensures that the data files are consistent up to and including the last checkpoint
- Checkpoints can act as recovery points.
- MongoDB can recover from the last valid checkpoint.

Journal

- To provide durability in the event of a failure, MongoDB uses write ahead logging to on-disk journal files.
- Journal persists all data modifications between checkpoints
- If MongoDB exits between checkpoints, it uses the journal to replay all data modified since the last checkpoint
- WiredTiger journal is compressed using the snappy compression library to change the lib use:

storage.wiredTiger.engineConfig.journalCompressor

WiredTiger use in YAML file

```yaml
storage:
wiredTiger:
engineConfig:
cacheSizeGB: <number>
journalCompressor: <string>
directoryForIndexes: <boolean>
collectionConfig:
blockCompressor: <string>
indexConfig:
prefixCompression: <boolean>
```

Oplog

- Capped collection** that stores an ordered history of logical writes to a MongoDB database. The oplog is the basic mechanism enabling replication in MongoDB.
- MongoDB applies database operations on the primary and then records the operations on the primary's oplog. The secondary members then copy and apply these operations in an asynchronous process
- All replica set members contain a copy of the oplog in the **local.oplog.rs** collection this allows them to maintain the current state of the database.

** **capped collection** - A fixed-sized collection that automatically overwrites its oldest entries when it reaches its maximum size

Oplog sizes

- When you start a replica set member for the first time, MongoDB creates an oplog of a default size if you do not specify the oplog size

For Unix and Windows systems

The default oplog size depends on the storage engine:

Storage Engine	Default Oplog Size	Lower Bound	Upper Bound
In-Memory Storage Engine	5% of physical memory	50 MB	50 GB
WiredTiger Storage Engine	5% of free disk space	990 MB	50 GB
MMAPv1 Storage Engine	5% of free disk space	990 MB	50 GB

Oplog sizes

- Workload that will Might need more oplog size :
  - Updates to Multiple Documents at Once
  - Deletions Equal the Same Amount of Data as Inserts
  - Significant Number of In-Place Updates
  - Adding new server to the cluster

- Get oplog status and size

rs.printReplicationInfo()

GridFS

- GridFS is a specification for storing and retrieving files that exceed the BSON-document size limit of 16 MB.
- Instead of storing a file in a single document, GridFS divides the file into parts, or chunks and stores each chunk as a separate document.
- By default chunk size of 255 kB

- Try not to use mongodb beyond its size limit . Use object storage like s3/google cloud storage etc…

MongoDB Config File

- You can configure mongod and mongos instances at startup using a configuration file
- On Linux default /etc/mongod.conf configuration file is included when using a package manager to install MongoDB.
- YAML-based configuration file format.

Mongod config file demo

```
# mongod.conf

# for documentation of all options, see:
# http://docs.mongodb.org/manual/reference/configuration-options/
# where to write logging data.
systemLog:
destination: file
logAppend: true
path: /var/log/mongodb/mongod.log
# Where and how to store data.
storage:
dbPath: /var/lib/mongo
journal:
enabled: true
# engine:
# wiredTiger:
# how the process runs
processManagement:
fork: true # fork and run in background
pidFilePath: /var/run/mongodb/mongod.pid # location of pidfile
timeZoneInfo: /usr/share/zoneinfo

# network interfaces
net:
port: 27017
bindIp: 10.40.20.100 # Enter 0.0.0.0,:: to bind to all IPv4 and
# IPv6 addresses or, alternatively, use the net.bindIpAll setting.
#security:

#operationProfiling:

#replication:

#sharding:

## Enterprise-Only Options

#auditLog:

#snmp:
```

Using the config file

```
mongod -f /etc/mongod.conf

mongos -f /etc/mongos.conf
```

WiXEngineering

Q&A

05 Installation guide

Production Notes

Windows and Linux are supported - best using Linux.

RHEL flavor Recommended (better io subsystem).

Recommended Platforms from mongodb:

- Amazon Linux
- Debian 8
- RHEL / CentOS 6.2+
- SLES 12
- Ubuntu LTS 16.04
- Windows Server 2012 R2

Allocate Sufficient RAM and CPU

- WiredTiger storage engine is multithreaded and can take advantage of additional CPU cores
  - Throughput increases as the number of concurrent active operations increases up to the number of CPUs.
  - Throughput decreases as the number of concurrent active operations exceeds the number of CPUs by some threshold amount.

The output from mongostat provides statistics on the number of active reads/writes in the (arlaw) column.

WiredTiger

- wiredTigerCacheSizeGB Defines the maximum size of the internal cache that WiredTiger will use for all data.

  - 50% of (RAM - 1 GB)
  - 256 MB.

- container (e.g. lxc, cgroups, Docker, etc.) that does not have access to all of the RAM available in a system, you must set --wiredTigerCacheSizeGB to a value less than the amount of RAM available in the container.

WiredTiger

- MongoDB also needs additional memory outside of this cache for aggregations, sorting, connection management, and the like, so it is important to make sure you leave MongoDB with enough memory to do its work. If not, there is a chance MongoDB can get killed by the OS Out of memory (OOM) killer.

db.serverStatus().wiredTiger.cache

IO

- Separate Components onto Different Storage Devices
  - separate storage devices for storing indexes (fast SSD) storage.wiredTiger.engineConfig.directoryForIndexes
  - To speed the frequent sequential writes that occur to the current journal file, you can ensure that the journal directory is on a different filesystem from the database data files.
- MongoDB recommends using RAID-10 for production deployments

Note Using different storage devices will affect your ability to create snapshot-style backups of your data, since the files will be on different devices and volumes.

FileSystem

- Avoid using EXT2 or EXT3.
- Use EXT4 with the following options:

rw,noatime,nodiratime,nobarrier,data=ordered

- Use **XFS** with the following options:
  - Turn off diratime (nodiratime)
  - Turn off atime (noatime)

SWAP

- Where possible, set swappiness to a value between 1 and 5, and set the deadline I/O scheduler.
- Use SWAP on linux > 7.x (oom_kill.c)

https://lwn.net/Articles/317814/ - Taming the OOM killer

Scheduling

- Scheduling for Virtual or Cloud Hosted Devices
  - guest operating system should use a noop scheduler for best performance. The noop scheduler allows the operating system to defer I/O scheduling to the underlying hypervisor.
- Scheduling for Physical Servers
  - the operating system should use a deadline scheduler. The deadline scheduler caps maximum latency per request and maintains a good disk throughput that is best for disk-intensive database applications.

MongoDB and NUMA Hardware

- Running MongoDB on a system NUMA** can cause a number of operational problems, including slow performance for periods of time and high system process usage.
- MongoDB checks NUMA settings on start up when deployed on Linux (since version 2.0) and Windows (since version 2.6) machines. If the NUMA configuration may degrade performance, MongoDB prints a warning.

** https://queue.acm.org/detail.cfm?id=2513149

Network

- Keep non-mongos processes off of 27017 to avoid mistakes
- /etc/sysctl.conf
  net.ipv4.tcp_keepalive_time = 300

Install_mongodb.md

mongodb101.md

WiXEngineering

Q&A

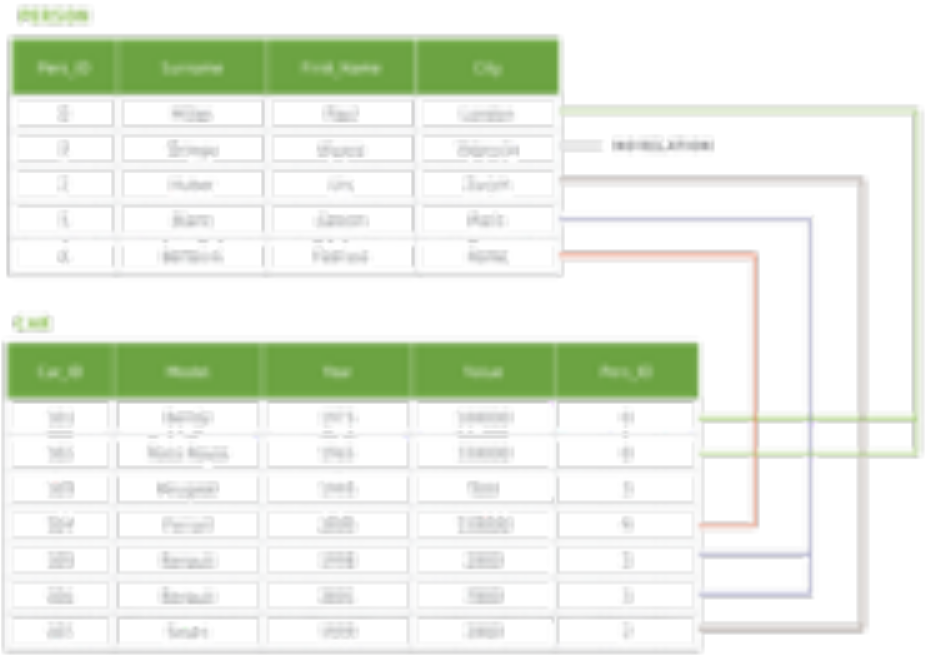
06

CRUD

Create
Read
Update
Delete

Document Data Model

Relational



MongoDB

```
{
  first_name: 'Paul',
  surname: 'Miller',
  city: 'London',
  location:
[45.123,47.232],
  cars: [
    { model: 'Bentley',
      year: 1973,
      value: 100000, … },
    { model: 'Rolls Royce',
      year: 1965,
      value: 330000, … }
  ]
}
```

Document Data Model

```
{
  first_name: 'Paul',
  surname: 'Miller',
  cell: 447557505611,
  city: 'London',
  location: [45.123,47.232],
  Profession: ['banking', 'finance',
'trader'],
  cars: [
    { model: 'Bentley',
      year: 1973,
      value: 100000, … },
    { model: 'Rolls Royce',
      year: 1965,
      value: 330000, … }
  ]
}
```

Fields

String

Number

Geo-Coordinates

Typed field values

Fields can contain arrays

Fields can contain an array of sub-documents

mongoDB

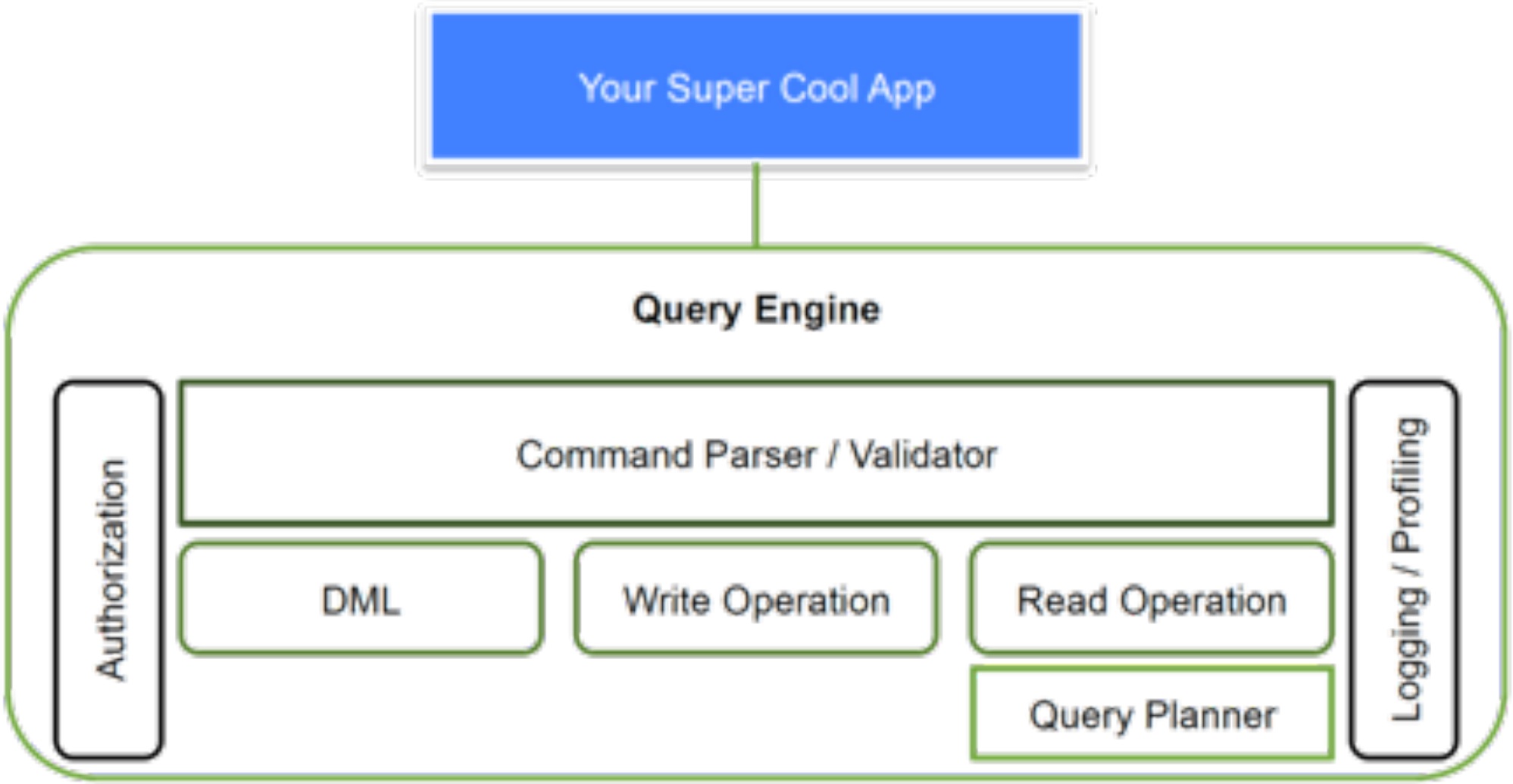
Dynamic Schemas

```
{
    policyNum: 123,
    type: auto,
    customerId: abc,
    payment: 899,

    deductible: 500,
    make: Taurus,
    model: Ford,
    VIN: 123ABC456,
}
```

```
{
    policyNum: 456,
    type: life,
    customerId: efg,
    payment: 240,

    policyValue: 125000,
    start: jan, 1995
    end: jan, 2015
    }
```

```
{
    policyNum: 789,
    type: home,
    customerId: hij,
    payment: 650,

    deductible: 1000,
    floodCoverage: No,
    street: "10 Maple Lane",
    city: "Springfield",
    state: "Maryland"
}
```

Query Engine



Principles of CRUD

CRUD defined as CREATE, READ/RETRIEVE, UPDATE, and DELETE.

CREATE procedures generate new records via INSERT statements.

READ procedures reads the data based on input parameters. Similarly, RETRIEVE procedures grab records based on input parameters.

UPDATE procedures modify records without overwriting them.

DELETE procedures delete where specified.

Read (find)

Read operations retrieves documents from a **collection**; queries a collection for documents.

db.collection.find()

```
db.users.find(
    { age: { $gt: 18 } },
    { name: 1, address: 1 }
).limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

Conditional Operators

$all, $exists, $mod, $ne, $in, $nin, $nor, $or, $size, $type
$lt, $lte, $gt, $gte

```
// find all documents that contain name field
db.teams.find( {name: {$exists: true }} )
// find names using regular expressions
db.names.find( {last: /^Nor*/i } )
// count people by city
db.teams.find( {city: "Madrid"} ).count()
```

Create

Create or insert operations add new **documents** to a **collection**.

If the collection does not currently exist, insert operations will create the collection.

db.collection.insertOne()

db.collection.insertMany()

```
db.users.insertOne(          ← collection
  {
    name: "sue",             ← field: value
    age: 26,                 ← field: value   } document
    status: "pending"        ← field: value
  }
)
```

Create Operations

Create or insert operations **add new documents to a collection**. If the collection does not currently exist, **insert operations will create the collection**.

db.collection.insertOne()

db.collection.insertMany()

All write operations in MongoDB are atomic on the level of a single document.

```
db.users.insertOne(        ← collection
   {
      name: "sue",         ← field: value
      age: 26,             ← field: value   } document
      status: "pending"    ← field: value
   }
)
```

Create Operations

The insert() method is deprecated in major driver so you should use the the .insertOne() method whenever you want to insert a single document into your collection and the .insertMany when you want to insert multiple documents into your collection.

Update

Update operations modify **existing documents** in a collection.

db.collection.updateOne()

db.collection.updateMany()

db.collection.replaceOne()

All write operations in MongoDB are atomic on the level of a single document.

```
db.users.updateMany(                    ← collection
  { age: { $lt: 18 } },                 ← update filter
  { $set: { status: "reject" } }        ← update action
)
```

Write Operators

Field: $set, $unset, $mul, $min, $max ,$inc, $currentDate, $setOnInsert
Array: $push, $pop, $addToSet, $pull, $
Modifiers: $each, $slice, $sort, $posi6on

```
// Increment value by one
db.events.update({_id:1}, {attendees: {$inc: 1}} )
// set array on insert
db.names.update( {name: 'Bryan'}, { $setOnInsert:
{talks:[0,0,0,0]}},
{upsert:true} )
// unset a field
db.names.update( {name: "Bryan"}, {
$unset: {mug_member: "" }})
```

Delete

Delete operations remove **documents** from a collection.

db.collection.deleteOne()

db.collection.deleteMany()

delete operations target a single collection.

All write operations in MongoDB are atomic on the level of a single document.

```
db.users.deleteMany(
   { status: "reject" }
)
```

collection

delete filter

Create index

```
// Index nested documents
db.customers.ensureIndex( “policies.agent”:1 )
db.customers.find({‘policies.agent’:’Fred’})

// geospatial index
db.customers.ensureIndex( “property.location”: “2d” )
db.customers.find( “property.location”
: { $near : [22,42] } )

// text index
db.customers.ensureIndex( “policies.notes”: “text” )
```

crud101.md

WiXEngineering

Q&A

07 indexes

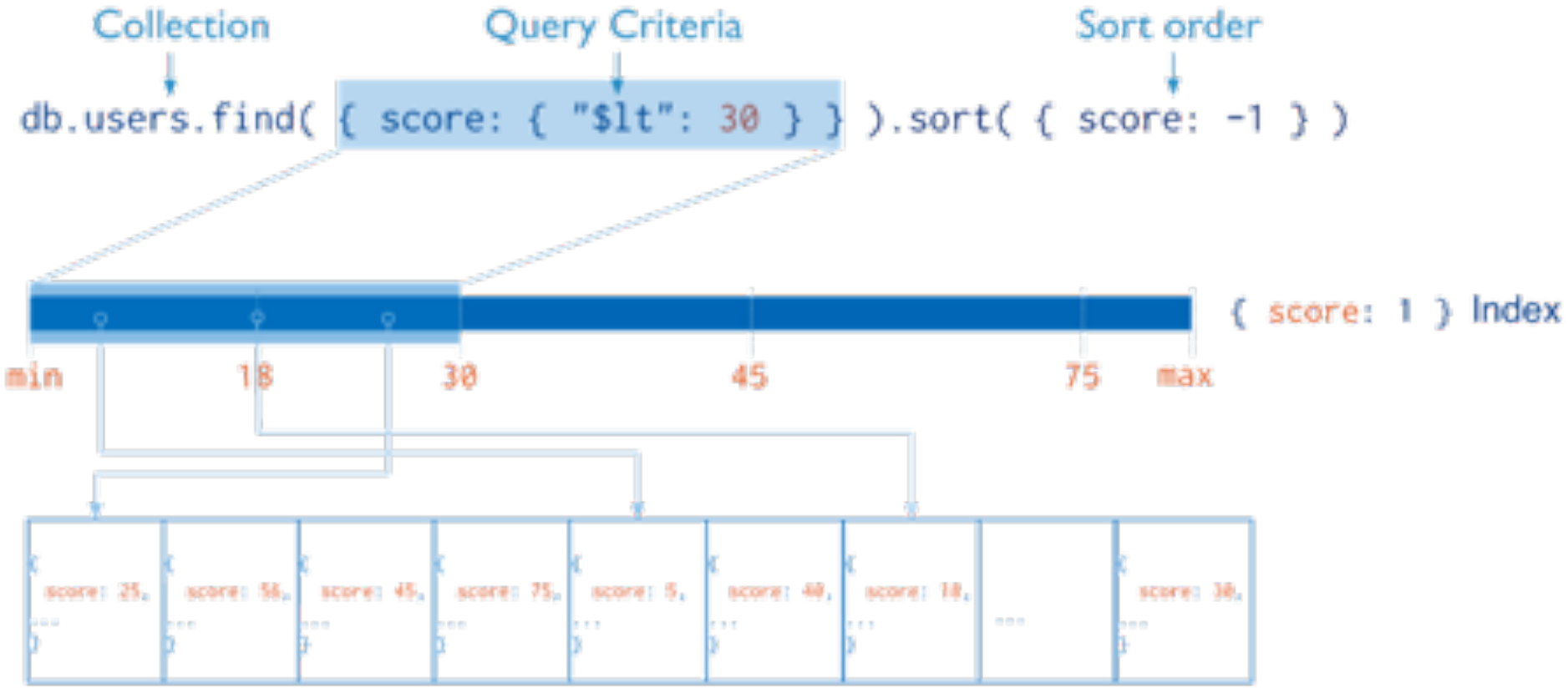
Indexes

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a collection scan (like FTS)

special data structures that store a small portion of the collection's data set in an easy to traverse form

Indexes

The following diagram illustrates a query that selects and orders the matching documents using an index:



Default _id Index

MongoDB creates a **unique index** on the **_id** field during the creation of a collection.

_id index prevents clients from inserting two documents with the same value for the _id field.

You cannot drop this index on the _id field.

Index Types

Single Field: user-defined ascending/descending indexes on a single field of a document

Compound Index: user-defined indexes on multiple fields, i.e. compound indexes. { userid: 1, score: -1 }

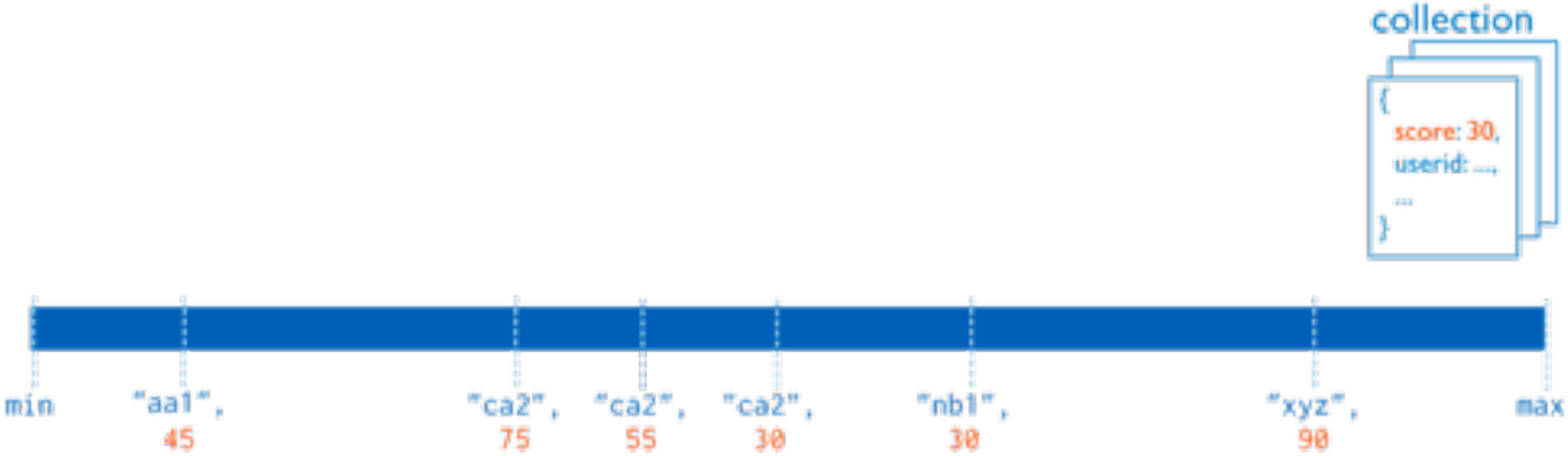
Multikey Index : content stored in arrays. If you index a field that holds an array value, MongoDB creates separate index entries for every element of the array

Geospatial Index: MongoDB provides two special indexes: 2d indexes that uses planar geometry when returning results and 2dsphere indexes that use spherical geometry

Text Indexes: supports searching for string content in a collection

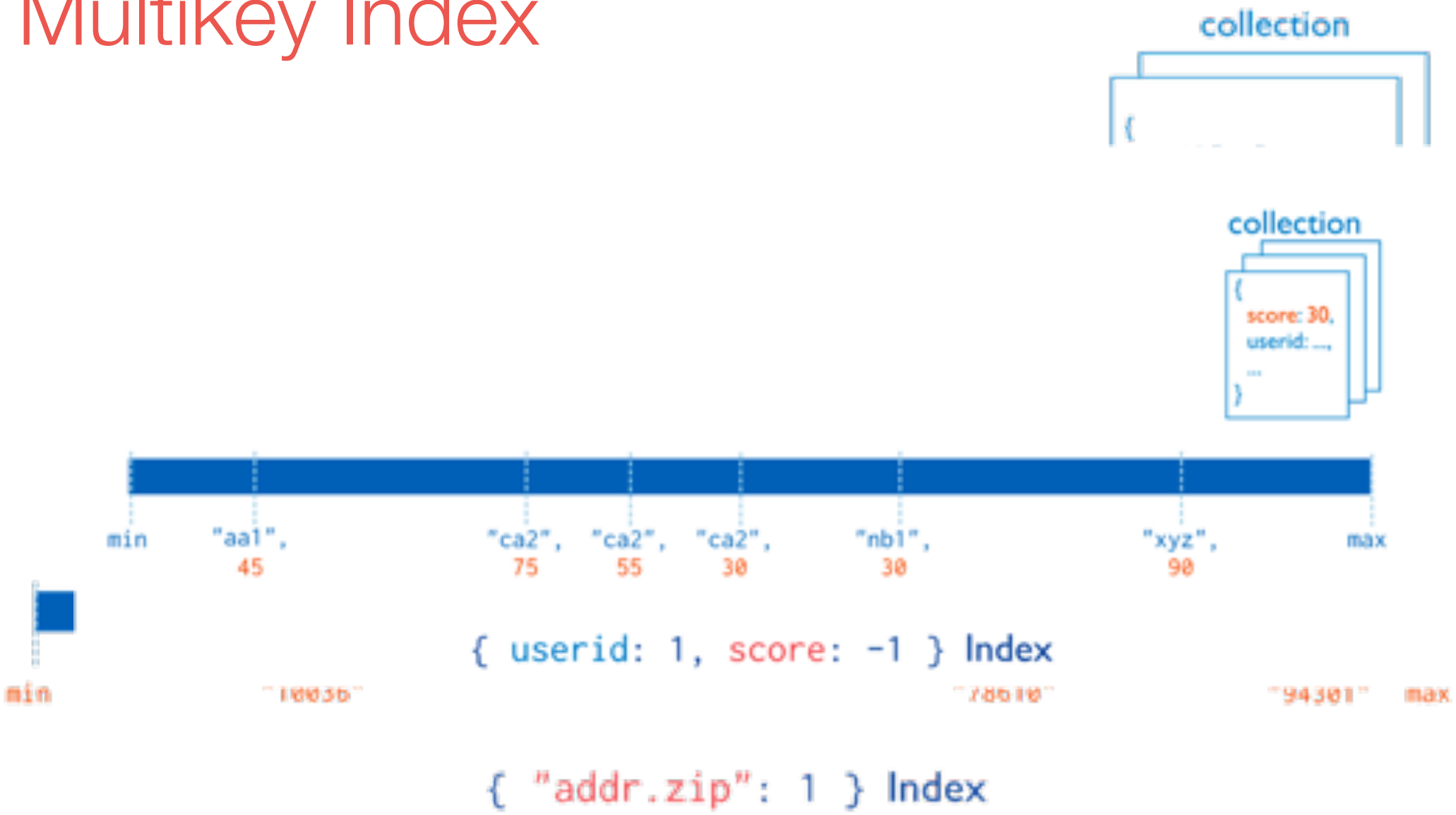
do not store language-specific stop words (e.g. “the”, “a”, “or”) and stem the words in a collection to only store root words.

Compound Index



{ userid: 1, score: -1 } Index

Multikey Index



Create an Index

```
db.collection.createIndex( <key and index type specification>,
<options> )
```

```
db.collection.createIndex( { name: -1 } )
```

Unique Index

on **groupNumber**, **lastname**, and **firstname** fields of the members collection

```
db.members.createIndex( { groupNumber: 1, lastname: 1,
firstname: 1 }, { unique: true } )
```

unique compound multikey index on **a.loc** and **a.qty**:

```
///
{ _id: 1, a: [ { loc: "A", qty: 5 }, { qty: 10 } ] }

db.collection.createIndex( { "a.loc": 1, "a.qty": 1 },
 { unique: true } )
```

Index Types

MongoDB cannot create a **unique index** on the specified index field(s) if the collection already contains data that would violate the unique constraint for the index.

If a document does not have a value for the indexed field in a unique index, the index will store a **null value** for this document. Because of the unique constraint, MongoDB **will only permit one document that lacks the indexed field.** If there is more than one document without a value for the indexed field or is missing the indexed field, the index build will fail with a duplicate key error.

Index Types

TTL Indexes special indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time

TTL Index

Index created on **createdAt** field where **expireAfterSeconds** Is the TTL time

```
db.log_events.createIndex( { "createdAt": 1 }, {
expireAfterSeconds: 3600 } )
```

mongod will automatically delete documents from the **log_events** collection when the document's **createdAt** value is older than the number of seconds specified in **expireAfterSeconds**.

```
db.log_events.insert( {
"createdAt": new Date(),
"logEvent": 2,
"logMessage": "Success!"
} )
```

TTL Index

Expire Documents at a Specific Clock Time

mongod will delete documents from the **log_events** collection when the documents' **expireAt** value is older than the number of seconds specified in **expireAfterSeconds**

```
db.log_events.createIndex( { "expireAt": 1 }, {
expireAfterSeconds: 0 } )
```

```
db.log_events.insert( {
"expireAt": new Date('July 22, 2013 14:00:00'),
"logEvent": 2,
"logMessage": "Success!"
} )
```

explain.md

WiXEngineering

Q&A

08 — Replication

Replication

A replica set in MongoDB is a group of mongod processes that maintain the same data set.

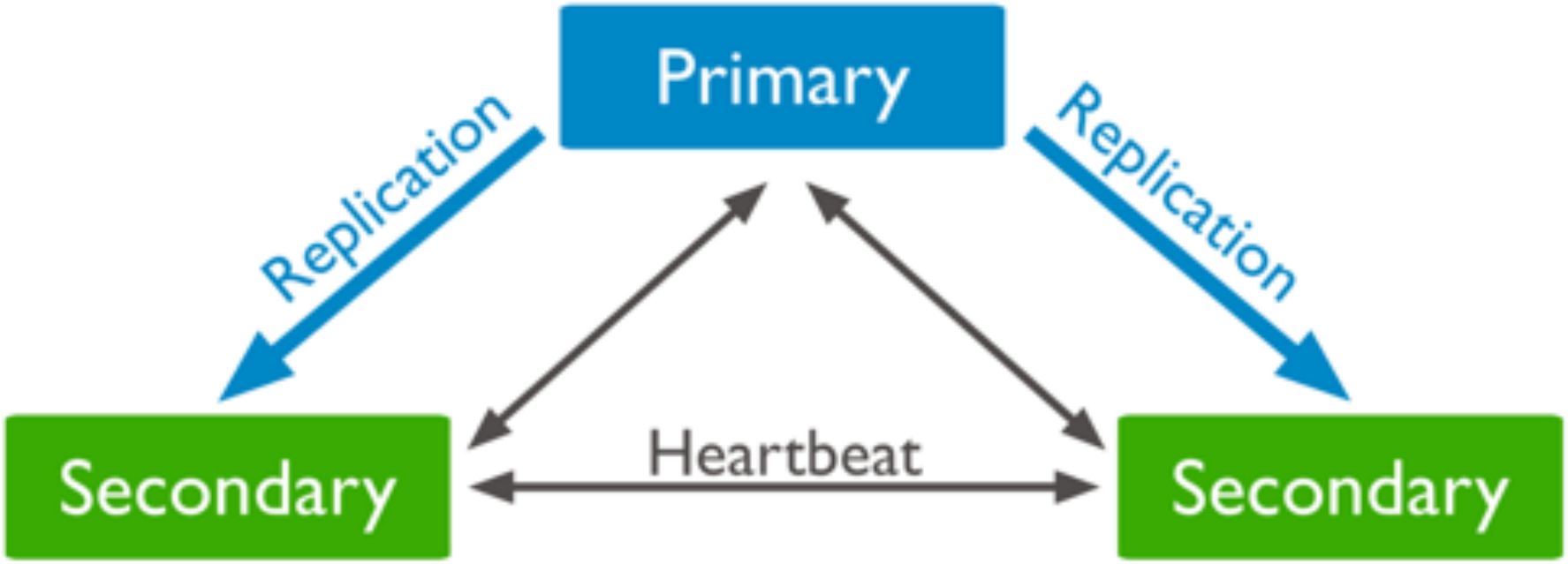
Replica sets provide redundancy and high availability,

basis for all production deployments.

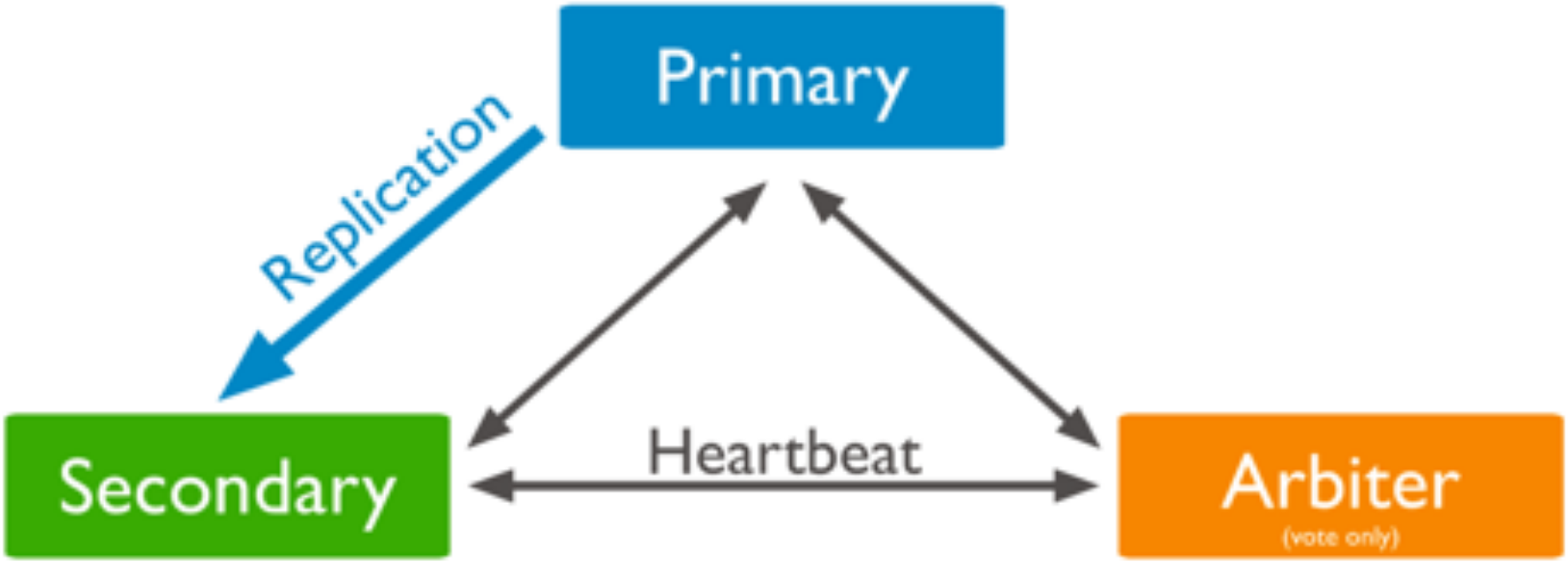
Replication provides redundancy and increases data availability and read throughput .

A replica set contains several data bearing nodes and optionally one arbiter node

Replication types



Replication types



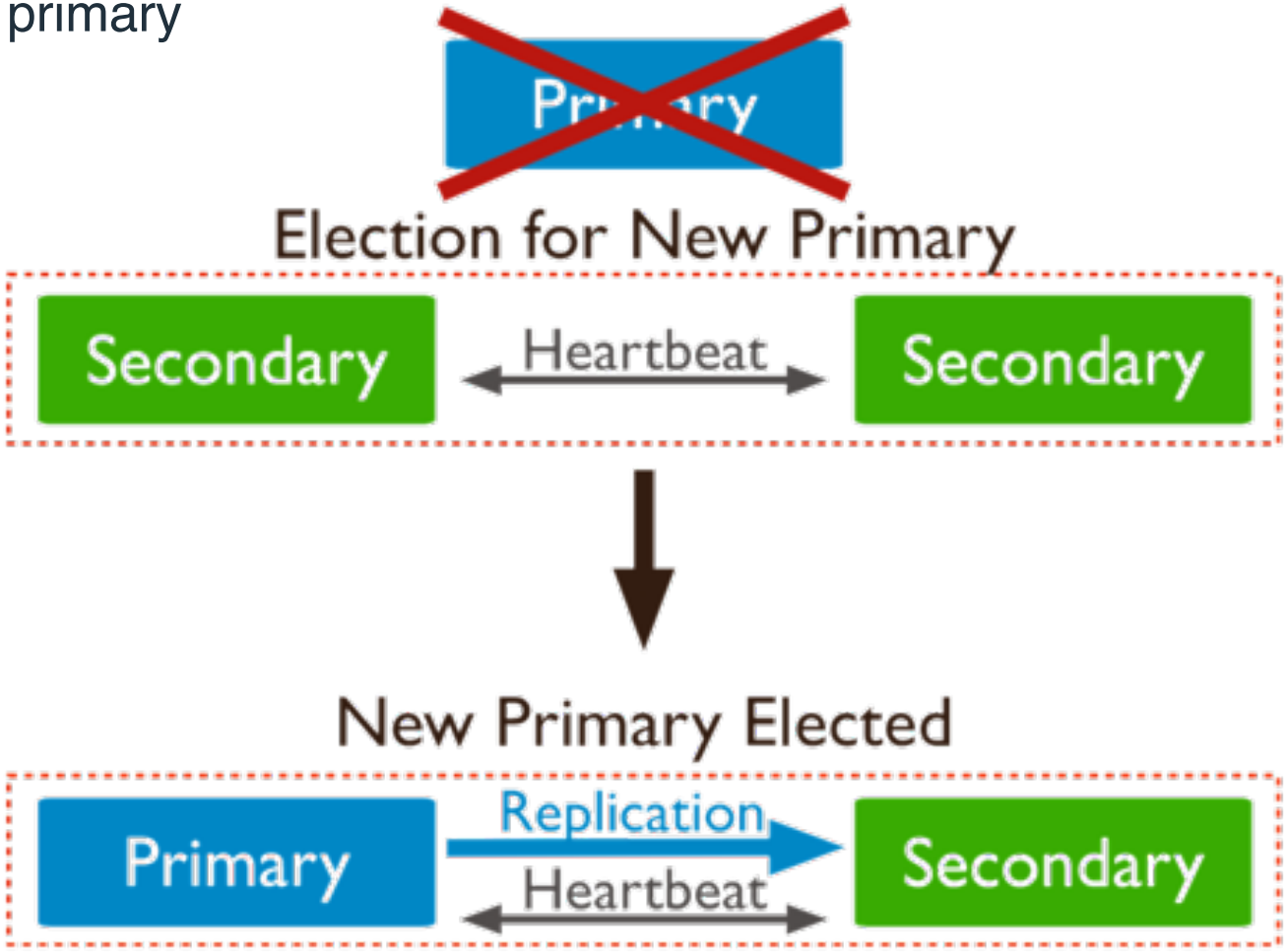
Replication

The primary node receives all write operations.

The secondaries replicate the primary's oplog and apply the operations to their data sets such that the secondaries' data sets reflect the primary's data set.

Replication fail over

If the primary is unavailable, an eligible secondary will hold an election to elect itself the new primary



Replication fail over

If the primary is unavailable, an eligible secondary will hold an election to elect itself the new primary

When a primary does not communicate with the other members of the set for more than the configured electionTimeoutMillis period (10 seconds by default) an eligible secondary calls for an election to nominate itself as the new primary.

The cluster attempts to complete the election of a new primary and resume normal operations.

The replica set cannot process write operations until the election completes successfully

The replica set can continue to serve read queries

Read Operations

By default, clients read from the primary.

clients can specify a read preference to send read operations to secondaries.

Asynchronous replication to secondaries means that reads from secondaries

may return data that does not reflect the state of the data on the primary

Read Preference Mode	Description
`primary`	Default mode. All operations read from the current replica set primary. Multi-document transactions that contain read operations must use read preference `primary`.
`secondaryPreferred`	In most situations, operations read from secondary members but if no secondary members are available, operations read from the primary.
`nearest`	Operations read from member of the replica set with the least network latency, irrespective of the member's type.

[1] *(1, 2)* In some circumstances, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with `{ w: "majority" }` write concern. The node that can complete `{ w: "majority" }` writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a network partition. When this occurs, clients that connect to the former primary may observe stale data despite having requested read preference `primary`, and new writes to the former primary will eventually roll back.

Read Operations

[1] *(1, 2)* In some circumstances, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with `{ w: "majority" }` write concern. The node that can complete `{ w: "majority" }` writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a network partition. When this occurs, clients that connect to the former primary may observe stale data despite having requested read preference `primary`, and new writes to the former primary will eventually roll back.

Run_replication.md

WiXEngineering

Q&A

09 — Sharding

Sharding

Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

Sharding

Vertical Scaling involves increasing the capacity of a single server, such as using a more powerful CPU, adding more RAM

Horizontal Scaling involves dividing the system dataset and load over multiple servers, adding additional servers to increase capacity as required. While the overall speed or capacity of a single machine may not be high, each machine handles a subset of the overall workload.

Sharded Cluster

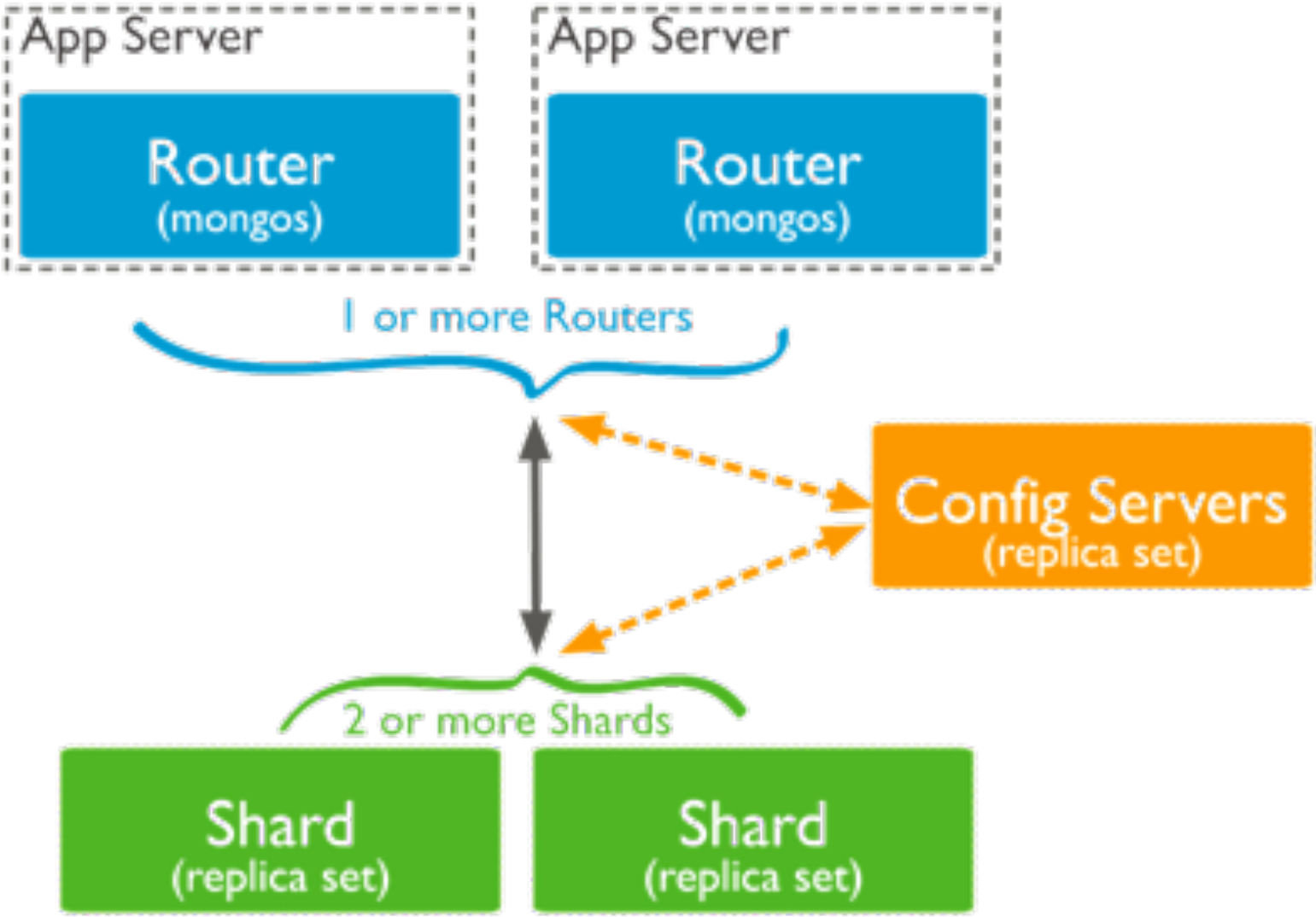
A MongoDB sharded cluster consists of the following components:

shard: Each shard contains a subset of the sharded data. Each shard can be deployed as a replica set.

mongos: The mongos acts as a query router, providing an interface between client applications and the sharded cluster.

config servers: Config servers store metadata and configuration settings for the cluster. As of MongoDB 3.4, config servers must be deployed as a replica set (CSRS).

Sharded Cluster



Sharding

Only shard the big collections

Pre-split if bulk loading

Pick shard key with care, they aren't easily changeable

Be cognizant of monotonically increasing shard keys

Adding a shard is easy but takes time

Use logical server names for config servers

Don't directly talk to anything except mongos except for dba work

Sharded Cluster Balancer

The MongoDB balancer is a background process that monitors the number of chunks on each shard.

When the number of chunks on a given shard reaches specific migration thresholds, the balancer attempts to automatically migrate chunks between shards and reach an equal number of chunks per shard.

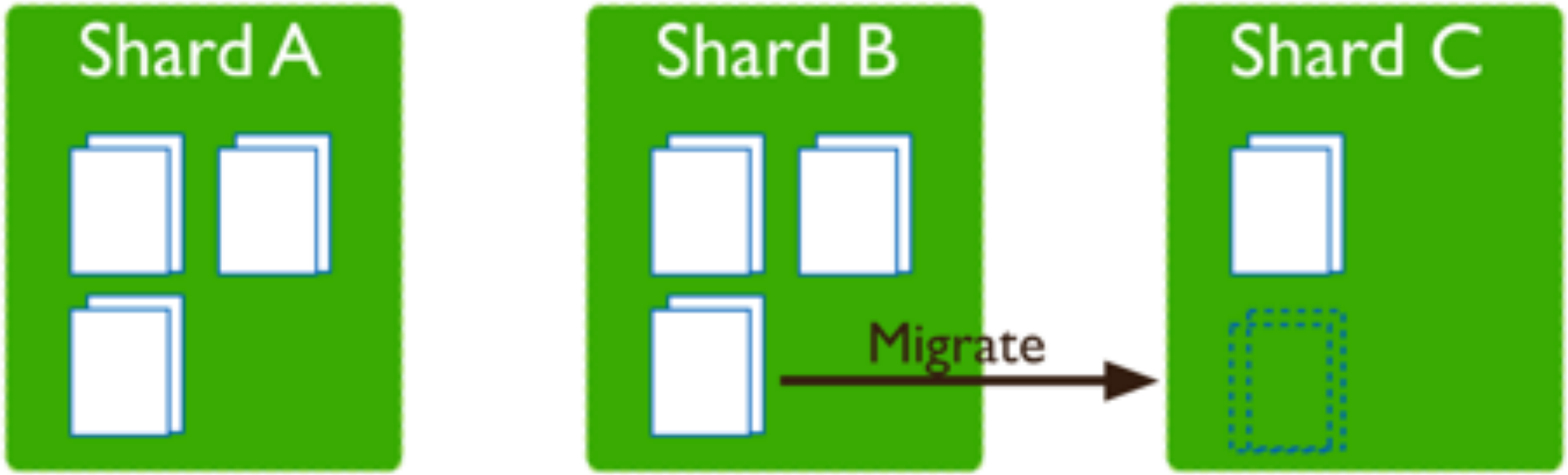
There may be some performance impact while the procedure takes place.

The balancer runs on the primary of the config server replica set (CSRS)

You may disable the balancer temporarily for maintenance

```
sh.disableBalancing("students.grades")
sh.isBalancerRunning()
```

Sharded Cluster Balancer



Sharded Cluster Balancer

Before starting a backup operation, confirm that the balancer is not active. You can use the following command to determine if the balancer is active:

```
!sh.getBalancerState() && !sh.isBalancerRunning()
```

run_cluster.md

WiXEngineering

Q&A

10 aggregation

Aggregation

Aggregation operations process data records and return computed results

Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result

MongoDB provides three ways to perform aggregation:

- aggregation pipeline
- map-reduce function
- single purpose aggregation methods.

Aggregation Pipeline

aggregation framework is modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result

- **First Stage**: The **$match** stage filters the documents by the status field and passes to the next stage those documents that have status equal to "A".

- **Second Stage**: The **$group** stage groups the documents by the **cust_id** field to calculate the sum of the amount for each unique **cust_id**.

```
db.orders.aggregate([
{ $match: { status: "A" } },
{ $group: { _id: "$cust_id", total: { $sum: "$amount" } } }
])
```

Map-Reduce

map-reduce operations have two phases:

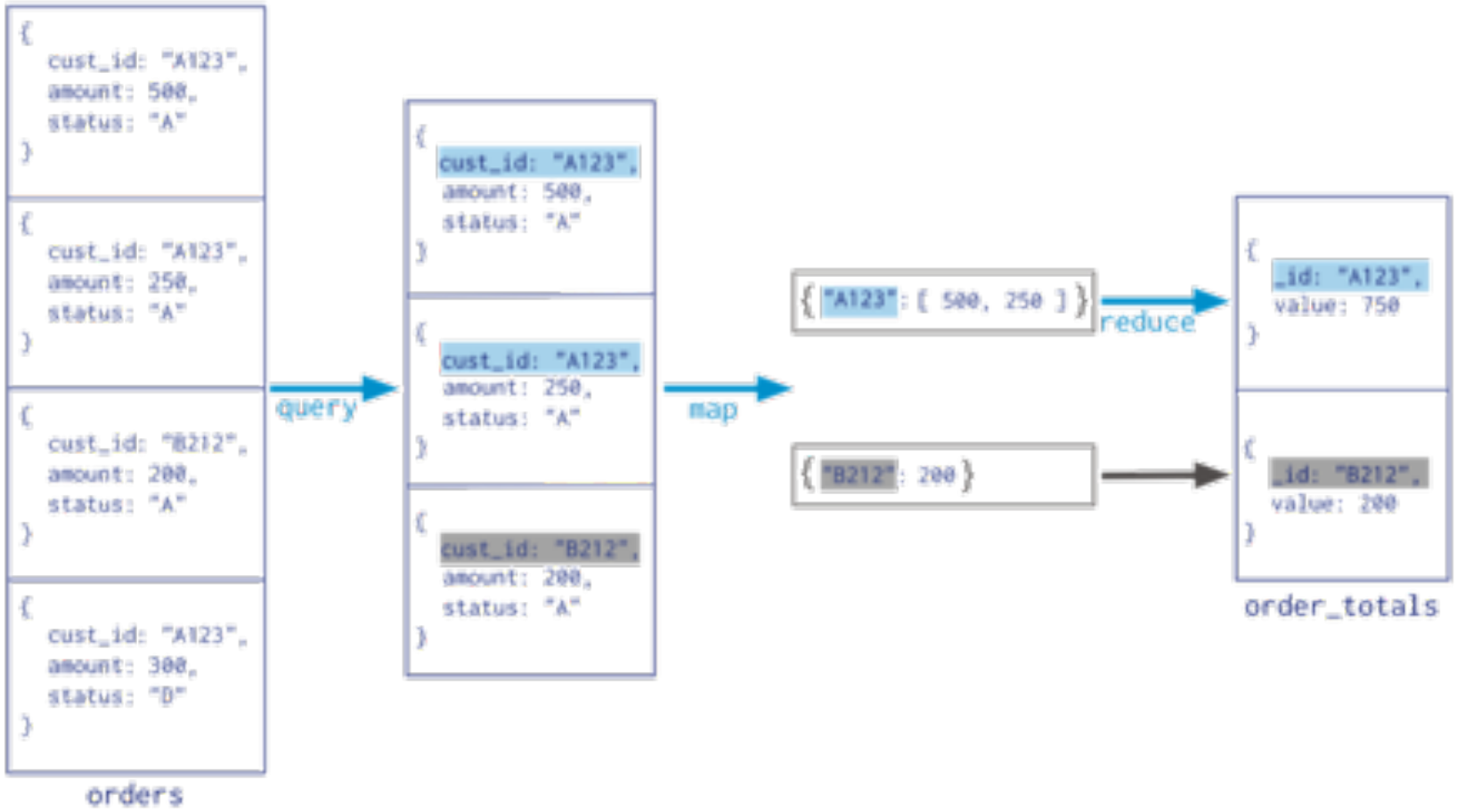
- map stage that processes each document and emits one or more objects for each input document
- reduce phase that combines the output of the map operation

Map-reduce uses custom JavaScript functions to perform the map and reduce operations

While the custom JavaScript provide great flexibility compared to the aggregation pipeline, in general, map-reduce is less efficient and more complex than the aggregation pipeline

```
db.orders.mapReduce(
        map      →  function() { emit( this.cust_id, this.amount ); },
        reduce   →  function(key, values) { return Array.sum( values ) },
                    {
        query    →    query: { status: "A" },
        output   →    out: "order_totals"
                    }
                )
```

Collection



Single Purpose Aggregation Operations

operations aggregate documents from a single collection.

provide simple access to common aggregation processes

lack the flexibility and capabilities of the aggregation pipeline and map-reduce.

db.collection.estimatedDocumentCount(), db.collection.count()

db.collection.distinct().

Collection

```
db.orders.distinct( "cust_id" )
```

```
{
  cust_id: "A123",
  amount: 500,
  status: "A"
}
```

```
{
  cust_id: "A123",
  amount: 250,
  status: "A"
}
```

```
{
  cust_id: "B212",
  amount: 200,
  status: "A"
}
```

```
{
  cust_id: "A123",
  amount: 300,
  status: "D"
}
```

orders

distinct →

```
[ "A123", "B212" ]
```

agreation.md

WiXEngineering

Q&A

11

MongoDB Change Streams

Real Time Data Streaming with MongoDB Change Streams

from version 3.6

allow applications to access real-time data changes without the complexity and risk of tailing the oplog

change streams can subscribe to

- all data changes on a single collection
- database
- entire deployment

Change stream is available for replica sets and sharded clusters

change_streams.md

WiXEngineering

Q&A

12 — Schema Design

Referencing vs Embedding

Procedure

```
{
    "_id" : 333,
    "date" : "2003-02-09T05:00:00"),
    "hospital" : "County Hills",
    "patient" : "John Doe",
    "physician" : "Stephen Smith",
    "type" : "Chest X-ray",
    "result" : 134
}
```

Results

```
{
    "_id" : 134
    "type" : "txt"
    "size" : Numbe
    "content" : {
        value1: 34
        value2: "a
        …
    }
}
```

Procedure

```
{
    "_id" : 333,
    "date" : "2003-02-09T05:00:00"),
    "hospital" : "County Hills",
    "patient" : "John Doe",
    "physician" : "Stephen Smith",
    "type" : "Chest X-ray",
    "result" : {
        "type" : "txt",
        "size" : NumberInt(12),
        "content" : {
            value1: 343,
            value2: "abc",
            …
        }
    }
}
```

Referencing vs Embedding

Referencing

Procedure

```
{
    "_id" : 333,
    "date" : "2003-02-09T05:00:00"),
    "hospital" : "County Hills",
    "patient" : "John Doe",
    "physician" : "Stephen Smith",
    "type" : "Chest X-ray",
    "result" : 134
}
```

Results

```
{
    "_id" : 134
    "type" : "txt",
    "size" : NumberInt(12),
    "content" : {
        value1: 343,
        value2: "abc",
        …
    }
}
```

Referencing vs Embedding

Referencing

- **Advantages**
  - Retrieve all relevant information in a single query/document
  - Avoid implementing joins in application code
  - Update related information as a single atomic operation
    - MongoDB doesn't offer multi-document transactions

- **Limitations**
  - Large documents mean more overhead if most fields are not relevant
  - Might mean replicating data
  - 16 MB document size limit

Referencing vs Embedding

Embedding

Procedure

```
{
    "_id" : 333,
    "date" : "2003-02-09T05:00:00"),
    "hospital" : "County Hills",
    "patient" : "John Doe",
    "physician" : "Stephen Smith",
    "type" : "Chest X-ray",
    "result" : {
        "type" : "txt",
        "size" : NumberInt(12),
        "content" : {
            value1: 343,
            value2: "abc",
            …
        }
    }
}
```

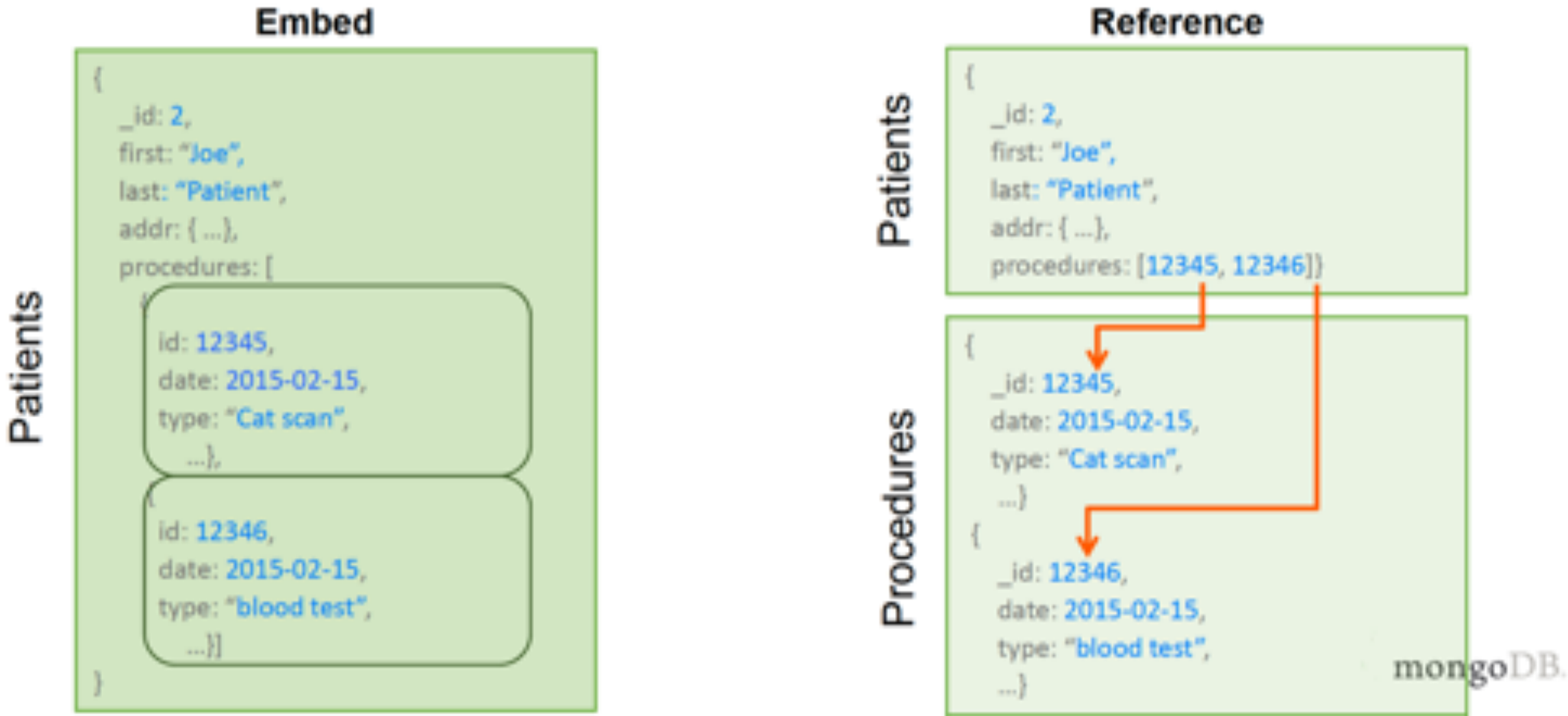
Referencing vs Embedding

Embedding

- **Advantages**
  - Smaller documents
  - Less likely to reach 16 MB document limit
  - Infrequently accessed information not accessed on every query
  - No duplication of data

- **Limitations**
  - Two queries required to retrieve information
  - Cannot update related information atomically

Referencing vs Embedding



Schema Design

Keep documents small MongoDB works best if you keep individual documents to a few kilobytes in size. (try not to use)

Try not to use GridFS.

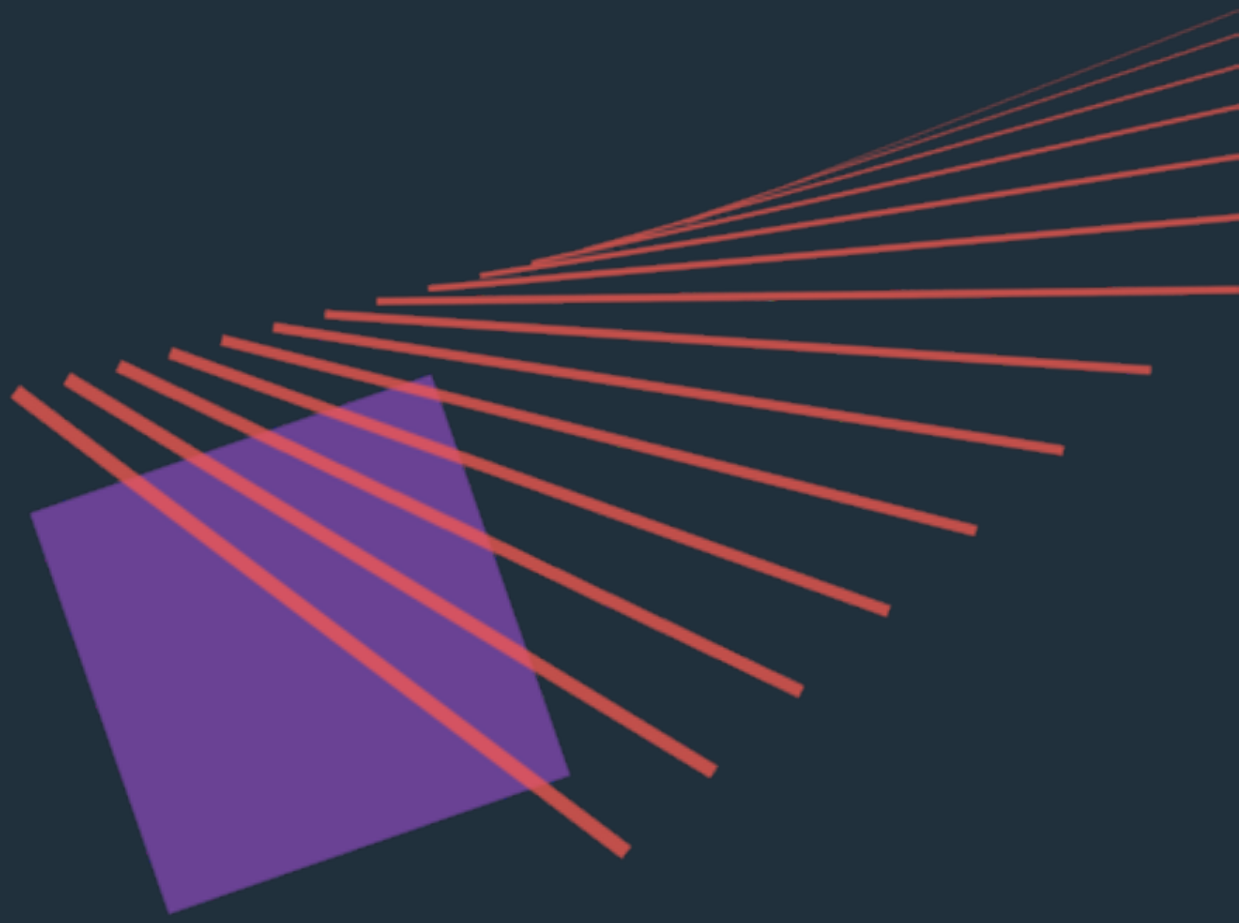
Keep the number of array elements well below four figures (If the array is added to frequently → outgrow the containing document so that its location on disk has to be moved → means every index must be updated.)

Try to use Index that can cover sort.

If MongoDB cannot use an index to get documents in the requested sort order, the combined size of all documents in the sort operation, plus a small overhead, must be less than 32 megabytes.

WiXEngineering

Q&A

WiXEngineering

Thank You