6. Introduction to NoSQL Databases

Semana 1: Introducing NoSQL

Overview of NoSQL

- new ways of storing and querying data

- big data
- typically address more specialized use cases
- simpler to develop

Characteristics of NoSQL Databases

Types of nosql databases:

- Key-Value
- Document
- Column
- Graph

Similar characteristics:

- opensource
  - roots in the open source community
  - used and leveraged in an open source manner
  - open source support community
- built to scale horizontally
- share data more easily
- use a global unique key to simplify data sharding or partinioning
- more use case specific than RDBMS
- more developer friendly than RDBMS
- more agile development via flexible schemas

Benefits:

- scalability
- performance: fast responce, high concurrency
- availability
- cloud architecture
- lower cost
- flexible schema

- varied data structures
- specialized capabilities
  - indexing and querying
  - data replication robustness
  - modern http APIs

NoSQL Database Categories

Key-Value based

- the least complex architectural speaking
- represented as hashmap
- ideal for basic CRUD operations (Create, read, update and delete)
- scale well
- shard easily
- not intended for complex queries
- atomic for single key operations only
- value blobs are opaque to database
- less flexibility for indexing and querying data

Suitable Use cases:

- quick basic CRUD operations on non-interconnected data
  - e.g. Storing and retrieving session information for web applications
- storing in-app user profiles and preferences
- shopping cart data for online stores

Unsuitable Use cases:

- for data that is interconnected with many-to-many relationships
  - social networks
  - recommendation engines

- when high-level consistency is required for multi-operation transactions with multiple keys
    - need a database that provides ACID transactions
- when apps runs queries based on value vs key
    - consider using 'Document' category of NoSQL database

Key-Value NoSQL database examples:

- amazon dynamoDB
- oracle nosql database
- redis
- riak
- memcached
- project voldemort

Document based

- values are visible and can be queried
- each piece of data is considered a document
    - typically json or xml
- each document offers a flexible schema
    - no two documents need to contain the same information
- index and query
    - key and value range lookups and search
    - analytical queries with MapReduce
- horizontally scalabe
- allow sharding across multiple nodes
- typically only garantee atomic operations on single documents

Suitable use cases:

- event logging for apps and processes — each event instance is represented by a new document
- online blogs — each user, post, comment, like, or action is represented by a document
- operational datasets and metadata for web and mobile apps — designed with Internet in mind (json, RESTful APIs, unstructured data)

Unsuitable use cases:

- when you require ACID transactions
  - document databases can't handle transactions that operate over multiple documents
  - relational database would be a better choice
- if your data is in an aggregate-oriented design
  - if data naturally falls into a normalized tabular model
  - relational database would be a better choice

examples:

- ibm cloudant
- mongoDB
- CouchDB
- couchbase
- terrastore
- orientDB

Column based

- spawned from google's BigTable
- bigtable clones or columnar or wide-column databases
- store data in columns or groups of columns

- column families are several rows, with unique keys, belonging to one or more columns
    - grouped in families as often accessed together
- rows in a column family are not required to share the same columns

Suitable use cases:

- great for large amounts of sparse data
- column databases can handle being deployed across clusters fo nodes
- can be used for event logging and blogs
- counters are a unique use case for column databases
- can have a TTL parameter (time to live), making them useful for data with an expiration value

Unsuitable use cases:

- for traditional ACID transactions
    - reads and writes are only atomic at the row level
- in early development, query patterns may change and require numerous changes to column-based databases

examples:

- cassandra
- apache hbase
- hypertable
- accumulo

Graph based

- store information in entities (or nodes) and relationships (or edges)
- impressive when your data set resembles a graph-like data structure

- do not shard well
    - traversing a graph with nodes split across multiple servers can become difficult and hurt performance
- ACID transaction compliant

Suitable use cases:

- for highly connected and related data
- social networking
- routing, spatial and map apps
- recommendation engines

Unsuitable use cases:

- when looking for advantages offered by other nosql DB categories
- when an application needs to scale horizontally
    - you will quickly reach the limitations associated with these types of data stores
- when trying to update all or a subset of nodes with a given parameter
    - there types of operations can prove to be difficylt and non-trivial

examples:

- neo4j
- orientdb
- arangodb
- amazon webservices
- apache giraph
- janus graph

Database Deployment Options

The major consideration when choosing the best database for your applications and organization is around where to host it and how it is being managed. It is important to understand all of the components to make sure you end up with the simplest and most cost-effective option.

In a traditional **in-house do-it-yourself scenario**, you will own the setup of the underlying hardware and operating system, installation and configuration of the chosen database management system, overall administration including patching and support, and of course how the application data is designed.

This contrasts a little bit with **hosted database solutions**. A hosted database solution means the provider is choosing what hardware your database runs on, and they are provisioning it for you. But it is your responsibility to provide and install the software, perform the administrative tasks, and do the design. So, at the end of the day, the hosting provider are handing the administrative keys over to you and it’s really up to your team of database administrators to keep things scaling and running smoothly.

In comparison, using a **fully managed database-as-a-service (or DBaaS)** is really meant to eliminate the complexity and risk of doing it all in-house, and help development teams get to market faster, scale more smoothly and massively, and provide better performance and availability for end users.

The only concern for users of a fully managed DBaaS is the design and development of their product. Guaranteed uptime, availability, and scalability are all the result of a fully managed DBaaS.

You mitigate risk by offloading database administration and data layer management issues from your development team. And you ensure that your developers need only concern themselves with what really matters – developing better applications for your customers.

Summary and Highlights

- NoSQL means Not only SQL.
- NoSQL databases have their roots in the open source community.
- NoSQL database implementations are technically different from each other.
- There are several benefits of adopting NoSQL databases including storing and retrieving session information, and event logging for apps.

- The four main categories of NoSQL database are Key-Value, Document, Wide Column, and Graph.
- Key-Value NoSQL databases are the least complex architecturally.
- Document-based NoSQL databases use documents to make values visible for queries.
- In document-based NoSQL databases, each piece of data is considered a document, which is typically stored in either JSON or XML format.
- Column-based databases spawned from the architecture of Google’s Bigtable storage system.
- The primary use cases for column-based NoSQL databases are event logging and blogs, counters, and data with expiration values.
- Graph databases store information in entities (or nodes) and relationships (or edges).

ACID vs BASE

- relational and nosql databases have different consistency models. The consistency models that exist are ACID and BASE
- relational db → acid
- nosql db → base

ACID

- **Atomic**: every operation in a transaction succeeds or everything is rolled back
- **Consistent**: on the completion of the transaction, the structural integrity of the data in the database is not compromised
- **Isolated**: transactions cannot compromise the intefrity of other transactions by interacting with them while they are still in progress
- **Durable**: the data related to the completed transaction will persist even in the case of network or power outages

- financial institutions will almos exclusively use acid for money transfers

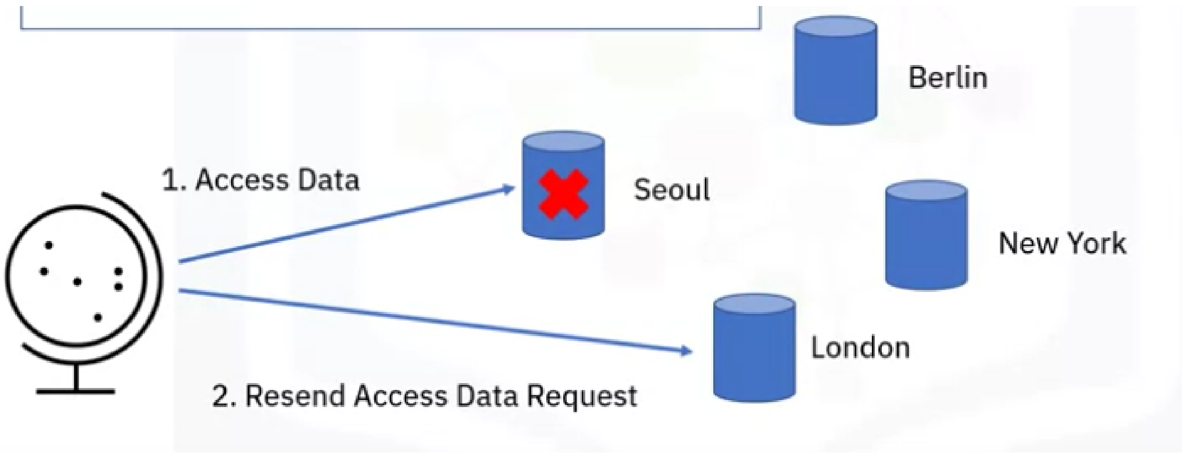
BASE

- **basically availabe:** Rather than enforcing immediate consistency, BASE-modelled NoSQL databases will ensure availability of data by spreading and replicating it across the nodes of the database cluster.
- **soft based:** due to the lack of immediate consistency, data values may change over time
- **eventually consistent:** The fact that the BASE model does not enforce immediate consistency does not mean that it never achieves it. However, until it does, data reads might be inconsistent

- nosql — few requirements ofr immediate consistency, data freshness and accuracy
- nosql benefits — availability, scale and resilience
- used by:
  - marketing and customer service companies
  - social media apps
  - worldwide available online services
- favor availability over consistency of data
- NoSQL databases use the BASE consistency model

use cases:

- worldwide online services — users' access to services
  - netflix, apple, spotify, uber

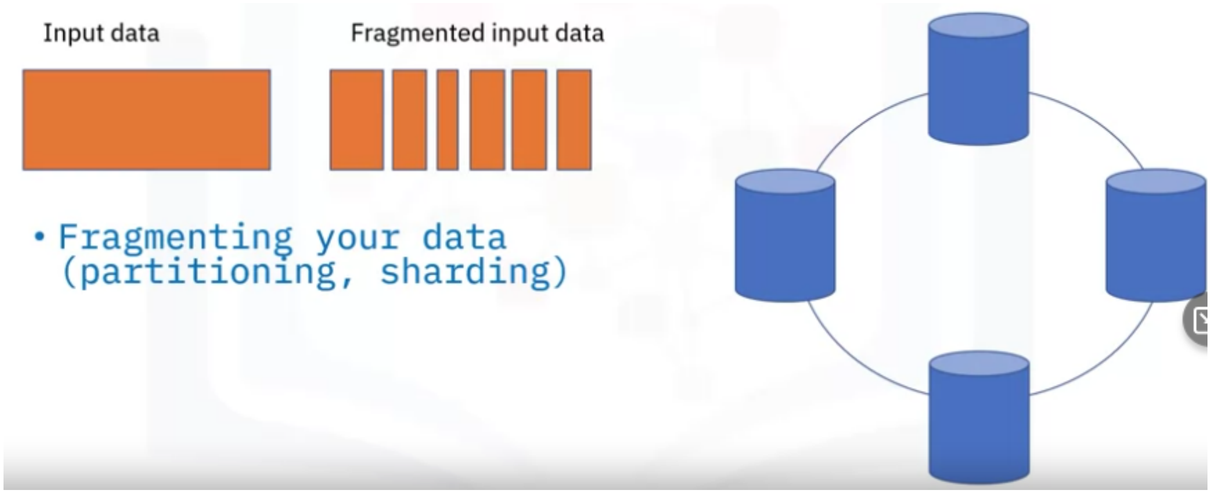


Distributed Databases

- A distributed database is a collection of multiple interconnected databases
- spread physically across various locations
- fragmentation and replication
- BASE consistency model

<u>**Fragmenting your data**</u> (partitioning, sharding)

- grouping keys lexically (for example, all records with A or A-C)
- grouping records by key (for example, all records with key StoreId = 123)



- Fragmenting your data (partitioning, sharding)

<u>**Replication**</u>

- protection of data for node failures
- replication: all data fragments are stored redundantly in two or more sites
- increases availability of the data
- replicated data needs to be syncronized ⇒ could lead to inconsistencies

<u>**Availability**</u>

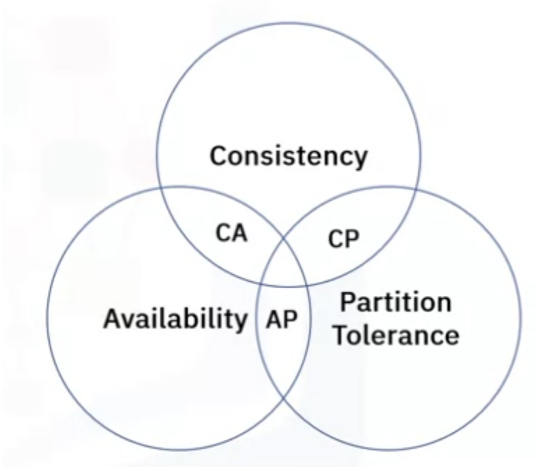
- reliability and availability
- improved perfomance

- query processing time reduced
- ease of growth / scale
- continuous availability

Architecture challenges

- concurrency control: consistency of data
    - write/reads to a single node per fragment of data ⇒ data is synchronized in the background
    - write operations go to all nodes holding a fragment of data, reads to a subset of nodes per consistency
    - developer-driven consistency of data
- no transaction support (or limited version)

The CAP Theorem (Brewer's theorem)



Three essential system requirements necessary for the successful design, implementation, and deployment of applications in distributed systems:

- CONSISTENCY
- AVAILABILITY
- PARTITION TOLERANCE

Consistency

- it refers to whether a system operates fully or not
- do all nodes within a cluster see all the data they are supposed to?

Availability

- does each request get a response outside of failure or success?

Partition Tolerance

- represents the fact that a given system continues to operate even under circumstances of data loss or network failure
- **partition**: communications break whithin a distributed system — a lost or temporarily delayed connection between nodes
- **partition tolerance:** the cluster must continue to work despite any number of communication breakdowns between nodes in the system
- in distributed systems partitions CANNOT be avoided
- it is a basic feature of NoSQL

NoSQL: CP or AP

- NoSQL — a choice between consistency and availability
- mongoDB → first ensures consistency
- apache cassandra → first ensures availability

Challenges in Migrating from RDBMS to NoSQL Databases

RDBMS:

- consistency
- structured data (fixed schema)
- transactions
- complicated joins

NoSQL:

- high performance
- unstructured data
- availability
- easy scalability

Data driven model to Query driven data model

- RDBMS: starts from the data integrity, relations between entities
- noSQL: starts from your queries, not from your data. Models based on the way the application interacts with the data

Normalized to denormalized data

- noSQL: think how data can be structured based on your queries
- RDBMS: start from your normalized data and then build the queries

From ACID to BASE model

- availability VS consistency
- cap theorem — choose between consistency and availability
- availability, performance, geographical presence, high data volumes

Transactions

- noSQL: by design, do not support transactions and joins (except in limited cases)

Summary and Highlights

- ACID stands for Atomicity, Consistency, Isolated, Durable.
- BASE stands for Basic Availability, Soft-state, Eventual Consistency.
- ACID and BASE are the consistency models used in relational and NoSQL databases.
- Distributed databases are physically distributed across data sites by fragmenting and replicating the data.
- Fragmentation enables an organization to store a large piece of data across all the servers of a distributed system by breaking the data into smaller pieces.
- You can use the CAP Theorem to classify NoSQL databases.
- Partition Tolerance is a basic feature of NoSQL databases.
- NoSQL systems are not a de facto replacement of RDBMS.
- RDBMS and NoSQL cater to different use cases, which means that your solution could use both RDBMS and NoSQL.

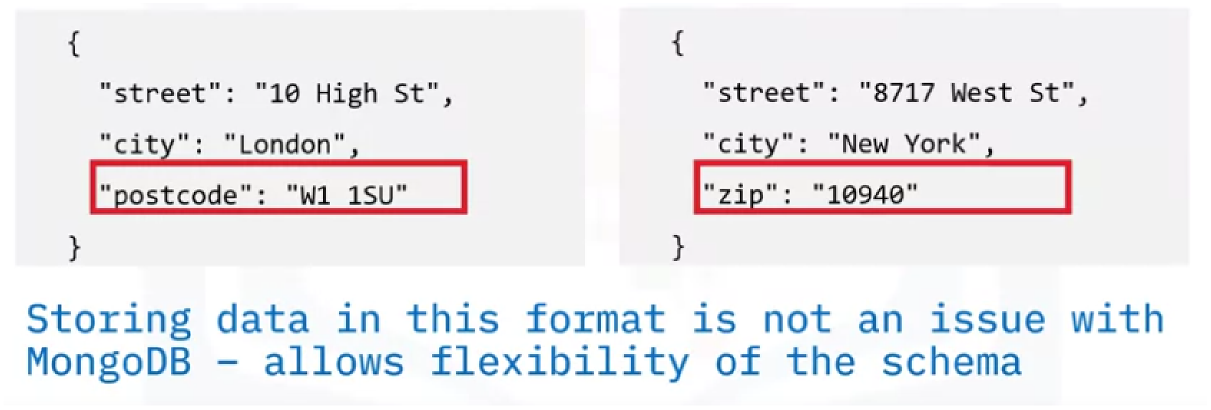
Semana 2: Introducing MongoDB

Overview on MongoDB

- Document and NoSQL database
- each record is a document
- structured in a non-relational database
- a document is an associative array like JSON objects or Python dictionaries
- documents of same type are stored as a collection
- the database stores collections
    - students and employees collections stored in a database called CampusManagementDB
- supports sub-documents (json con sub jsons)
- Why use mongodb?
    - model data as you read/write, not the other way
        - traditional relational databases: create schema first, then create tables
            - to store another field, you have to alter tables
        - in mongodb you change as you go along
        - bring structured and unstructured data
        - high availability
    - large and unstructured
    - complex
    - flexible
    - high scalable applications
    - self-managed, hybrid, or cloud hosted
- https://www.ibm.com/cloud/databases-for-mongodb

Advantages of MongoDB

Flexibility of schema



```
{
  "street": "10 High St",
  "city": "London",
  "postcode": "W1 1SU"
}
```

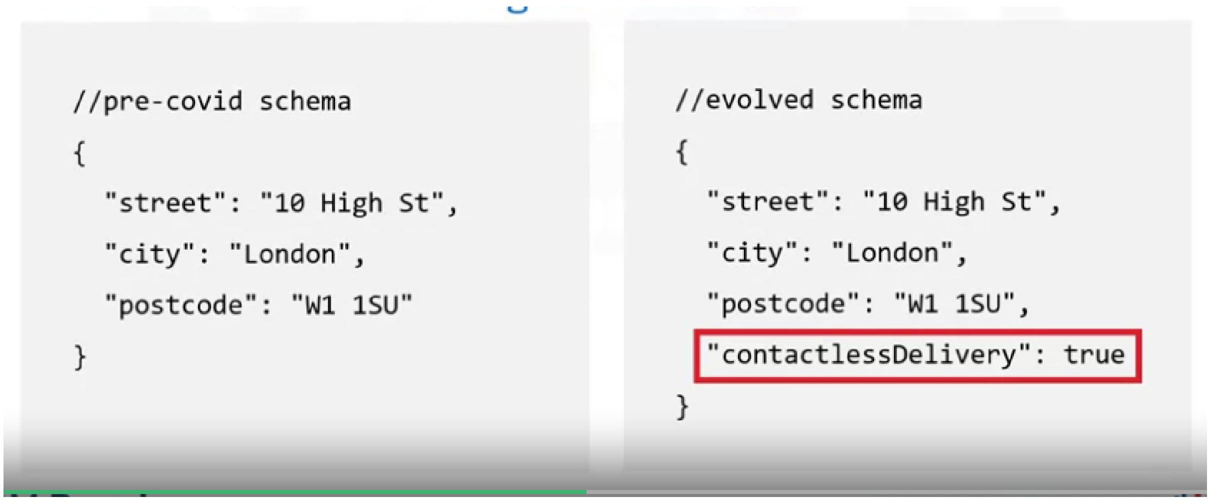
```
{
  "street": "8717 West St",
  "city": "New York",
  "zip": "10940"
}
```

Storing data in this format is not an issue with MongoDB – allows flexibility of the schema

Code-first approach

- no complex table definitions
- write as sonn as you connect

Evolving schema



```
//pre-covid schema
{
  "street": "10 High St",
  "city": "London",
  "postcode": "W1 1SU"
}
```

```
//evolved schema
{
  "street": "10 High St",
  "city": "London",
  "postcode": "W1 1SU",
  "contactlessDelivery": true
}
```

Querying and analytics

- mongoDB querying language is called MQL
- wide range of operators
- aggregation pipelines (group all students by their year group and find top scoring students by semester)

High availability

- resilience through redundancy
- no system maintenance downtime

- no upgrade downtime

Use Cases for MongoDB

- many sources — one view
  - no more data in silos
  - easy data ingestion
  - flexible schema
- can be used with IOT devices
  - vast amount of data
  - scale
  - expressive querying
- e-commerce
  - products with different attributes
  - optimise for read
  - dynamic schema
- real-time analytics
  - quick response to changes
  - simplified ETL
  - real time, along with operational data
- gaming
  - globally scalable
  - no downtime
  - supporting rapid development
- finance
  - speed of operations
  - security: encrypred info
  - reliability

LAB: mongoDB from command line

```
start_mongo

#Your mongodb server is now ready to use and available with username: root password: M
jE2NzUtbWFqb2Nh

#You can access your mongodb database via:
# • The browser at: https://majocarbajal-8081.theiadocker-5-labs-prod-theiak8s-3-tor0
1.proxy.cognitiveclass.ai
# • CommandLine: mongo -u root -p MjE2NzUtbWFqb2Nh --authenticationDatabase admin loca
l

mongo -u root -p MjE2NzUtbWFqb2Nh --authenticationDatabase admin local

db.version()

show dbs

use training #creates and switches to database training. If it exists only switches

db.createCollection("mycollection") #create collection inside training

show collections

db.mycollection.insert({"color":"white","example":"milk"}) #insert a document

db.mycollection.insert({"color":"blue","example":"sky"}) #another document

db.mycollection.count() #number of documents in the collection

db.mycollection.find() #list all documents in the collection mycollection. Notice that
mongodb automatically adds an '_id' field to every document in order to uniquely ident
ify the document.

exit
```

```
use mydatabase

db.createCollection("landmarks")

db.landmarks.insert({"name":"Statue of Liberty","city":"New York","country":"USA"})
db.landmarks.insert({"name":"Big Ben","city":"London","country":"UK"})
db.landmarks.insert({"name":"Taj Mahal","city":"Agra","country":"India"})
db.landmarks.insert({"name":"Pyramids","country":"Egypt"})
db.landmarks.insert({"name":"Great Wall of China","country":"China"})
```

Summary and Highlights

- MongoDB is a document and NoSQL database.

- It is easy to access by indexing.
- It supports various data types, including dates and numbers.
- The database schema can be flexible when working with MongoDB.
- You can change the database schema as needed without involving complex data definition language statements.
- Complex data analysis can be done on the server using Aggregation Pipelines.
- The scalability MongoDB provides makes it easier to work across the globe.
- MongoDB enables you to perform real-time analysis on your data.

CRUD Operations

```
#Create
db.students.insertOne(JSON)

#Read
db.students.findOne() #returns the first one in natural document reading order
db.students.findOne(with certain criteria)
students.find({"lastName": "Doe"}) #find all that match
students.count({"lastName": "Doe"}) #how many have this lastname

#Replace
#sending the whole document back with the new changes
  student = db.students.findOne({"lastName": "Doe"})
  db.student["onlineOnly"] = true
  db.student["email"] = "asjdkb"
  db.students.replaceOne({"lastName": "Doe"}, student)
#for larger documents we send small changes
  changes = {"$set": { "onlineOnly": true, "email" : "askjdb" } }
  db.students.updateOne({"lastName": "Doe"}, changes)
  db.students.updateMany({}, {"$set": { "onlineOnly": true}})

#Delete
db.students.deleteOne({"studentId" : 2231})
db.students.deleteMany({"graduatedYear" : 2019})
```

if an ID is not provided, mongoDB defines one automaticallly. Every document has an ID

Indexes

- help quickly locate data without looking for it everywhere

```
db.courseEnrollment.find({"courseId": 1547})

db.courseEnrollment.createIndex({"studentId" : 1})

#sorting
db.courseEnrollment.find({"courseId": 1547}).sort({"studentId" : 1}) #in memory sort,
 not efficient

db.courseEnrollment.createIndex({"courseId": 1, "studentId" : 1} #compound index, bett
er
```

- indexes in mongoDB are special data structures
- they store the fields you are indexing
- they also store the location of the document on disk
- stored in tree form

LAB

```
use training

db.createCollection("bigdata")

#The code given below will insert 200000 documents into the 'bigdata' collection.
#Each document would have a field named account_no which is a simple auto increment nu
mber.
#And a field named balance which is a randomly generated number, to simulate the bank
 balance for the account.
use training
for (i=1;i<=200000;i++){print(i);db.bigdata.insert({"account_no":i,"balance":Math.roun
d(Math.random()*1000000)})}

db.bigdata.count()

#Let us run a query and find out how much time it takes to complete.
db.bigdata.find({"account_no":58982}).explain("executionStats").executionStats.executi
onTimeMillis #returns 97 millis

#Before you create an index, choose the field you wish to create an index on. It is us
ually the field that you query most.
db.bigdata.createIndex({"account_no":1})

db.bigdata.getIndexes()
#[
#        {
#                "v" : 2,
#                "key" : {
#                        "_id" : 1
#                },
#                "name" : "_id_",
#                "ns" : "training.bigdata"
#        },
#        {
#                "v" : 2,
#                "key" : {
#                        "account_no" : 1
#               },
#                "name" : "account_no_1",
#                "ns" : "training.bigdata"
#        }
#]

db.bigdata.find({"account_no": 69271}).explain("executionStats").executionStats.execut
ionTimeMillis #now it returns 1 milli, because we created an index

#delete the index
db.bigdata.dropIndex({"account_no":1})
```

Aggregation Framework/ Pipeline

- Process ... Stages ... Outcome
- is a series of operations that you apply on your data to get a desired outcome.

```
db.courseResults.aggregate([
{ $match: {"year": 2020} },
{ $group: { "_id": "$courseId", "avgScore" : { $avg: $score} } }
])
```

Common aggregation stages

- $project
    - if you want to change the shape of documents or project out certain fields
- $sort
    - to sort your documents
- $count
    - to count and assign the outcome to a new field
- $merge
    - takes the outcome from the previous stage and stores it into a target collection

Use cases

- reporting
- analysis

LAB: aggregation pipelines

```
use training

db.marks.insert({"name":"Ramesh","subject":"maths","marks":87})
db.marks.insert({"name":"Ramesh","subject":"english","marks":59})
db.marks.insert({"name":"Ramesh","subject":"science","marks":77})
db.marks.insert({"name":"Rav","subject":"maths","marks":62})
db.marks.insert({"name":"Rav","subject":"english","marks":83})
db.marks.insert({"name":"Rav","subject":"science","marks":71})
db.marks.insert({"name":"Alison","subject":"maths","marks":84})
db.marks.insert({"name":"Alison","subject":"english","marks":82})
db.marks.insert({"name":"Alison","subject":"science","marks":86})
db.marks.insert({"name":"Steve","subject":"maths","marks":81})
db.marks.insert({"name":"Steve","subject":"english","marks":89})
db.marks.insert({"name":"Steve","subject":"science","marks":77})
db.marks.insert({"name":"Jan","subject":"english","marks":0,"reason":"absent"})

use training
#Using the $limit operator we can limit the number of documents printed in the output.
This command will print only 2 documents from the marks collection.
db.marks.aggregate([{"$limit":2}])

#We can use the $sort operator to sort the output.
db.marks.aggregate([{"$sort":{"marks":1}}])
db.marks.aggregate([{"$sort":{"marks":-1}}]) #descending order

#Aggregation usually involves using more than one operator. A pipeline consists of one
or more operators declared inside an array. The operators are comma separated. Mongodb
executes the first operator in the pipeline and sends its output to the next operator.
#Let us create a two stage pipeline that answers the question "What are the top 2 mark
s?".
db.marks.aggregate([
{"$sort":{"marks":-1}},
{"$limit":2}
])

#The operator $group by, along with operators like $sum, $avg, $min, $max, allows us t
o perform grouping operations.
#This aggregation pipeline prints the average marks across all subjects.
#EQUIVALENT TO:
#SELECT subject, average(marks) FROM marks GROUP BY subject
db.marks.aggregate([
{
    "$group":{
        "_id":"$subject",
        "average":{"$avg":"$marks"}
        }
}
])

#finding the average marks per student.
#sorting the output based on average marks in descending order.
#limiting the output to two documents.
db.marks.aggregate([
{
    "$group":{
        "_id":"$name",
        "average":{"$avg":"$marks"}
        }
},
{
    "$sort":{"average":-1}
},
{
    "$limit":2
}
])
```

Replication & Sharding

- A typical MongoDB cluster is made of three data bearing nodes. All three nodes have the same data, hence the name ‘replica set’. Data is written to the Primary node. Which then gets replicated to the Secondary nodes.
- redundancy → one server fails, you have multiple copies
- high availability
- fault tolerance
- does NOT equeal disaster recovery
- whay happens on primary replicates to secondary
- data is growing beyond hardware capacity
    - scale vertically: bigger, faster hardware
    - scale horizontally: partition the data
        - helps with increased throughput and capacity
        - could be for legal requirements (split on regions)

Accessing MongoDB from Python

```
from pymongo import MongoClient #official client

uri = "mongodb://USER:PASSWORD@uri/test" #address where mongodb is
client = MongoClient(uri)
campusDB = client.campusManagementDB
students = campusDB.students

students.insert_one( json )
students.insert_many(students list)

students.find_one()
students.find({"lastName": "Doe"})
students.count_documents({"lastName": "Doe"})

#print read documents, find
from bson.json_util import dumps

cursor = students.find({"lastName": "Doe"})
print(dumps(cursor, ident=4))

#replace
student = students.find_one({"lastName": "Doe"})
student["onlineOnly"] = True
student["email"] = "ajdsB"
students.replace_one({"lastName": "Doe"}, student)

#update
student = students.find_one({"lastName": "Doe"})
changes = {"$set": { "onlineOnly": True, "email": "sadasd"}}
students.update_one({"lastName": "Doe"}, changes)
students.update_many({}, {"$set": { "onlineOnly": True }})

#delete
studets.delete_one({"studentId": 123123})
students.delete_many({"graduatedYear": 2019})
```

LAB mongoDB in Python

```
pip install pymongo

start_mongo

python mongo_connect.py
```

mongo_connect.py

```
from pymongo import MongoClient
user = 'root'
password = 'MjE2NzUtbWFqb2Nh' # CHANGE THIS TO THE PASSWORD YOU NOTED IN THE EARLIER E
XCERCISE - 2
```

```
host='localhost'
#create the connection url
connecturl = "mongodb://{}:{}@{}:27017/?authSource=admin".format(user,password,host)


# connect to mongodb server
print("Connecting to mongodb server")
connection = MongoClient(connecturl)


# get database list
print("Getting list of databases")
dbs = connection.list_database_names()

# print the database names

for db in dbs:
    print(db)
print("Closing the connection to the mongodb server")

# select the 'training' database

db = connection.training

# select the 'python' collection

collection = db.python


# create a sample document

doc = {"lab":"Accessing mongodb using python", "Subject":"No SQL Databases"}

# insert a sample document

print("Inserting a document into collection.")
db.collection.insert(doc)

# query for all documents in 'training' database and 'python' collection

docs = db.collection.find()

print("Printing the documents in the collection.")

for document in docs:
    print(document)

# close the server connection
print("Closing the connection.")
connection.close()
```

Summary and Highlights

- CRUD operations consist of Create, Read, Update, and Delete.

- The Mongo shell is an interactive command line tool provided by MongoDB to interact with your databases.
- Indexes help quickly locate data without looking for it everywhere.
- MongoDB stores data being indexed on the index entry and a location of the document on disk.
- Using an aggregation framework, you can perform complex analysis on the data in MongoDB.
- You can build your aggregation process in stages such as match, group, project, and sort.
- Replication is the duplication of data and any changes made to the data.
- Replication provides fault tolerance, redundancy, and high availability for your data.
- For growing data sets, you can use sharding to scale horizontally.
- MongoClient is a class that helps you interact with MongoDB.

Semana 3: Introducing Apache Cassandra

Overview of Cassandra

Apache Cassandra is “an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tunable and consistent database” that bases its distribution design on Amazon's Dynamo and its data model on Google's Bigtable.

Created at Facebook, it is now used at some of the most popular sites on the Web.

Some of the services that use Cassandra are Netflix, Spotify, and Uber.

MongoDB	**Apache Cassandra**
• search use cases, ecommerce websites	• "always available" type of services (netflix, spotify)
• read with indexes	• fast writes — capture all data
• consistency	• availability and scalability

• primary-secondary architecture	• Peer to peer architecture

Key features of Cassandra

- distributed and decentralized
- always available with tunable consistency
- fault tolerant
- high write thorughput by having no read before write by default
- fast and linear scalability
- multiple data center support
- sql-like query language

- does NOT support joins
- limited aggregations support
- limited support for transactions
- no concept of referential integrity or FK

- For joins and aggregations:
  - Cassandra + Spark

Usage scenarios

- when writes exceed read requests
  - storing all the clicks on your website or all the access attempts on your service
- when using append-like type of data
  - not many updates or deletes
- when you can predefine your queries and your data access is by a known primary key

    - data can be partitioned via a key that allows the database to be spread evenly across multiple nodes
- when there's no need for joins and complex aggregations

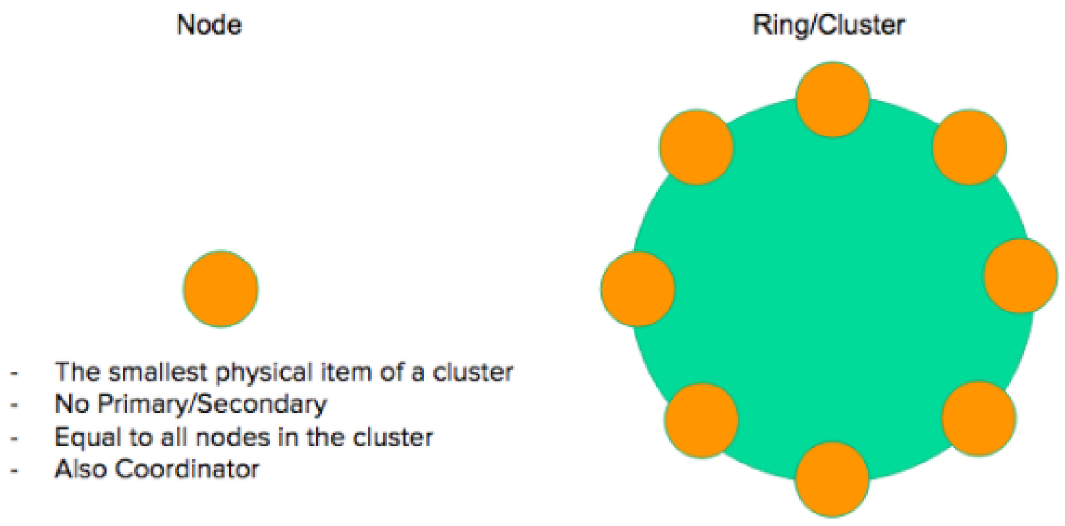
- eCommerce websites
- online services
- timeseries (monitoring servers' logs, sensors, etc)

Architecture of Cassandra

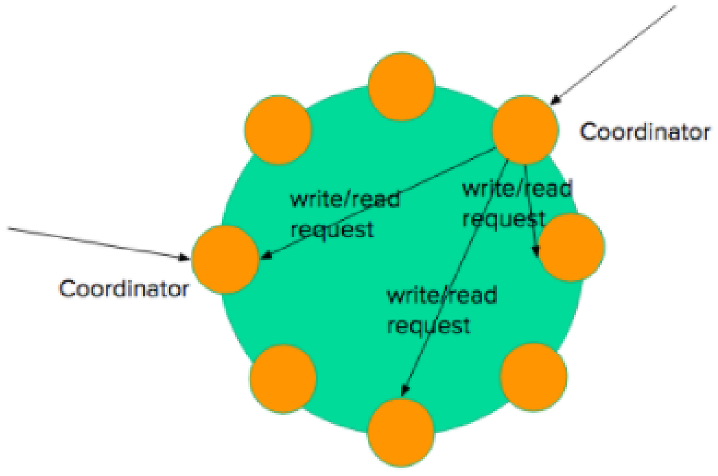
The Apache Cassandra architecture is designed to provide scalability, availability, and reliability to store massive amounts of data. After reading this document, you will have a basic understanding of the components.

Apache Cassandra Topology

Cassandra is based on a distributed system architecture. In its simplest form, Cassandra can be installed on a single machine or container. A single Cassandra instance is called a node. Cassandra supports horizontal scalability achieved by adding more than one node as a part of a Cassandra cluster.


Node
Ring/Cluster
- The smallest physical item of a cluster
- No Primary/Secondary
- Equal to all nodes in the cluster
- Also Coordinator

As well as being a distributed system, Cassandra is designed to be a peer-to-peer architecture, with each node connected to all other nodes. Each Cassandra node can perform all database operations and can serve client requests without the need for a primary node.

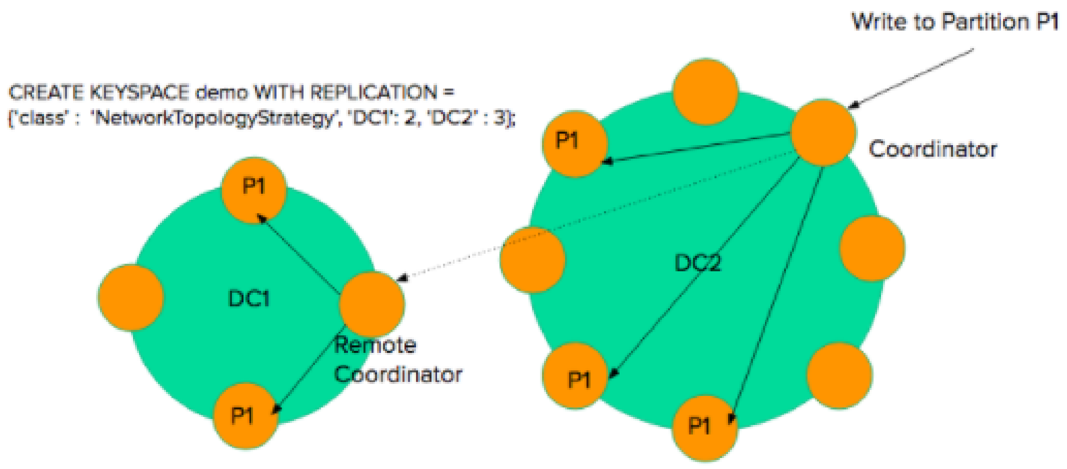


How do the nodes in this peer-to-peer architecture (no primary node) know to which node to route a request and if a certain node is down or up? Through Gossip.

Gossip is the protocol used by Cassandra nodes for peer-to-peer communication. The gossip protocol informs a node about the state of all other nodes. A node performs gossip communications with up to three other nodes every second. The gossip messages follow a specific format and use version numbers to make efficient communication, thus shortly each node can build the entire metadata of the cluster (which nodes are up/down, what are the tokens allocated to each node, etc..).

Multi Data Centers Deployment

A Cassandra cluster can be a single data center deployment (like in the above pics), but most of the time Cassandra clusters are deployed in multiple data centers. A multi data-center deployment looks like below – where you can see depicted a 12 nodes Cassandra cluster, topology wise installed in 2 datacenters. Since replication is being set at keyspace level, demo keyspace specifies a replication factor 5: 2 in data center 1 and 3 in data center 2.


Write to Partition P1
CREATE KEYSPACE demo WITH REPLICATION =
{'class' : 'NetworkTopologyStrategy', 'DC1': 2, 'DC2' : 3};
Coordinator
DC1
DC2
Remote
Coordinator
P1
P1
P1
P1
P1

Note: since a Cassandra node can be as well a coordinator of operations, in our example since the operation came in data center 2 the node receiving the operation becomes the coordinator of the operation, while a node in data center 1 will become the remote coordinator – taking care of the operation in only data center 1.

Components of a Cassandra Node

There are several components in Cassandra nodes that are involved in the write and read operations. Some of them are listed below:

Memtable

Memtables are in-memory structures where Cassandra buffers writes. In general, there is one active Memtable per table. Eventually, Memtables are flushed onto disk and become immutable SSTables.

This can be triggered in several ways:

- The memory usage of the Memtables exceeds a configured threshold.
- The CommitLog approaches its maximum size, and forces Memtable flushes in order to allow Commitlog segments to be freed.
- When we set a time to flush per table.

CommitLog

Commitlogs are an append-only log of all mutations local to a Cassandra node. Any data written to Cassandra will first be written to a commit log before being written to a Memtable. This provides durability in the case of unexpected shutdown. On startup, any mutations in the commit log will be applied to Memtables.

SSTables

SSTables are the immutable data files that Cassandra uses for persisting data on disk. As SSTables are flushed to disk from Memtables or are streamed from other nodes, Cassandra triggers compactions which combine multiple SSTables into one. Once the new SSTable has been written, the old SSTables can be removed.

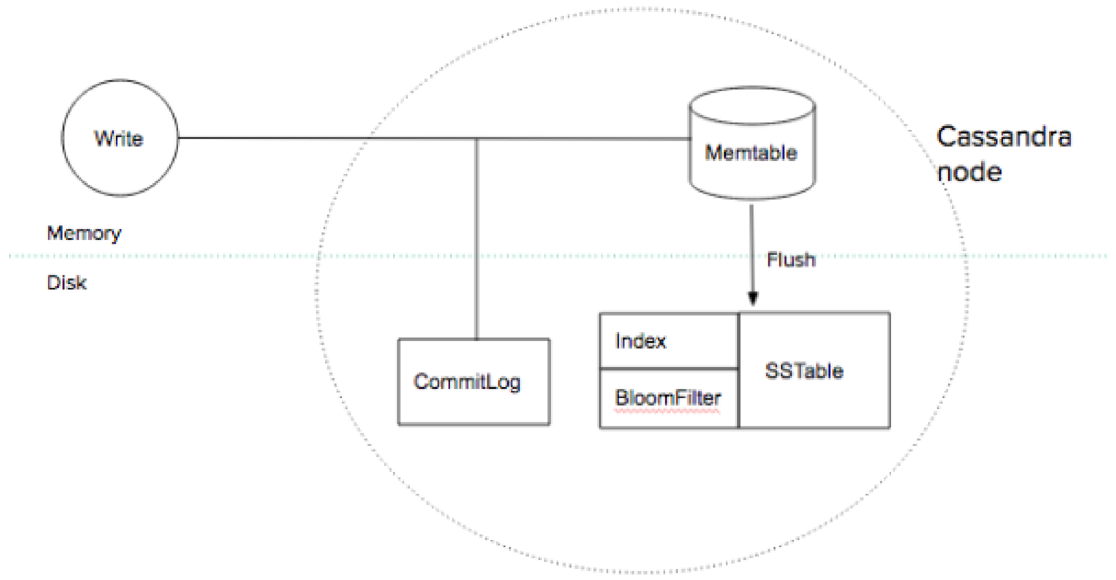
Each SSTable is comprised of multiple components stored in separate files, some of which are listed below:

- **Data.db:** The actual data.
- **Index.db:** An index from partition keys to positions in the Data.db file.
- **Summary.db:** A sampling of (by default) every 128th entry in the Index.db file.
- **Filter.db:** A Bloom Filter of the partition keys in the SSTable.
- **CompressionInfo.db**: Metadata about the offsets and lengths of compression chunks in the Data.db file.

Write Process at Node Level

Cassandra processes data at several stages on the write path, starting with the immediate logging of a write and ending with a write of data to disk:

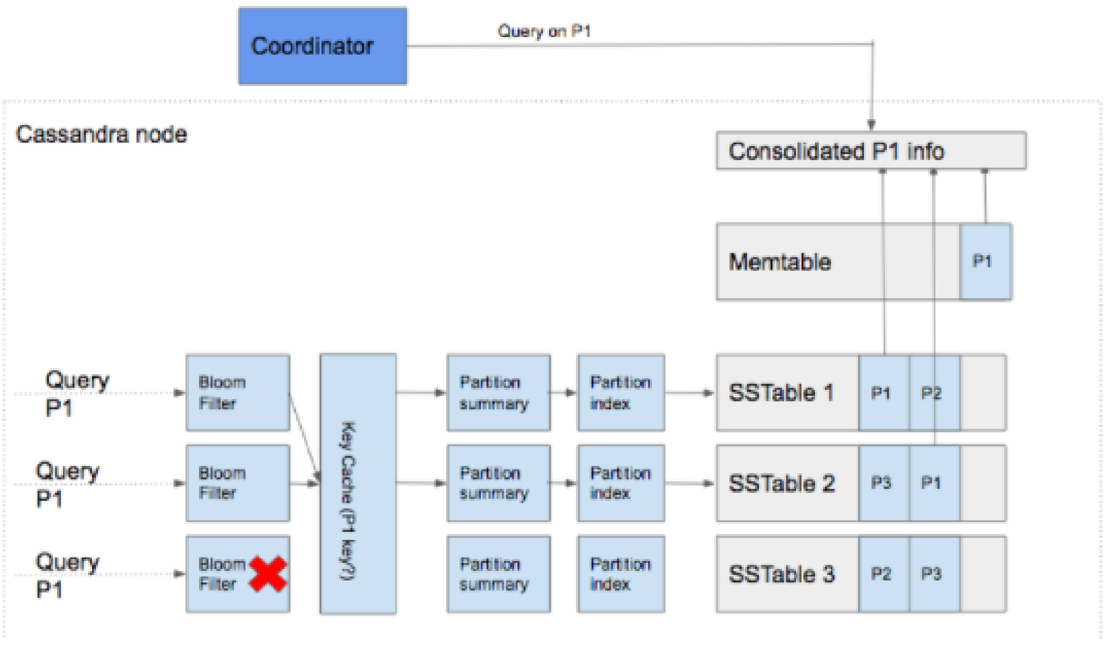
- Logging data in the commit log
- Writing data to the Memtable
- Flushing data from the Memtable
- Storing data on disk in SSTables



Read at node level

While writes in Cassandra are very simple and fast operations, done in memory, the read is a bit more complicated, since it needs to consolidate data from both memory (Memtable) and disk (SSTables). Since data on disk can be fragmented in several SSTables, the read process needs to identify which SSTables most likely contain info about the partitions we are querying - this selection is done by the Bloom Filter information. The steps are described below:

- Checks the Memtable
- Checks Bloom filter
- Checks partition key cache, if enabled
- If the partition is not in the cache, the partition summary is checked
- Then the partition index is accessed
- Locates the data on disk
- Fetches the data from the SSTable on disk
- Data is consolidated from Memtable and SSTables before being sent to coordinator



Key Features of Cassandra

Cassandra groups data based on your declared partition key and then distributes the data in the cluster by hashing each partition key (called tokens). Each Cassandra node has a predefined list of supported token intervals and data is routed to the appropriate node based on the key value hash, and this predefined token allocation in the cluster.

After data is initially distributed in the cluster, Cassandra proceeds with replicating the data. The number of replicas refer to how many nodes contain a certain piece of data at a particular time.

Data Replication is done clockwise in the cluster, taking into consideration the rack and the data center's placement of the nodes. Data Replication is done according to the set Replication Factor – which specifies the number of nodes that will hold the replicas for each partition.

CAP theorem: Cassandra favor availability over consistency

Tunable: Strong or eventual consistency

Consistency conflicts solved during read

NO reading before writing

Cassandra uses CQL, an SQL like syntax

```
CREATE TABLE test (
  groupid uuid,
  name text,
  occupation text,
  age int,
  PRIMARY KEY ((groupid), name));

INSERT INTO test (groupid, name, occupation, age)
  VALUES (1001, 'Thomas', engineer, 24), (1001, 'James', 'designer', 30);

SELECT * FROM test WHERE groupid = 1001;
```

Cassandra Data Model - Part 1

Local Entities

TABLE

- logical entity that organized data storage at cluster and node level (according to declared schema)
- data is organized in tables containing rows of columns
- tables can be created, dropped, altered at runtime without blocking updates and queries
- to create a table you must define a primary key and other data columns (regular columns)
- types
  - STATIC
    - primary key (username)
  - DYNAMIC
    - primary key ((groupid), username)

KEYSPACE

- logical entity that contains one or more tables
- replication and data centers' distribution is defined at keyspace level
- recommended 1 keyspace/applicatio

```
CREATE KEYSPACE intro_cassandra WITH REPLICATION = { 'class' : 'NetworkTopologytrateg
y', 'datacenter1': 2, 'datacenter2': 3 };

USE intro_cassandra; #if you dont specify the keyspace you will need to prefix your ta
ble with the keyspace name: intro_cassandra.groups

CREATE TABLE groups(
  groupid int,
  group_name text STATIC,
  username text,
  age int,
  PRIMARY KEY ((groupid), username)); //groupid is the PARTITION KEY and username is t
he CLUSTERING KEY
```

Primary Key:

- subset of the declared columns
- mandatory and cannot be changed
- two main roles:
    - optimize read performance for table queries - query driven data design
    - provide uniqueness to the entries
- has two ocomponents:
    - partition key — mandatory
        - partition key ⇒ hash (token) ⇒ node
        - determines data locality in cluster
    - clustering key(s) — optional
        - stores the data in ascending or descending order within the partition for the fast retrieval of similar values

Cassandra Data Model - Part 2

- data modeling — build a primary key that optimized query execution time
- choose a partition key — starts answering your query and spreads the data uniformly in the cluster
- minimize the number of partitions read in order to answer the query

Introduction to Cassandra Query Language Shell (cqlsh)

- cql keywords are case-insensitive
- identifiers in cql are case-insensitive unless enclosed in double quotation marks
- comment //

Command line options:

```
>cqlsh [options] [host [port]]

#options:
--help #shows help about the cqlsh command options
--version
```

```
-u -user
-p -password
-k -keyspace #specifies a keyspace to authenticate to
-f -file #enables execution of commands from a given file
--request-timeout #timeout for your queries. Default 10
```

Special commands:

```
CAPTURE #captures the output of a command and adds it to a file

CONSISTENCY #shows the current consistency level and sets a new one

COPY #copies data to and from cassandra
COPY TO #copies to a csv
COPY FROM #copies from csv

DESCRIBE

EXIT

PAGING

TRACING
```

LAB

```
start_cassandra

cqlsh --username cassandra --password NTEwNC1tYWpvY2Fy

show host #returns: Connected to My Cluster at 127.0.0.1:9042.

show version

describe keyspaces

cls #clear command line

exit
```

Summary and Highlights

- Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault tolerant, and tunable and consistent database.

- Apache Cassandra is best used by "always available" type of applications that require a database that is always available.
- Data distribution and replication takes place in one or more data center clusters.
- Its distributed and decentralized architecture helps Cassandra be available, scalable, and fault tolerant.
- Cassandra stores data in tables.
- Tables are grouped in keyspaces.
- A clustering key specifies the order that the data is arranged inside the partition (ascending or descending).
- Dynamic tables partitions grow dynamically with the number of entries.
- CQL is the primary language for communicating with Apache Cassandra clusters.
- CQL queries can be run programmatically using a licensed Cassandra client driver, or they can be run on the Python-based CQL shell client provided with Cassandra.

CQL data types

- BUILT-IN
  - predefined in cassandra
  - ascii, boolean, text, blob, double, uuid
  - blob = binary large object (1MB). For sstoring image or short string
  - bigInt — user for 64-bit signed long integer
- COLLECTION
  - group and store data together in a column
  - relational: a many-to-one joined relationship between a 'users' table and an 'email' table
  - cassandra: no joins ⇒ we store all the data in a collection column in the 'users' table
  - three types
    - lists

            - ordered and with duplicates
        - maps
            - key-value pairs
        - sets
            - unique elements not ordered

```
USE intro_cassandra;

ALTER TABLE users ADD jobs list<text>;

UPDATE users SET jobs = ['Walmart'] + jobs where username = 'Alaidajsbd';

UPDATE users SET jobs[0] = 'Reiss' where username = 'Alaidajsbd'; //replaces Walma
rt with Reiss (lists start from 0)
```

- USER-DEFINED
    - UDTs
    - can attach multiple data fields, each named and typed, to a single column
    - the fields used to create a UDT may be any valid data type, including collections and other existing UDTs
    - once created, the user can alter, verify, and drop a field or the whole data type
    - used in one-to-one relationships
    - once created, UDTs may be used to define a column in a table

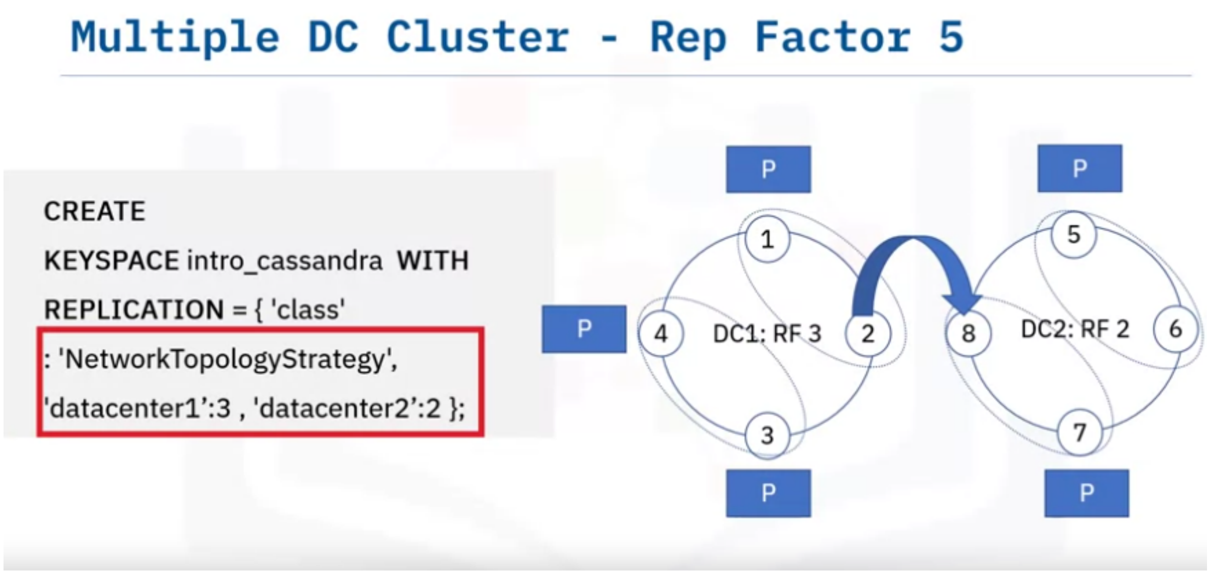
```
CREATE TYPE address (
  Street text,
  Number int,
  Flat text);

//now address is a datatype

CREATE TABLE users_w_address (
  ....
  Location address,
  ....
```

Keyspace Operations

- needs to be defined BEFORE creating tables
- can contain any number of tables, and a table belongs to only one keyspace
- replication is specified at the keyspace level
- you need to specify the replication factor during the creation of keyspace — can be modified later

- REPLICATION FACTOR
  - number of replicas placed on different nodes in the cluster
- REPLICATION STRATEGY
  - which nodes are going to house the replicas
- REPLICAS
  - all replicas are equally important — no primary or secondary replicas
  - replication factor should not exceed the number of cluster nodes

```
CREATE KEYSPACE intro_cassandra WITH REPLICATION = {
  'class' : 'NetworkTopologyStrategy',
  'datacenter1': 3,
  'datacenter2': 2 };

ALTER KEYSPACE intro_cassandra WITH REPLICATION = {
  'class' : 'NetworkTopologyStrategy',
  'datacenter1': 3,
  'datacenter2': 3 };


DROP KEYSPACE intro_cassandra; //cassandra takes a snapshot before dropping the keyspa
ce, so you will be able to recover your data
//Remove the keyspace. Immediate, irreversible removal of the keyspace, including obje
cts such as tables, functions, and data it contains.
```



Multiple DC Cluster - Rep Factor 5

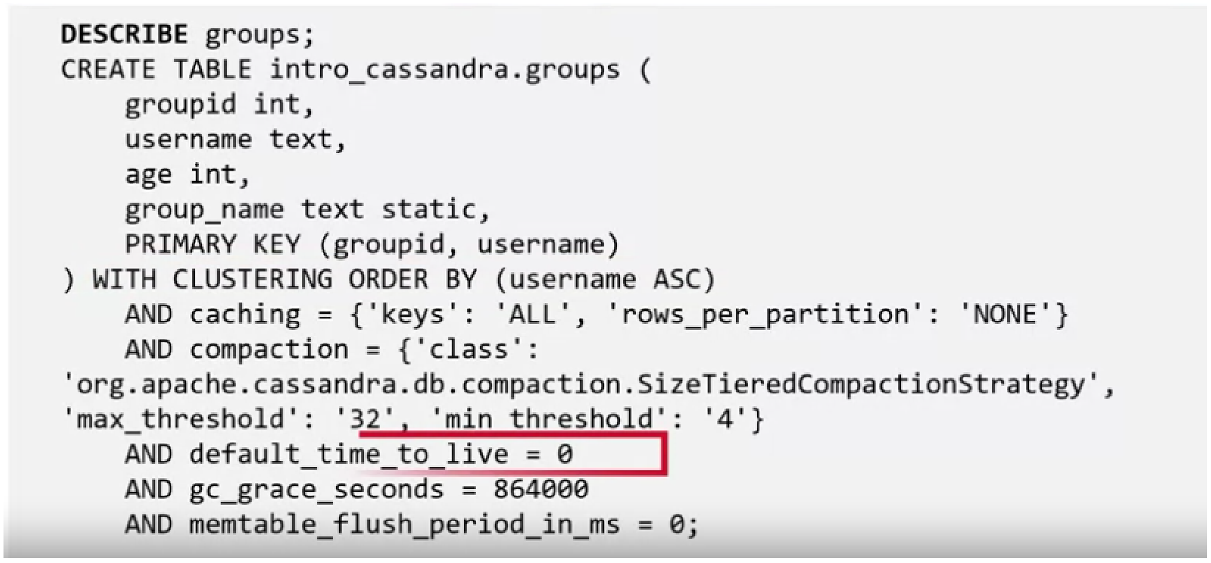
CREATE
KEYSPACE intro_cassandra WITH
REPLICATION = { 'class'
: 'NetworkTopologyStrategy',
'datacenter1':3 , 'datacenter2':2 };

Table Operations

```
CREATE TABLE [IF NOT EXISTS] [keyspace_name.]table_name
( column definition [, ...]
  PRIMARY KEY (column_name [, column_name ...])
[WITH table_options
  | CLUSTERING ORDER BY (clustering_column_name order])
```

```
column_name cql type definition [STATIC | PRIMARY KEY], [, ...]
```

static columns cannot be used in PRIMARY KEYS



```
DESCRIBE groups;
CREATE TABLE intro_cassandra.groups (
    groupid int,
    username text,
    age int,
    group_name text static,
    PRIMARY KEY (groupid, username)
) WITH CLUSTERING ORDER BY (username ASC)
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND compaction = {'class':
'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
'max_threshold': '32', 'min_threshold': '4'}
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND memtable_flush_period_in_ms = 0;
```

CRUD Operations - Part 1

Write operations

- receiving node is the coordinator for the operation
- writes directed to all partition replicas
- acknowledgement expected from consistency number of node
- no reading before writing (by default)
- at node level
  - writes are done in memory and later flushed on disk (SSTables)
  - every flush ⇒ new SSTable
  - all disk writes are sequential ones. Data will be reconciled through Compaction
- cassandra attaches a timestamp to every write. The most recent data wins.

INSERT

- require full primary key
- doesnt perform read before write: an insert can behave as an UPSERT
  - if we insert data in an existing entry, data will be updated
- inserts require a value for all primary key columns, but not other columns

    - only specified columns will be inserted/updated
- insert data with time to live

Lightweight transactions (LWT)

- by using IF syntax we can ask for a read before write through LWT

```
UPDATE groups SET AGE = 62 WHERE groupid = 12 and username = 'adasd' IF EXISTS;

//IF age = 62;

//IF NOT EXISTS
```

Lightweight transactions are at least **four times slower** than the normal INSERT/UPDATE in Cassandra. Use them sparingly in your application.

CRUD Operations - Part 2

Read operations

- receiving node is coordinator for the operation
- reads are directed only to number of replicas required for consistency
- inconsistencies between contacted nodes are repaired during Read process

- start your query using the partition key to limit reads to specific nodes containing your data
- follow the order of your primary key columns
- SELECT * FROM GROUPS: Not okay performance wise, because it will send the request to all nodes in the cluster

Secondary indexes

```
 //WILL NOT WORK
  SELECT * FROM groups WHERE age = 12; //WILL NOT WORK because age is a regular column
//-----

CREATE INDEX ON groups(age);
SELECT * FROM groups WHERE age = 12; // OK
```

Delete

```
//record
DELETE FROM groups WHERE groupid = 12 AND username = 'ASDASD';

//cell
DELETE age FROM groups WHERE groupid = 12 AND username = 'ASDADS'

//partition
DELETE FROM groups WHERE groupid = 12;
```

- DELETE operations in cassandra have a great impact in performance
- deleting data in distributed systems is trickier than in relational databases
- especially in peer-to-peer ones like cassandra
- reads and writes can be directed to any of partitions's replicas, so there's no primary node for a write/read
- cassandra marks all the deleted items with a "tombstone" containing the time of the delete operation. The data still resides on disk for a period of time

<u>**TOMBSTONES**</u>

- a special value to indicate data has been deleted + time of delete
- tombstones prevent deleted data from being returned during reads
- tombstones are deleted at 'gc_grace_seconds' during compaction process. 10 days by default.

LAB

```
//Create a keyspace named training using SimpleStrategy and replication factor of 3.
CREATE KEYSPACE training
WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 3};

//Create a table named movies with columns 'movieid' which is the primary key and inte
ger type, 'moviename' of type text, and 'yearofrelease' of type integer.
use training;
CREATE TABLE movies(
movie_id int PRIMARY KEY,
movie_name text,
year_of_release int
);

//Let us insert a row into the table movies.
```

```
INSERT into movies(
movie_id, movie_name, year_of_release)
VALUES (1,'Toy Story',1995);

//Query the movie name with movie_id 1.
select movie_name from movies where movie_id = 1;

//The movie_id for Scream is 4. It was released in 1996 and not 1995. Here is how you
 modify it.
UPDATE movies
SET year_of_release = 1996
WHERE movie_id = 4;

//Delete the movie with the movie_id 5.
DELETE from movies
WHERE movie_id = 5;
```

Summary and Highlights

- Cassandra supports built-in, collection, and user-defined data types.
- Both collection and user-defined data types offer a way to group and store data together.
- Keyspaces are defined before creating tables, and a keyspace can contain any number of tables.
- Common keyspace operations are CREATE KEYSPACE, ALTER KEYSPACE, and DROP KEYSPACE.
- Cassandra organizes data logically in tables.
- A table’s metadata specifies the primary key – instructing Cassandra how to distribute the table data at the cluster and node level.
- Cluster level writes are sent to all the partition's replicas, irrespective of the consistency factor.
- By default, Cassandra doesn't perform a read before writes, therefore INSERT and UPDATE operations behave similarly.
- Reads at cluster level are sent only to the number of replicas according to the consistency setting.
- Reads should follow the Primary Key columns order for best performance.

Semana 4: Introducing IBM Cloudant

Overview of Cloudant

- DBaaS
- built on open source apache couchDB
- utilises a json document store
- data layer for web and mobile
- simple to use and rich in features
- fully integrated capabilities
- flexible schema
- distributed database optimized for large workloads
- SLA-backed DBaaS means no admin overhead
- DATA REPLICATION technology is engineered to reduce risks, costs, and distractions
- enables developerar to focus on the business application
- poferful replication protocol and API compatible with open source ecosystems
- also compatible with open source libraries for most prevalent mobile and web development stacks
- cloudant and couchdb sharing a common replication protocol allows developers to sync copies of cloudant data to remote couchdb instances at push of a button
- **Cloudant search** implements apache lucene for search speed and simplicity
- **Cloudant geospatial** GeoJSON storage supports the encoded geographic data structures for built-in spatial querying and map visualization
- **Offline first** mobile web app capabilities, build with **cloudant sync**, enable mobile data sync
  For web and mobile applications that require offline sync, Cloudant uses a replication protocol that is compatible with common libraries that exist in a browser, or with Cloudant-provided libraries.
- android and iOS users can work offline and mobile, using stored local data, and then sync their data to the cloud databases

Cloudant Architecture and Key Technologies

- over 55 data centers around the world
- supported on:
  - ibm cloud
  - rackspace
  - microsoft azure
  - amazon web services
- robust and easy-to-use replication protocol
- never corrupting data
- replicates from one data center to another around the world
- geographic load balancing for your data
- if one data center goes offline, requests get routed to another active data center
  - high availability
  - disaster recovery
  - optimal performance
- users are routed to 'closest' data center, using PING timing → users actually get routed to the data center 'fastest' to them. Closest in terms of time, no distance.
- many web and mobile applications require offline sync
- cloudant's replication protocol is compatible with common libraries in your browser or cloudant-provider libraries
- <u>security</u>
  - ISO27001
  - SOC 2 Type 2 compliant
  - HIPAA ready
  - all data is encrypted, at rest and on transist
  - optional user-defined encryption key management through IBM key protect
  - integrates with ibm identity and access management (IAM) for granular access control at the API level

- offerings
    - fully managed service
    - on-premises version
    - hybrid cloud deployment
- **Key technologies**
    - operational datastore — good fit for web/mobile apps
    - document-based NoSQL database — NOT a data warehouse
    - uses a well-defined HTTP API — fits into a modern, service-oriented architecture
    - HTTP API allows interactivity through browser or cURL — returning in JSON format
- **Fully integrated replication and sync processes**
    - MapReduce
    - Full-text search
    - Geospatial

Cloudant Benefits and Solutions

- **Scalability**
    - ibm cloudant scales to an enormous degree
        - scales in size of data
        - scales in number of concurrent users
    - ibm cloudant can be appropiate for:
        - a small startup with an app handling 1GB of data and 10 users
        - an enterprise organization with multiple apps handling petabytes of data and 20 million active users
- **Availability**
    - an operational datastore for online apps
    - aims to be available ALWAYS
- **Durability**

    - ibm cloudant aims to never lose data
    - stores multiple copies of all data across separate physical nodes
- **Partition tolerance**
    - meets the most stringent high availability disaster recovery requirements
    - handles node failures in a cluster
    - handles full data center outages
- **Solutions**
    - horizontally scalable — distribute database across a cluster
    - auto-shards data across a cluster (auto-partitions)
    - start with a small number of nodes and let cloudant run management scripts as your data grows
    - data replicated to servers within the cluster to keep them in sync with each other.
      IBM Cloudant data replication technology provides options engineered to reduce scalability risks, costs, and distractions These capabilities enable developers to focus on the business's applications.
    - failed nodes are re-synced when back online
- When your application reads and writes data, Cloudant uses a load balancer that distributes the reads and writes evenly across the cluster, so the workload is distributed. So if one node fails, the data is still available on another node
- **AI solutions**
    - case study: https://www.ibm.com/case-studies/adventhealth
- **IoT apps**
    - case study: https://www.ibm.com/case-studies/plm-industries-llc

Deployment Options for Cloudant

Three plans:

- lite
- standard

- dedicated hardware

Lite plan

- evaluation, prototyping and development
- fixed limited amount of throughput capacity and data storage
- deleted after 30 days of inactivity

Standard plan

- serverless scaling
- configurable throughput and data storage
- pay-per-hour pricing
- 20gb free data storage

Dedicated hardware plan

- dedicated single-tenant clusters
- choose from ibm cloud, rackspace, amazon or azuer data centers
- optional add-on to run standard plans in a dedicated enviroment
- private endpoints and ip whitelisting
- recurring monthly fee based on number of servers in cluster
    - additional to consumption costs of the standard plans deployed on it

LAB

Credentials for cloudant

```
"apikey": "EmONHe4dXTKLo9ynVQyh8-skDhi17fsVQRi5k5MzeLXB",
"password": "09443dfb22d111f4a9117529dbd5ee15",
"url": "https://apikey-v2-mymxpa79o27yx46ufl2f3c200ouuqncr0dhq7ceklbu:09443dfb22d111f4
a9117529dbd5ee15@53872227-bc02-472d-a0c3-d24f5f0acf43-bluemix.cloudantnosqldb.appdomai
n.cloud",
"username": "apikey-v2-mymxpa79o27yx46ufl2f3c200ouuqncr0dhq7ceklbu"
```

Note: If you do not see the password field in your credentials, it is because in Exercise 2, Step 5 you have not selected "**IAM and legacy credentials**". You can fix it by deleting your current Cloudant instance and following this lab from the beginning to create a new instance of Cloudant.

Summary and Highlights

- Cloudant is a fully managed DBaaS built on Apache CouchDB that uses a JSON document store.
- Cloudant is a distributed database optimized for large workloads, web, mobile, IoT, and serverless apps.
- Cloudant offers a powerful replication protocol.
- Cloudant's cloud architecture provides high availability, disaster recovery, and optimal performance.
- Cloudant can be offered as a fully managed service, an on-premises deployment, or a hybrid cloud deployment.
- Key technology components of Cloudant are HTTP API, MapReduce, Full-text Search, and Geospatial.
- Cloudant’s key benefits are scalability, availability, durability, partition tolerance, and online upgrades.
- Cloudant uses a document database for improved security and querying.
- Typical use cases for Cloudant include building web and mobile apps, AI solutions, and analysis of IoT data.
- IBM Cloudant has three deployment options

Dashboards in Cloudant

It allows me to:

- create and manage databases
    - create DB
    - add documents
    - replicate a DB
    - set permissions

    - delete a DB
    - run queries on DB documents
- configure replication
    - type
        - one time: occurs once after replication is configured
        - continuous: changes replicated persistently until cancelled
            - increases cost
            - constant impact on performance
    - source database
    - target
    - database
    - authentication
- view active tasks
- monitor current operations
- monitor denied requests
    - 429: too many resuests
    - when throughput capacity is exceeded
- monitor storage consumption
    - daily record of current storage capacity

LAB

```
//first document created
{
  "_id": "1",
  "Topic" : "NoSQL Databases",
  "Lesson" : "IBM Cloudant"
}
```

```
//this is te equivalent in cloudant to do select * from ...
{
   "selector": {}
}

//Select all fields in all documents with _id greater than 4
{
   "selector": {
      "_id": {
         "$gt": "4"
      }
   }
}

//Select the fields _id, square_feet and price in all documents
{
   "selector": {},
   "fields": [
      "_id",
      "price",
      "square_feet"
   ]
}

//Select the fields _id, bedrooms and price in documents with _id greater than 2 and s
ort by _id ascending
{
   "selector": {
      "_id": {
         "$gt": "2"
      }
   },
   "fields": [
      "_id",
      "price",
      "square_feet"
   ],
   "sort": [
      {
         "_id": "asc"
      }
   ]
}
```

Working with Databases in Cloudant

Non-partitioned databases

- no partitioning scheme defined
- only provide global querying

- older database tyoe for backward compatibility
- documents are distributed randomly
- no relation between document's ID and shard it is located on

Partitioned databases

- provide partitioned and global querying
- offer performance and cost benefits
- newer database type
- require specification of logical partitioning
- documents are assigned to a partition using a partiton key
- partitioned databases are **highly** recommended
- IMPORTANT: you **cannot** change the partition type after database creation

Cloudant database

- document based
- uses json document store
- collections = databases in cloudant (not tables)
- databases contain a number of documents

HTTP API Basics

- cloudant dashboard uses web-based HTTP calls to cloudant's API
- databases can be accessed from any internet device using standard HTTP library

CURL

- free, opensource, command-line tool. Makes HTTP requests
- part of Client URL (cURL) project
- get and send data using URL syntax
- also supports HTTPS using secure sockets layer
- github.com/curl/curl

```
//retrieve webpage
curl https://www.ibm.com/cloud

//to create a variable for cloudant's url:
export URL = "https://username:password@host"

//create an alias. sgH SILENT: doesnt show messages
alias acurl="curl -shH 'Content-type: application/json'"

//to test connectivity
acurl $URL/

//view all databases
acurl $URL/_all_dbs

//view all databases sending to python json formatter
acurl $URL/_all_dbs | python -m json.tool
//another json formatter
acurl $URL/_all_dbs | jq

//view the details for a single database
acurl $URL/training | python -m json.tool

//view the documents in a database
acurl $URL/training/_all_docs?include_docs=true

//retrieve single document
acurl $URL/training/<document id>
```

Working with the HTTP API

- when we run a command in CURL without a specific method, it defaults using HTTP GET

Other HTTP methods (VERBS)

- PUT = create a database, create a document or modify an existing document (use with document ID)
- POST = run a query, create an index, create a document (use with -d switch)
- DELETE = delete a database, or document, or index

Curl uses -X to specify the HTTP method

```
curl -X PUT $URL/training

//dropping a database
curl -X DELETE $URL/training
```

```
//inserting a document
curl -X PUT $URL/training/"212" -d '{"coursname":"Excel basics", "level":"beginner"}'

//updating a document
curl -X PUT $URL/training/212 -d {"coursname":"Excel basics", "level":"beginner", "edi
tion":"third", "_rev":"1-123456"}'
//error response
//{"error":"conflict", "reason":"document update conflict."}
//expected response
//{"ok":true, "id":"212", "rev":"1-234567"} ----> NOTE DIFFERENT NUMBERS

//delete a document
curl -X DELETE $URL/training/212?rev=1-234567
```

Cloudant uses a query language based on MongoDB

```
//find document with ID '212'
curl -X POST $URL/training/_find \
-H "Content-Type: application/json" \
-d '{"selector":{"_id":"212"}}'

//greater than $gt
//less than $lt
curl -X POST $URL/training/_find \
-H "Content-Type: application/json" \
-d '{"selector":{"modules":"{"$gt":4}}'

//store query in .json file
curl -X POST $URL/training/_find_ \
-d@myquery.json
```

Cloudant Indexes

Cloudant - Query optimization with indexes

In databases, frequently used data and related queries are indexed to speed up queries. Cloudant supports two types of indexes:

- “type”: “text”
- “type”: “json”

There is a significant difference between these two types of indexes related to the purpose of use and the method of use.

From the perspective of purpose of use, the index of the text type focuses on the specific content of Cloudant documents, rather than the structure of the document itself. So, if a user is not familiar with the structures of the documents in the database, or the structures of the documents vary a lot and their formats are complex, text index will be preferred.

In contrast, the JSON index has high requirements on the structure of the document, and it is built on one or more specific fields. Therefore, if the user is familiar with the document structure
in the database, he or she can choose to create a JSON index.

From the perspective of method of use, the text index can only be used in the Cloudant data search interface, of which the supported syntax is Apache Lucene Query Parse Syntax.

In contrast, the JSON index can only be used in Cloudant's data query interface, allowing users to query with a JSON object in accordance with the Cloudant query parsing syntax. Interfaces of both
indexing type are powerful and support various custom queries.

In terms of the speed of indexing, the indexing of text type is slower than that of JSON type when dealing with the same amount of data. The reason is, when creating a text index, Cloudant deals with not only the specified data structure, but also its content. In contrast, the JSON index is only concerned with the structure itself. In some cases, the text index can be used for full-text search of the entire database.

In addition, some of Cloudant's advanced features, such as sophisticated aggregations and geospatial-based calculations, can only be used in query interfaces based on JSON index. Therefore, the key to selecting which of these two types of index to use lies in the understanding of existing data.

How to create indexes on Cloudant

```
//Example 1: This command creates a json index on the field "firstname", on the employ
ees database
curl -X POST $SERVERURL/employees/_index \
-H"Content-Type: application/json" \
-d'{
    "index": {
        "fields": ["firstname"]
    }
}'

//Example 2: This command creates a json index on the field "firstname", and names the
index as firstname-index. You can mention JSON index explicitly. The default is JSON i
ndex.
curl -X POST $SERVERURL/employees/_index \
-H"Content-Type: application/json" \
-d'{
    "index": {
        "fields": ["firstname"]
    },
    "name" : "firstname-index",
    "type" : "json"
}'

//Example 3: This command creates a text index on the employees database.
curl -X POST $SERVERURL/employees/_index \
-H"Content-Type: application/json" \
-d'{ "index": {},
     "type": "text"
}'
```

LAB

From command-line:

```
export CLOUDANTURL="https://apikey-v2-mysdssdmxpa79o27yx46ufl2f3c200ouuqncr0dhq7ceklb
u:09443dfb22d111f4a9117sda29dbd5ee15@53872227-bc02-472d-a0c3-d24f5f0acf43-bluemix.clou
dantnosqldb.appdomain.cloud"

//check connection
curl $CLOUDANTURL

//check credentials
curl $CLOUDANTURL/_all_dbs

//create database
curl -X PUT $CLOUDANTURL/animals

//insert a document
curl -X PUT $CLOUDANTURL/planets/"1" -d '{
    "name" : "Mercury" ,
    "position_from_sum" :1
     }'

//update document
curl -X PUT $CLOUDANTURL/planets/1 -d '{
    "name" : "Mercury" ,
    "position_from_sum" :1,
    "revolution_time":"88 days",
    "_rev":"1-3fb3ccfe80573e1ae334f0cfa7304f6c"
    }'

//verify by listing document with id = 1
curl -X GET $CLOUDANTURL/planets/1

//create and populate DIAMOND database
curl -X DELETE $CLOUDANTURL/diamonds
curl -X PUT $CLOUDANTURL/diamonds
curl -X PUT $CLOUDANTURL/diamonds/1 -d '{
    "carat": 0.31,
    "cut": "Ideal",
    "color": "J",
    "clarity": "SI2",
    "depth": 62.2,
    "table": 54,
    "price": 339
  }'

curl -X PUT $CLOUDANTURL/diamonds/2 -d '{
    "carat": 0.2,
    "cut": "Premium",
    "color": "E",
    "clarity": "SI2",
    "depth": 60.2,
    "table": 62,
    "price": 351

  }'

curl -X PUT $CLOUDANTURL/diamonds/3 -d '{
    "carat": 0.32,
    "cut": "Premium",
    "color": "E",
    "clarity": "I1",
    "depth": 60.9,
    "table": 58,
    "price": 342

  }'


curl -X PUT $CLOUDANTURL/diamonds/4 -d '{
    "carat": 0.3,
    "cut": "Good",
    "color": "J",
    "clarity": "SI1",
    "depth": 63.4,
    "table": 54,
    "price": 349

  }'

curl -X PUT $CLOUDANTURL/diamonds/5 -d '{
    "carat": 0.3,
    "cut": "Good",
    "color": "J",
    "clarity": "SI1",
    "depth": 63.8,
    "table": 56,
    "price": 347

  }'

curl -X PUT $CLOUDANTURL/diamonds/6 -d '{
    "carat": 0.3,
    "cut": "Very Good",
    "color": "J",
    "clarity": "SI1",
    "depth": 62.7,
    "table": 59,
    "price": 349
  }'

curl -X PUT $CLOUDANTURL/diamonds/7 -d '{
    "carat": 0.3,
    "cut": "Good",
    "color": "I",
    "clarity": "SI2",
    "depth": 63.3,
    "table": 56,
    "price": 343
  }'

curl -X PUT $CLOUDANTURL/diamonds/8 -d '{
    "carat": 0.23,
    "cut": "Very Good",
    "color": "E",
    "clarity": "VS2",
    "depth": 63.8,
    "table": 55,
    "price": 339
  }'

curl -X PUT $CLOUDANTURL/diamonds/9 -d '{
    "carat": 0.23,
    "cut": "Very Good",
    "color": "H",
    "clarity": "VS1",
    "depth": 61,
    "table": 57,
    "price": 323
  }'

curl -X PUT $CLOUDANTURL/diamonds/10 -d '{
    "carat": 0.31,
    "cut": "Very Good",
    "color": "J",
    "clarity": "SI1",
    "depth": 59.4,
    "table": 62,
    "price": 346
  }'

//Query for diamonds with carat size of 0.3
curl -X POST $CLOUDANTURL/diamonds/_find \
-H"Content-Type: application/json" \
-d'{ "selector":
        {
            "carat":0.3
        }
    }'

//the last query returns warning: "No matching index found, create an index to optimiz
```

```
e query time."

//create index
curl -X POST $CLOUDANTURL/diamonds/_index \
-H"Content-Type: application/json" \
-d'{
    "index": {
        "fields": ["price"]
    }
}'

//query for diamonds
curl -X POST $CLOUDANTURL/diamonds/_find \
-H"Content-Type: application/json" \
-d'{ "selector":
        {
            "price":
                {
                    "$gt":345
                }
        }
    }'
```

How to Access Documentation and Support Resources

There is a plethora of further information about IBM Cloudant online, including documentation, training content, and support resources.

Here are some useful direct links to those resources:

- Documentation – https://cloud.ibm.com/docs/Cloudant
- Training Content – https://cloud.ibm.com/docs/Cloudant?topic=Cloudant-learning-center
- Support Resources – https://cloud.ibm.com/docs/get-support

Summary and Highlights

- You use the Cloudant dashboard to create and manage databases, configure replication, view active tasks, and monitor Cloudant.
- You can access your Cloudant service instance on the IBM Cloud dashboard from the Resources list in the Services list.
- You use partitioned databases to get the best long-term performance out of your databases if your data model allows for logical partitioning of documents.

- When you create an IBM Cloudant database, you must select whether you want the database to be partitioned or non-partitioned.
- A Cloudant document must be a JSON object that contains key-value attributes.
- You can view your database documents by three different views (Metadata, Table, JSON).
- You can use the curl command line tool to make HTTP requests to the Cloudant API.
- You specify an HTTP method to create a database, add a document, or delete a document.
- To update a document in one of your databases, you need the document's revision token value.
- You can run a querying operation to find out information about the documents in your database or to retrieve a subset of documents that match certain criteria.

Semana 5: Final project

Hands-on Lab: Setup & Practice Assignment

```
#You need the 'couchimport' and 'couchexport' tools to move data in and out of the Clo
udant database.
sudo npm install -g couchimport

#Verify that the tool got installed, by running the below command on the terminal.
couchimport --version

#You need the 'mongoimport' and 'mongoexport' tools to move data in and out of the mon
godb database.
wget https://fastdl.mongodb.org/tools/db/mongodb-database-tools-ubuntu1804-x86_64-100.
3.1.tgz
tar -xf mongodb-database-tools-ubuntu1804-x86_64-100.3.1.tgz
export PATH=$PATH:/home/project/mongodb-database-tools-ubuntu1804-x86_64-100.3.1/bin
echo "done"

#verify
mongoimport --version

#Before going ahead set the environment varible CLOUDANTURL to your actual cloudant ur
l from your service credentials.
export CLOUDANTURL="https://apikey-v2-mymxpa79asdasdsdo27yx46ufl2f3c200ouuqncr0dhq7cek
lbu:09443ddfb22d111f4a9117529dadbd5ee15@53872227-bc02-472d-a0c3-d24f5f0acf43-bluemix.c
loudantnosqldb.appdomain.cloud"
```

```
#Export data from the 'diamonds' database into csv format.
couchexport --url $CLOUDANTURL --db diamonds --delimiter ","

#Export data from the 'diamonds' database into json format (one document per line).
couchexport --url $CLOUDANTURL --db diamonds --type jsonl

#Export data from the 'diamonds' database into json format and save to a file named 'd
iamonds.json'.
couchexport --url $CLOUDANTURL --db diamonds --type jsonl > diamonds.json

#Export data from the 'diamonds' database into csv format and save to a file named 'di
amonds.csv'.
couchexport --url $CLOUDANTURL --db diamonds --delimiter "," > diamonds.csv

#start mongo db server
start_mongo
#• The browser at: https://majocarbajal-8081.theiadocker-1-labs-prod-theiak8s-4-tor01.
proxy.cognitiveclass.ai
#CommandLine: mongo -u root -p MzE1NTctbWFqb2Nh --authenticationDatabase admin local

#Import data in 'diamonds.json' into a collection named 'diamonds' and a database name
d 'training'.
mongoimport -u root -p MzE1NTctbWFqb2Nh --authenticationDatabase admin --db training -
-collection diamonds --file diamonds.json

#Export data into json format.
mongoexport -u root -p MzE1NTctbWFqb2Nh --authenticationDatabase admin --db training -
-collection diamonds --out mongodb_exported_data.json

#Export the fields _id,clarity,cut,price from the 'training' database, 'diamonds' coll
ection into a file named 'mongodbexporteddata.csv'
mongoexport -u root -p MzA2NDAtcnNhbm5h --authenticationDatabase admin --db training -
-collection diamonds --out mongodb_exported_data.csv --type=csv --fields _id,clarity,c
ut,price

#Start the Cassandra server.
start_cassandra

#Import csv into cassandra.
#Import 'diamonds.csv' into the 'training' keyspace and the 'diamonds' table/column fa
mily.
#Step - 1: Login to cqlsh.
#Step - 2: Create a keyspace named 'training'.
#Step - 3: In the 'training' keyspace create a table named 'diamonds' with the below s
chema
#    id - primary key (use 'id' as the primary key; Cassandra does not allow you to cr
eate a column starting with underscore(_))
#    clarity - text
#    cut - text
#    price - integer.
use training;
COPY training.diamonds(id,clarity,cut,price) FROM 'mongodb_exported_data.csv' WITH DEL
IMITER=',' AND HEADER=TRUE;

COPY diamonds TO 'cassandra-diamonds.csv';

create index price_index on diamonds(price);
```

Final Assignment

Instructions

Now that you are equipped with the knowledge and skills to work with several different NoSQL databases you have the opportunity in the final project to practice and apply your skills by working with different databases to move and analyze data.

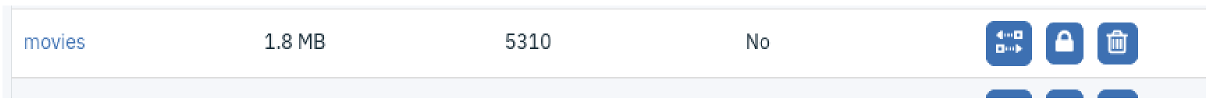
Scenario

You are a data engineer at a Data Analytics Consulting Company. Your company prides itself in being able to efficiently handle data in any format on any database on any platform. Analysts in the offices need to work with data on different databases, and with data in different formats. While they are good at analyzing data, they count on you to be able to move data from external sources into various databases, move data from one type of database to another, and be able to run basic queries on various databases.

```
#You need the 'couchimport' and 'couchexport' tools to move data in and out of the Clo
udant database.
sudo npm install -g couchimport

#Verify that the tool got installed, by running the below command on the terminal.
couchimport --version

#You need the 'mongoimport' and 'mongoexport' tools to move data in and out of the mon
godb database.
wget https://fastdl.mongodb.org/tools/db/mongodb-database-tools-ubuntu1804-x86_64-100.
3.1.tgz
tar -xf mongodb-database-tools-ubuntu1804-x86_64-100.3.1.tgz
export PATH=$PATH:/home/project/mongodb-database-tools-ubuntu1804-x86_64-100.3.1/bin
echo "done"

#verify
mongoimport --version

#CLOUDANTURL
export CLOUDANTURL="https://apikey-v2-mymxpa79o27yx46ufl2f3c200ouuqncr0dhq7ceklbu:0944
```

```
3dfb22d111f4a9117529dbd5ee15@53872227-bc02-472d-a0c3-d24f5f0acf43-bluemix.cloudantnosq
ldb.appdomain.cloud"
```

Replicate database (remote): https://examples.cloudant.com/query-movies



movies	1.8 MB	5310	No

```
#Create an index for the "director" key, on the 'movies' database using the HTTP API
curl -X POST $CLOUDANTURL/movies/_index \
-H"Content-Type: application/json" \
-d'{
    "index": {
        "fields": ["director"]
    }
}'

#Write a query to find all movies directed by 'Richard Gage' using the HTTP API
curl -X POST $CLOUDANTURL/movies/_find \
-H"Content-Type: application/json" \
-d'{ "selector":
        {
            "director":"Richard Gage"
        }
    }'

#Create an index for the "title" key, on the 'movies' database using the HTTP API
curl -X POST $CLOUDANTURL/movies/_index \
-H"Content-Type: application/json" \
-d'{
    "index": {
        "fields": ["title"]
    }
}'

#Write a query to list only the "year" and "director" keys for the 'Top Dog' movies us
ing the HTTP API
curl -X POST $CLOUDANTURL/movies/_find \
-H"Content-Type: application/json" \
-d'{
   "selector": {
      "title": "Top Dog"
   },
   "fields": [
      "year",
      "director"
   ]
}'
```

```
#Export the data from the 'movies' database into a file named 'movies.json'
couchexport --url $CLOUDANTURL --db movies --type jsonl > movies.json
```

```
#Import 'movies.json' into mongodb server into a database named 'entertainment' and a
 collection named 'movies'
start_mongo

mongoimport -u root -p MjYwMTUtbWFqb2Nh --authenticationDatabase admin --db entertainm
ent --collection movies --file movies.json

#login
mongo -u root -p MjYwMTUtbWFqb2Nh --authenticationDatabase admin local

use entertainment
```

```
#Write a mongodb query to find the year in which most number of movies were released
#select title from movies
#group by year order desc
#limit 1

#select title, year from movies
db.movies.find({}, {title:1, year:1})
#returns
{ "_id" : "213249", "title" : "Skazka o zvezdnom malchike", "year" : 1983 }
{ "_id" : "221028", "title" : "Bon Jovi: Live from London", "year" : 1995 }
{ "_id" : "219176", "title" : "Michael Jordan, Above and Beyond", "year" : 1996 }

#select title, year from movies order by year asc
db.movies.find({}, {title:1, year:1}).sort( {year:1} )
#returns
{ "_id" : "989831", "title" : "Jibon Thekey Neya", "year" : 1970 }
{ "_id" : "67517", "title" : "Officers", "year" : 1971 }
{ "_id" : "67242", "title" : "In the Name of the Italian People", "year" : 1971 }

#SELECT year, SUM(year) AS total FROM movies GROUP BY year ORDER BY total
db.movies.aggregate( [
   {
     $group: {
        _id: "$year",
        total: { $sum: "$year" }
     }
   },
   { $sort: { total: 1 } }
] )

#SELECT year, count(*) FROM movies GROUP BY year HAVING count(*) > 1
db.movies.aggregate( [
   {
     $group: {
        _id: "$year",
        count: { $sum: 1 }
     }
   },
   { $match: { count: { $gt: 1 } } }
] )
#returns
{ "_id" : null, "count" : 2 }
{ "_id" : 1990, "count" : 110 }
{ "_id" : 1997, "count" : 166 }
{ "_id" : 1974, "count" : 71 }
{ "_id" : 2003, "count" : 346 }
{ "_id" : 1980, "count" : 78 }

#SELECT year, count(*) FROM movies GROUP BY year ORDER BY year LIMIT 2
db.movies.aggregate( [
   {
     $group: {
        _id: "$year",
        count: { $sum: 1 }
     }
   },
   { $sort: { count: 1 } },
   { $limit : 2 }
] )
#returns
{ "_id" : null, "count" : 2 }
{ "_id" : 1970, "count" : 61 }

#another solution with same result
db.movies.aggregate([
{
    "$group":{
        "_id":"$year",
        "moviecount":{"$sum":1}
        }
},
{ $sort: { moviecount: 1 } },
])
```

```
#Write a mongodb query to find the count of movies released after the year 1999
db.movies.aggregate([{ $match: {"year": {$gt: 1999}} }, { $count: "movies_released_afe
r_1999" }])
```

```
#Write a query to find out the average votes for movies released in 2007
db.movies.aggregate([
   { $match: {year: 2007} },
   { $project: { avgVotes: { $avg: "$imdb.votes"}, title: 1 , _id: 0} }
])
```

```
#Export the fields _id, "title", "year", "rating" and "director" from the 'movies' col
lection into a file named partial_data.csv
mongoexport -u root -p MjYwMTUtbWFqb2Nh --authenticationDatabase admin --db=entertainm
```

```
ent --collection=movies --type=csv --fields=_id,title,year,'imbd.raiting',director  --
out=partial_data.csv
#exports everything but still cant get imbd.rating !!!!!!! #check
```

```
#Import 'partial_data.csv' into cassandra server into a keyspace named 'entertainment'
and a table named 'movies'
start_cassandra

cqlsh --username cassandra --password MTc4NTUtbWFqb2Nh

CREATE KEYSPACE entertainment WITH REPLICATION = {
  'class' : 'NetworkTopologyStrategy',
  'datacenter1': 3,
  'datacenter2': 2 };

use entertainment;

CREATE TABLE movies (
  id text PRIMARY KEY,
  title text,
  year text,
  raiting text,
  director text);

COPY entertainment.movies (id,title,year,raiting,director) FROM 'partial_data.csv' WIT
H DELIMITER=',' AND HEADER=TRUE;
```

```
#Write a cql query to count the number of rows in the 'movies' table
SELECT COUNT(*) FROM MOVIES;
```

```
#Create an index for the "rating" column in the 'movies' table using cql
CREATE INDEX ON movies(raiting);
```

```
#Write a cql query to count the number of movies that are rated 'G'.
SELECT COUNT(*) FROM MOVIES WHERE RAITING = 'G';
```

MongoDB help

PDF con varios ejemplos:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/6e194df1-8e0f-4823-8c03-80c2797b2046/MongoDB_tutorial.pdf

Equivalencias:

SQL to Aggregation Mapping Chart

The aggregation pipeline allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL. The following table provides an

https://docs.mongodb.com/manual/reference/sql-aggregation-comparison/

```
#This command lists all the documents without the name field in the output.
db.languages.find({},{"name":0})

#This command lists all the "object oriented" languages with only "name" field in the output.
db.languages.find({"type":"object oriented"},{"name":1})

#Using the $limit operator we can limit the number of documents printed in the output.
This command will print only 2 documents from the marks collection.
db.marks.aggregate([{"$limit":2}])

#This command sorts the documents based on field marks in ascending order.
db.marks.aggregate([{"$sort":{"marks":1}}])

#This command sort the documents based on field marks in descending order.
db.marks.aggregate([{"$sort":{"marks":-1}}])

#Let us create a two stage pipeline that answers the question "What are the top 2 marks?".
db.marks.aggregate([
{"$sort":{"marks":-1}},
{"$limit":2}
])

#This aggregation pipeline prints the average marks across all subjects.
db.marks.aggregate([
{
    "$group":{
        "_id":"$subject",
        "average":{"$avg":"$marks"}
        }
}
])
```

```
#The above query is equivalent to the below sql query.
#SELECT subject, average(marks) FROM marks GROUP BY subject

#Now let us put together all the operators we have learnt to answer the question. "Who are the top 2 students by average marks?" This involves:
#    finding the average marks per student.
#    sorting the output based on average marks in descending order.
#    limiting the output to two documents.
db.marks.aggregate([
{
    "$group":{
        "_id":"$name",
        "average":{"$avg":"$marks"}
        }
},
{
    "$sort":{"average":-1}
},
{
    "$limit":2
}
])

#Find the total marks for each student across all subjects.
db.marks.aggregate([
    {
        "$group":{"_id":"$name","total":{"$sum":"$marks"}}
    }
])

#Find the maximum marks scored in each subject.
db.marks.aggregate([
    {
        "$group":{"_id":"$subject","max_marks":{"$max":"$marks"}}
    }
])
```

Final quiz

Which requirement would prompt you to consider choosing NoSQL over RDBMS?

Flexible schema.

What is one way that a distributed NoSQL database usually shards data?

By grouping all records that have the same key on the same server.

What does the acronym ACID stand for?

Atomic, Consistent, Isolated, Durable

What are the four main types of NoSQL databases?

Key-value, Document, Column, Graph

Which of the following statements best describes NoSQL?

NoSQL is a family of open source non-relational databases that differ greatly in style and technology

In MongoDB, which of the following common aggregation stages takes the outcome from the previous stage and stores it in a target collection?

WRONG ANSWER: $project

True or False: MongoDB follows a design-first, code-later approach.

False

In MongoDB, what is a group of similar stored documents called?

A collection

Which of the following is a distinguishing characteristic of a compound index in MongoDB?

When a single index structure holds reference to more than one field. MongoDB stores data being indexed on the index entry and a location of the document on disk.

How does sharding work to improve performance in MongoDB?

WRONG ANSWER: Replication

What would you most likely use blobs for in Cassandra Query Language (CQL)?

Storing multimedia objects

Which of these four approaches is the slowest way to make data changes in Apache Cassandra?

> Lightweight transactions → You can instruct Cassandra to look for the data, read it, and only then perform a given operation by using Lightweight Transactions, but Lightweight Transactions are slower than the normal INSERT/UPDATE in Cassandra

What is Apache Cassandra most useful for?

> Fast data storafe and retrieval

When would you be more likely to select MongoDB instead of Apache Cassandra?

> When the need for consistency outweights the neeed for high availability and scalabilitiy

What do you need to start data distribution in Cassandra?

> WRONG ANSWER: Data sets

When you create a new IBM Cloudant database, what do you need to select?

> The partitioning type

Which of the following are valid HTTP methods used in curl for managing IBM Cloudant databases?

> GET, PUT, POST, DELETE

What would be an especially good use case for the replication protocol in IBM Cloudant?

> Web and mobile applications that require offline sync

What does IBM Cloudant use ping timing to do?

> WRONG ANSWER: Geographic load balancing

What is the IBM Cloudant Query equivalent of the WHERE clause in SQL?

Selector