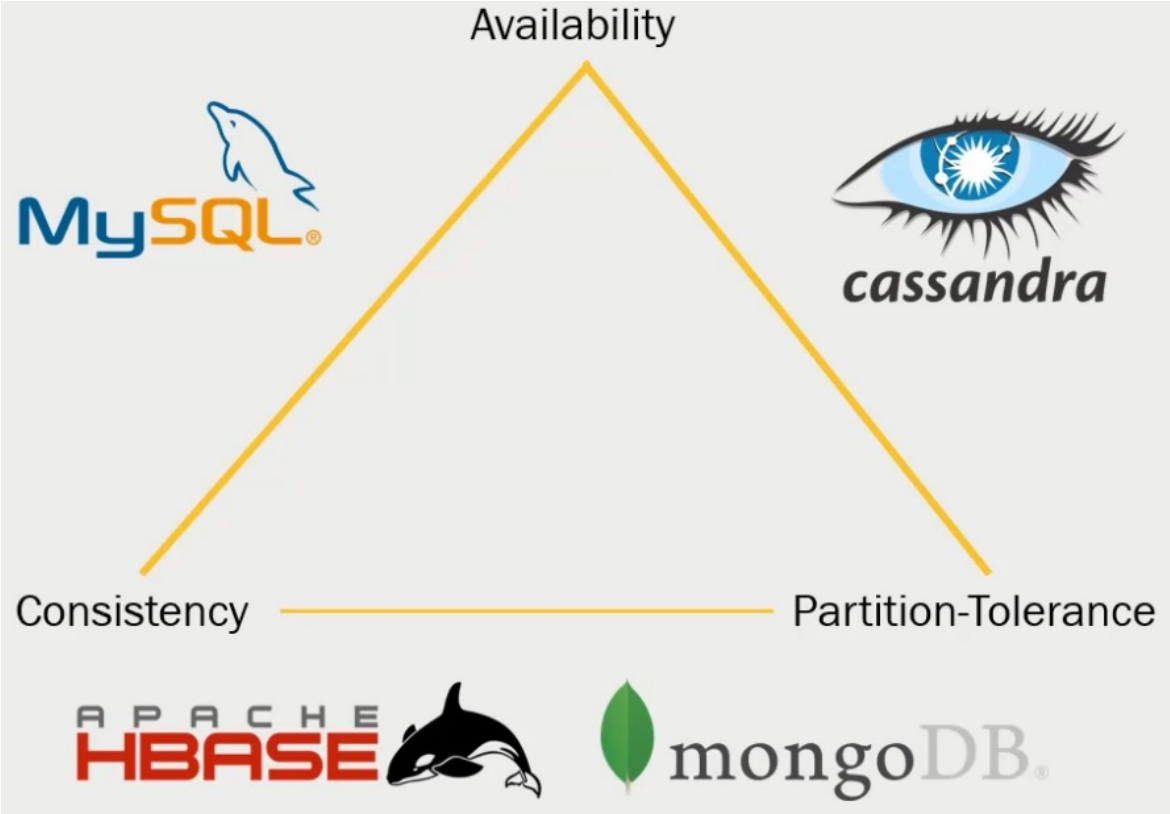
# MongoDB

Managing HuMONGOus data

Popular choice in the corporate world

# mongoDB in CAP theorem




- mongoDB has a single primary node
- We need *"to talk to"* the primary node all the time to ensure consistency
- A period of unavailability occurred if the master node goes down

  **-> need to wait for a new primary node to put in place**

- **we can still read from the Mongo Database**
  **-> writes will be disabled until the issue is resolved**

# Document-based data model

Looks like JSON. Example:

```
{
        "_id" : ObjectID("7b33e366ae32223aee34fd3"),
        "title" : "A blog post about MongoDB",
        "content" : "This is a blog post about MongoDB",
        "comments": [
                {
                        "name" : "Frank",
                        "email" : fkane@sundog-soft.com,
                        "content" : "This is the best article ever written!"
                        "rating" : 1
                }
        ]
}
```

Generated automatically

**A blog post document**

- Doesn't have to be structured
- Don't need to have same schema across each document
- Can put whatever we want
- Automatically append ***"_id"*** to the documents
  - as ***unique identifier***

# No real schema is enforced

- We *can* enforce a schema, but it's not required
- We can have different fields in every documents if you want to
- No single "key" as in other databases such as Cassandra or HBase
  - But we can create indices on any field we want, or even combinations of fields
  - If we want to "shard", then we must do so on some indexed field

- **will automatically create an "_id" field to act as primary key**

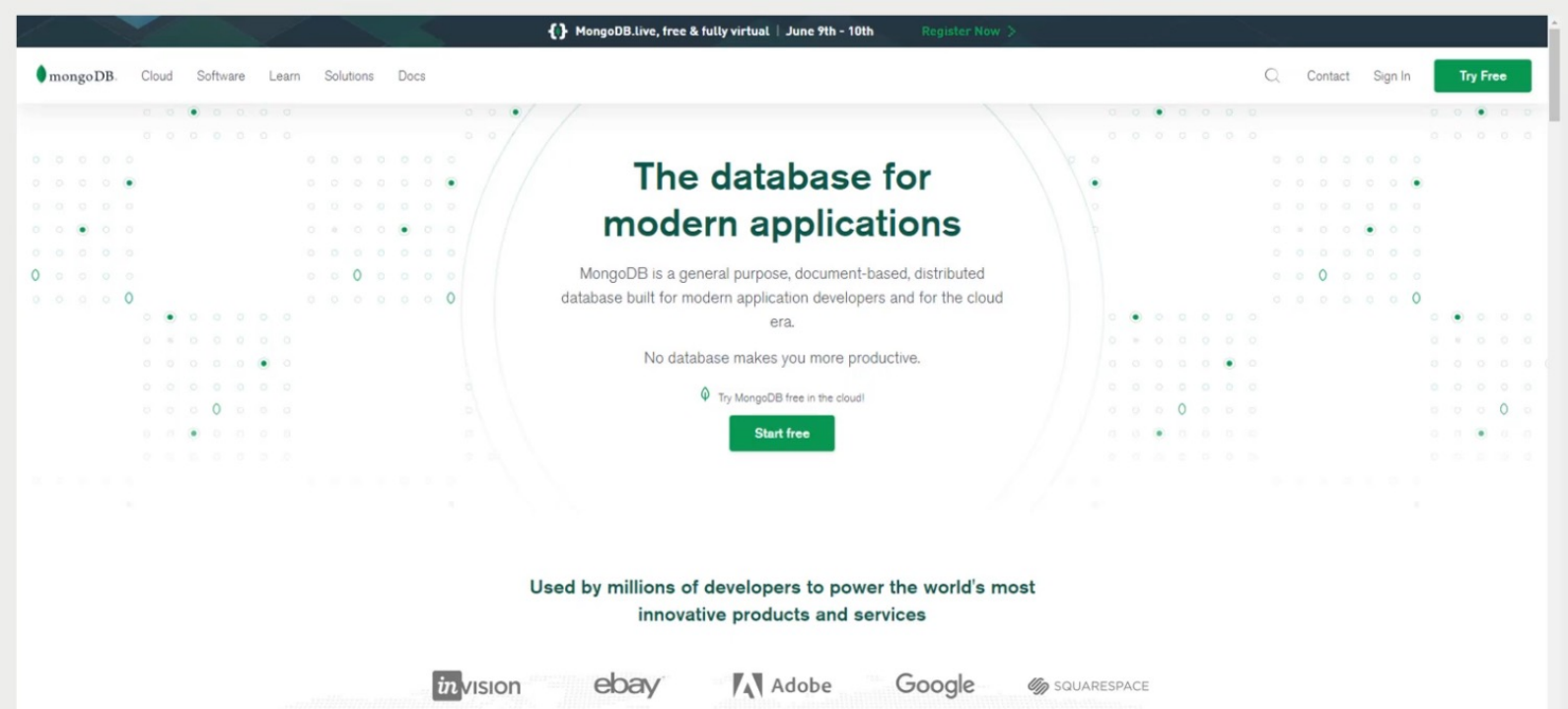
# MongoDB terminology

- Databases
- Collections – can contain pretty much anything; cannot move data between collections across different databases [collection of documents]

  [need to be within the same database]
- Documents

Collections that contain documents
vs
tables that contain rows

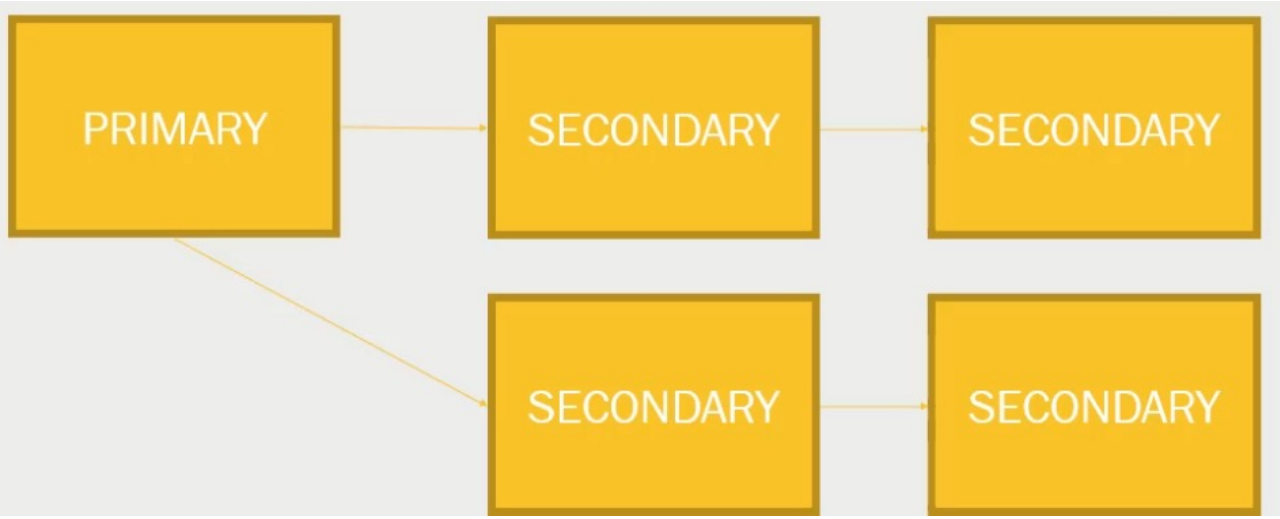
# Aimed at enterprises




- Enterprise-based solution primarily
- Offered professional supports

# MongoDB architecture

- Single-master → **[focus on consistency over availability]**
- Replication sets – select secondary node which has the lowest ping time in the event of primary node goes down
- Maintains backup copies of our database instance
  - Secondaries can elect a new primary within seconds if the primary goes down



# Replication set

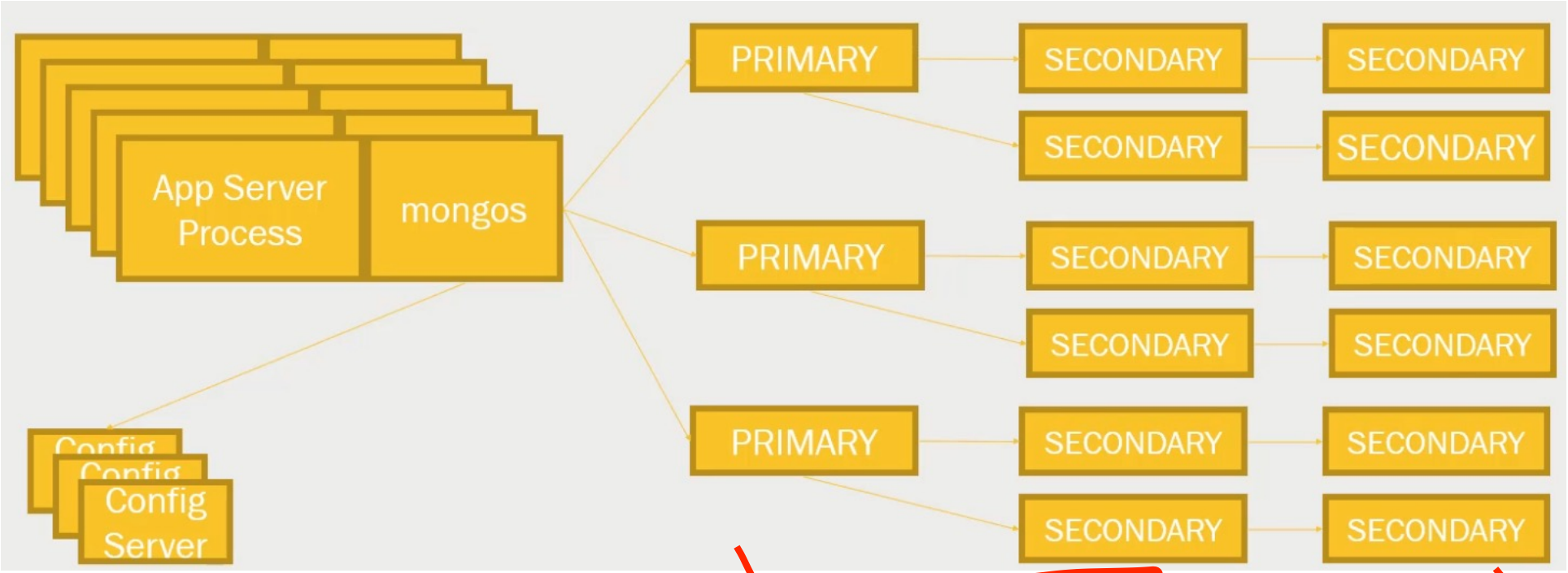
data redundancy;
high availability (automatic failover)

- Replicas only address durability, not the ability to scale
- Delayed secondaries can be set up as insurance against accidental mishaps [e.g. an hour delay for secondary node]

• need to agree on who is the next primary node (majority wins)
-> Even numbers of servers (like 2) don't work well

# Sharding – scaling up for Big Data

- Ranges of some indexed value we specify are assigned to different replica sets
- Index is used to balance the load of information among multiple replica sets **[balancer]**




RS1: users
Min -> 1000

RS2: users
1001 -> 5000

RS3: users
5001 -> max

**can change over time and get rebalanced over time**

# Neat things about MongoDB

- Very flexible documents model
- Built-in aggregation  capabilities, MapReduce
  - For some application we might not need Hadoop at all
  - But MongoDB still integrates with Hadoop, Spark, and most language
- A SQL connector is available
  - But mongoDB still isn’t designed for joins and normalized data
  - **still can’t deal with normalized data efficiently**

# Install MongoDB and Integrate MongoDB with Spark

Bernard Lee Kok Bang

# Install mongoDB

- Login to puTTY

  *su root*

  *cd /var/lib/ambari-server/resources/stacks/*

  *cd HDP*

  *cd 2.6 #change accordingly to the version of HDP you installed*

  *cd services*

  *git clone https://github.com/nikunjness/mongo-ambari.git*

  *sudo service ambari-server restart*

# Login to Ambari

- *127.0.0.1:8080*
- sign in as *admin* user
- Choose *Actions* > *Add Service* > choose *MongoDB 3.2* > click *Next*
- Accept all *default parameters* > click *Next* > *Proceed Anyway* > *Deploy*
- in putty: *pip install pymongo==3.4.0*

# Running MongoDB + Spark script

- Back to ~~home directory~~ ***[own directory in puTTY]***
  - *cd ~*
  - *vi MongoSpark.py*

```
from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.sql import functions

def parseInput(line):
    fields = line.split('|')
    return Row(user_id = int(fields[0]), age = int(fields[1]), gender = fields[2], occupation = fields[3], zip = fields[4])
```

```python
if __name__ == "__main__":
    # Create a SparkSession
    spark = SparkSession.builder.appName("MongoDBIntegration").getOrCreate()

    # Get the raw data
    lines = spark.sparkContext.textFile("hdfs:///user/maria_dev/ml-100k/u.user")
    # Convert it to a RDD of Row objects with (userID, age, gender, occupation, zip)
    users = lines.map(parseInput)
    # Convert that to a DataFrame
    usersDataset = spark.createDataFrame(users)

    # Write it into MongoDB
    usersDataset.write\
        .format("com.mongodb.spark.sql.DefaultSource")\
        .option("uri","mongodb://127.0.0.1/movielens.users")\
        .mode('append')\
        .save()

    # Read it back from MongoDB into a new Dataframe
    readUsers = spark.read\
    .format("com.mongodb.spark.sql.DefaultSource")\
    .option("uri","mongodb://127.0.0.1/movielens.users")\
    .load()

    readUsers.createOrReplaceTempView("users")

    sqlDF = spark.sql("SELECT * FROM users WHERE age < 20")
    sqlDF.show()

    # Stop the session
    spark.stop()
```

***change accordingly***

- *spark-submit --packages org.mongodb.spark:mongo-spark-connector_2.11:2.3.2 MongoSpark.py*

# Output

"_id" field is automatically added by MongoDB; unique identifier for each roow

| _id | age | gender | occupation | user_id | zip |
|---|---|---|---|---|---|
| [588a1c2046e0fb17... | 18 | F | student | 482 | 40256 |
| [588a1c2046e0fb17... | 18 | F | writer | 507 | 28450 |
| [588a1c2046e0fb17... | 19 | M | student | 521 | 02146 |
| [588a1c2046e0fb17... | 18 | M | student | 528 | 55104 |
| [588a1c2046e0fb17... | 19 | F | student | 541 | 84302 |
| [588a1c2046e0fb17... | 16 | F | student | 550 | 95453 |
| [588a1c2046e0fb17... | 16 | M | student | 580 | 17961 |
| [588a1c2046e0fb17... | 17 | M | student | 582 | 93003 |
| [588a1c2046e0fb17... | 18 | F | student | 588 | 93063 |
| [588a1c2046e0fb17... | 18 | M | student | 592 | 97520 |
| [588a1c2046e0fb17... | 19 | F | artist | 601 | 99687 |
| [588a1c2046e0fb17... | 13 | F | student | 609 | 55106 |
| [588a1c2046e0fb17... | 15 | F | student | 618 | 44212 |
| [588a1c2046e0fb17... | 17 | M | student | 619 | 44134 |
| [588a1c2046e0fb17... | 18 | F | writer | 620 | 81648 |
| [588a1c2046e0fb17... | 17 | M | student | 621 | 60402 |
| [588a1c2046e0fb17... | 19 | M | student | 624 | 30067 |
| [588a1c2046e0fb17... | 13 | M | none | 628 | 94306 |
| [588a1c2046e0fb17... | 18 | F | student | 631 | 38866 |
| [588a1c2046e0fb17... | 18 | M | student | 632 | 55454 |

# Using the MongoDB shell from puTTY

- *mongo*
- *use movielens*

*# Retrieve any collections in mongoDB that match the expression*

- *db.users.find( {user_id: 100} )*

```
> db.users.find( {user_id: 100 } )
{ "_id" : ObjectId("588a1c2046e0fb17ab62db7d"), "age" : NumberLong(36), "gender"
 : "M", "occupation" : "executive", "user_id" : NumberLong(100), "zip" : "90254"
 }
```

# Indexing mongoDB

- Without indexing, mongoDB would do a full table scan to retrieve the information we want [takes time if the data is BIG!!!]
- ***db.users.explain().find( {user_id:100} )***


```
                        "user_id" : {
                                "$eq" : 100
                        }
                },
                "winningPlan" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "user_id" : {
                                        "$eq" : 100
                                }
                        },
                        "direction" : "forward"
                },
                "rejectedPlans" : [ ]
        },
        "serverInfo" : {
                "host" : "sandbox.hortonworks.com",
                "port" : 27017,
                "version" : "3.2.11",
                "gitVersion" : "009580ad490190ba33d1c6253ebd8d91808923e4"
        },
        "ok" : 1
}
```

# Indexing mongoDB (cont…)

***"-1" means in descending order***

- *db.users.createIndex( {user_id: 1} )* #"1" means in ascending order
- *db.users.explain().find( {user_id: 100} )*
- *db.users.find( {user_id: 100} )*

```
},
"winningPlan" : {
        "stage" : "FETCH",
        "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                        "user_id" : 1
                },
                "indexName" : "user_id_1",
                "isMultiKey" : false,
                "isUnique" : false,
                "isSparse" : false,
                "isPartial" : false,
                "indexVersion" : 1,
                "direction" : "forward",
                "indexBounds" : {
                        "user_id" : [
                                "[100.0, 100.0]"
```

# MongoDB aggregation function [more complex application]

- Aggregate all users by occupation and find their average age
- Code starts with *"$" sign* have special meaning / function in mongoDB
- *db.users.aggregate( [ {$group: { _id: {occupation: "$occupation"}, avgAge: { $avg: "$age"} } } ] )*

```
... { $group: { _id: { occupation: "$occupation"}, avgAge: { $avg: "$age" } } }
... ])
{ "_id" : { "occupation" : "none" }, "avgAge" : 26.555555555555557 }
{ "_id" : { "occupation" : "entertainment" }, "avgAge" : 29.22222222222222 }
{ "_id" : { "occupation" : "salesman" }, "avgAge" : 35.666666666666664 }
{ "_id" : { "occupation" : "healthcare" }, "avgAge" : 41.5625 }
{ "_id" : { "occupation" : "librarian" }, "avgAge" : 40 }
{ "_id" : { "occupation" : "marketing" }, "avgAge" : 37.61538461538461 }
{ "_id" : { "occupation" : "writer" }, "avgAge" : 36.31111111111111 }
{ "_id" : { "occupation" : "homemaker" }, "avgAge" : 32.57142857142857 }
{ "_id" : { "occupation" : "administrator" }, "avgAge" : 38.74683544303797 }
{ "_id" : { "occupation" : "student" }, "avgAge" : 22.081632653061224 }
{ "_id" : { "occupation" : "executive" }, "avgAge" : 38.71875 }
{ "_id" : { "occupation" : "programmer" }, "avgAge" : 33.121212121212125 }
{ "_id" : { "occupation" : "educator" }, "avgAge" : 42.01052631578948 }
{ "_id" : { "occupation" : "other" }, "avgAge" : 34.523809523809526 }
{ "_id" : { "occupation" : "engineer" }, "avgAge" : 36.38805970149254 }
{ "_id" : { "occupation" : "lawyer" }, "avgAge" : 36.75 }
{ "_id" : { "occupation" : "technician" }, "avgAge" : 33.148148148148145 }
{ "_id" : { "occupation" : "retired" }, "avgAge" : 63.07142857142857 }
{ "_id" : { "occupation" : "doctor" }, "avgAge" : 43.57142857142857 }
{ "_id" : { "occupation" : "scientist" }, "avgAge" : 35.54838709677419 }
```

# Other functions

- *db.users.count()* # Count number of rows **[943]**
- *db.getCollectionInfos()* # get lists of collections available
- *db.users.drop()* **# get rid of the collection**
- *db.getCollectionInfos()*

# Shutting Down MongoDB PROPERLY!!!

- 1st: *exit* # exit *mongoDB shell*
- 2nd: *Stop mongoDB* # in Ambari
- 3rd: exit # exit from *shell terminal*
- 4th: shut down HDP virtual machine: *Machine > ACPI shutdown*

**Word of cautions:**

Remember to shut down MongoDB correctly; might not start up again!

Need to delete HDP virtual image and reinstalled from ground up!!

set up root user and admin account from scratch!!!

# Choosing the right database

Bernard Lee Kok Bang

**MySQL, PostgreSQL, MongoDB, Cassandra, HBase**

# Integration considerations

- What systems do we need to integrate together?
- Different technologies have different connectors
- If we have a big analytics job that is currently running in Apache Spark, what external databases should we choose, that can connect easily to Apache Spark?
- Maybe we have a (i) front-end system that depends on SQL interface to a back-end database; (ii) thinking to migrate from monolithic relational database to a distributed non-relational systems, it would be easier if the non-relational database offers SQL-like interface that can be easily migrated from the front-end application

# Scaling Requirements

- How much data are we really talking about?
- Is it going to grow unbounded over time?
- May need some sort of a database technology that is not limited to the data that we can store only on one PC
- May need MongoDB, HBase or Cassandra where we can distribute the storage of the data across an entire cluster and scale horizontally
- Also think about transaction rates: How many requests do we intend to get per second?
- May need distributed database so that we can spread out the load of those transaction more evenly
- →big website where a lot of web servers serving a lot of people at the same time

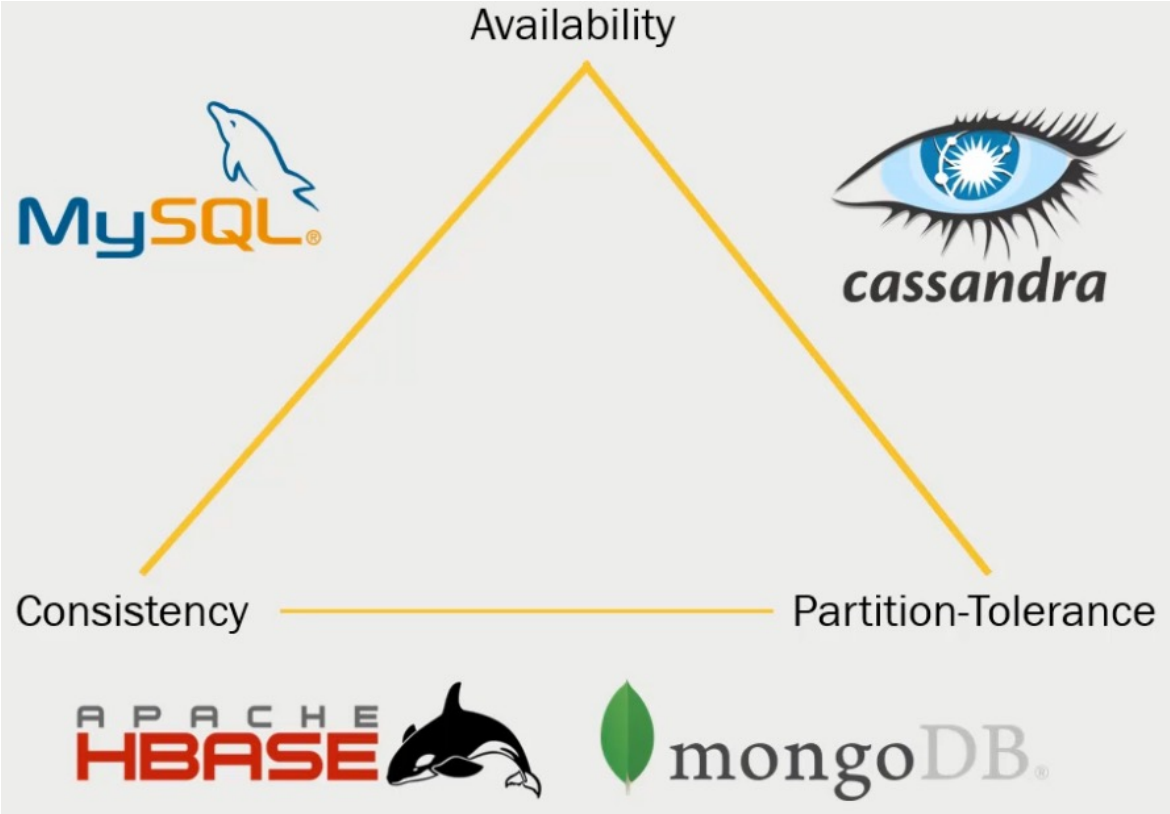
# Support considerations

- Do we have in-house expertise to spin up the new database technology? **[and configure properly]**
- Do we have the right expertise who can handle customer sensitive information such personal identification information?
- Need to think deeply about the security of our system
- If the No-SQL database were configured using default settings, there will be *NO SECURITY* at all!!!
- Whether the database system offer *professional paid support*?
- Whether we can outsource the database administration?
- Corporate solution such as MongoDB might be a good choice

# Budget considerations

- Probably minimal, as most of these databases are open-source
- Need (i) cost for support; (ii) cost for setting up the servers
- Can rent server such as Amazon EC2 servers, Amazon Web Service, Google Cloud platform, etc…

# CAP considerations




- Partition-tolerance: Scale requirement
- Availability: is it OK if your system goes down for a few seconds or a few minutes?
- Consistency: dealing with real transactional information such as stock transaction, value consistency above all else..

# Keep it simple

- If we don’t need to set up complex NoSQL cluster and database that needs a lot of maintenance, don’t do it!!!
- Think about the minimum requirements that we need for our system
- If don’t need to deal with massive scale, don’t deploy a NoSQL database
- Simplicity is the guiding principle that our architecture decisions should be based on

# Case study 1:

- We are building an internal phone directory **app**
  - -Scale: limited
  - -Consistency: eventual is fine
  - -Availability requirements: Not mission critical

# Case study 2

- We want to mine web server logs for interesting patterns
- What are the most popular times of the day?
- What’s the average session length?
- **All we wanted to do is analytics -> Hadoop , Spark, Hive, Pig…**
- **Do I have enough scale to warrant non-relational database?**

# Case study 3

- We have a big Spark job that produces movie recommendations for end users nightly
- Something needs to vend these data to our web applications
- We work for some huge company with massive scale
- Downtime is not tolerated [availability]
- Must be fast
- Eventual consistency OK – it’s just reads

# Case study 4

- You are building a massive stock trading system
- Consistency is more important than anything else
- “Big Data” is present **[we care about partition-tolerance]**
- It’s really, really important – so having access to professional support might be a good idea. And you have enough budget to pay for it
- **you need a lot of security requirements**