Session 3

NoSQL Database Systems

Big Data Analytics Technology, MSc in Data Science,
Coventry University UK

Miyuru Dayarathna

Outline

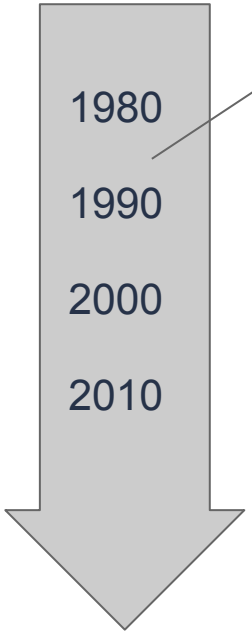
- Introduction to NoSQL
- CAP Theorem
- Categories of NoSQL data stores
  - Document Stores
  - Column Stores
  - KV Stores
  - Graph DB
- Architecture and Performance aspects

The problem of Big Data

- Increasingly organizations have been collecting and storing vast amounts of data.
  - Billions of videos and images with descriptive annotations
  - Hundreds of billions of text documents
  - Trillions of log records capturing human activity
- Big data storage systems have to solve distributed data storage problems

History of Relational Databases

- Persistence
- Transactions
  - Follows ACID properties
- SQL
  - feature-rich and uses a simple, declarative syntax
- Integration/Reporting



Rise of relational

SQL

- SQL provides
  - Relational Model
  - Strong typing
  - ACID Compliance
  - Normalization

ACID Model

- ACID database transaction model ensures that a performed transaction is always consistent

- ACID is a good fit for businesses which deal with online transaction processing (OLTP) or online analytical processing (OLAP)
  - These organizations need system which can handle many small transactions simultaneously
  - There must be zero tolerance for invalid states

ACID Model (Contd.)

- **Atomic** : Each transaction is either properly carried out or the process halts and the database reverts back to the state before the transaction started.
- **Consistent** : A processed transaction will never endanger the structural integrity of the database.
- **Isolated** : Transactions cannot compromise the integrity 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 cases of network or power outages. If a transaction fails, it will not impact the manipulated data.

ACID Model - Example use cases

- Financial institutions almost exclusively will use ACID databases.
  - Fund transfers depends on the atomic nature of ACID
  - An interrupted transaction which is not immediately removed from the database can cause lots of issues. Money could be debited from one account and due to an error never credited to another

ACID Model - Some compliant DBMSs

- Microsoft SQL Server
- MySQL
- PostgreSQL
- Oracle
- SQLite

- Some NoSQL DBMSs such as Apache CouchDB possess a certain degree of ACID compliance

Relational DB Pros.

- Simplicity of the model
- Ease of use
- Accuracy
  - Relational databases are strictly defined and well-organized, therefore data doesn't get duplicated
- Data Integrity
  - Relational databases provide consistency across all tables
- Normalization
  - Normalization is a method that breaks down information into manageable chunks to reduce storage size.
- Collaboration
  - Multiple users can access the database to retrieve information at the same time and even if data is being updated.

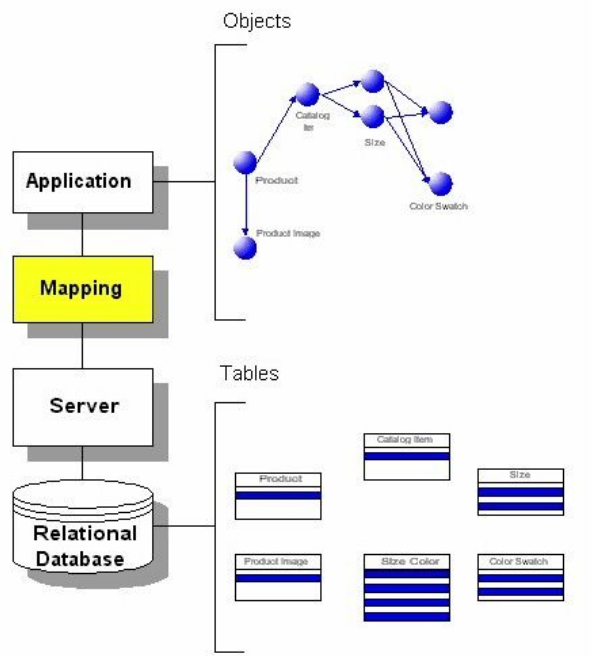
https://databasetown.com/relational-database-benefits-and-limitations/

Relational DB Cons

- Maintenance Problem
  - The maintenance of the relational database becomes difficult over time due to the increase in the data
- Physical Storage
  - A relational database is comprised of rows and columns, which requires a lot of physical memory because each operation performed depends on separate storage. The requirements of physical memory may increase along with the increase of data.
- Lack of scalability
  - While using the relational database over multiple servers, its structure changes and becomes difficult to handle. Due to the growing amounts of data, the data is not scalable on different physical storage servers
- Decrease in performance over time
  - When there is a large number of tables and data in the system, it causes an increase in complexity. It can lead to slow response times over queries or even complete failure for them depending on how many people are logged into the server at a given time.

Relational DB Cons (Contd.)

- Impedance mismatch (Complexity in structure)
  - Impedance mismatch occurs when you need to map objects used in an application to tables stored in a relational database.
  - Relational databases can only store data in tabular form which makes it difficult to represent complex relationships between objects.



https://www.service-architecture.com/articles/object-oriented-databases/impedance-mismatch.html

Raise of NoSQL

- Traditional RDBMS Strong consistency over availability under a partition

- Traditional RDBMS could not scale well with the growing amounts of big data
  - Tied to its data and query processing models
  - NoSQL gives up some of the RDBMS restrictions to achieve better performance

FAST. CHEAP. GOOD. PICK TWO

The basic idea is that if you pick two, the third option will be the opposite.

https://audioinsurgent.substack.com/p/fast-cheap-goodpick-two

Common Law of Business Balance

Common Law Of Business Balance

Quality, Speed, and Price. Choose two

BEACON MEDIA + MARKETING

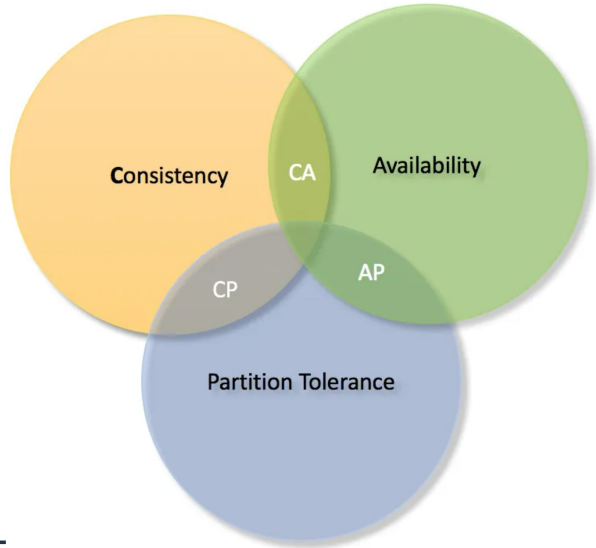
https://www.beaconmm.com/2023/05/31/the-beacon-way-episode-10-brand-messaging-part-3/

CAP Theorem

- How reliable can we build a distributed system?
- Formalizes the trade-offs you have to make between consistency and availability if your system ever suffered from partitions

CAP Theorem

- A distributed system can deliver only two out of three desired characteristics.
  - **Consistency** - Consistency means that all clients see the same data at the same time, no matter which node they connect to.
  - **Availability** - Availability means that any client making a request for data gets a response, even if one or more nodes are down.
  - **Partition Tolerance** - A partition is a communications break within a distributed system — the cluster must continue to work despite any number of communication breakdowns between nodes in the system.



CAP Theorem - History

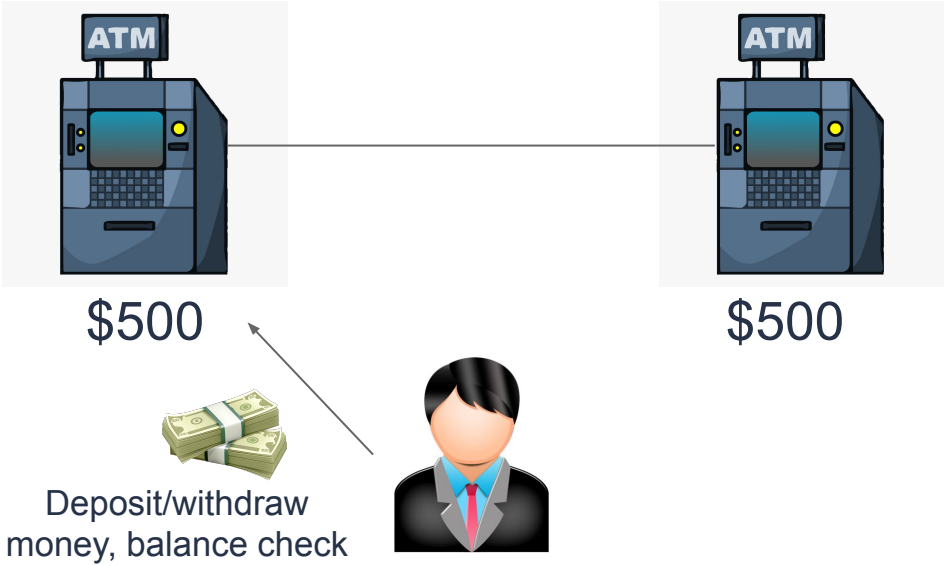
- The CAP theorem is also called Brewer's Theorem
- Professor Eric A. Brewer presented the CAP theorem during a talk he gave on distributed computing (at PODC conference) in 2000
- In 2002 MIT professors Seth Gilbert and Nancy Lynch published a proof of "Brewer's Conjecture."

Professor Eric A. Brewer

https://www.ibm.com/topics/cap-theorem

CAP Theorem - Consistency

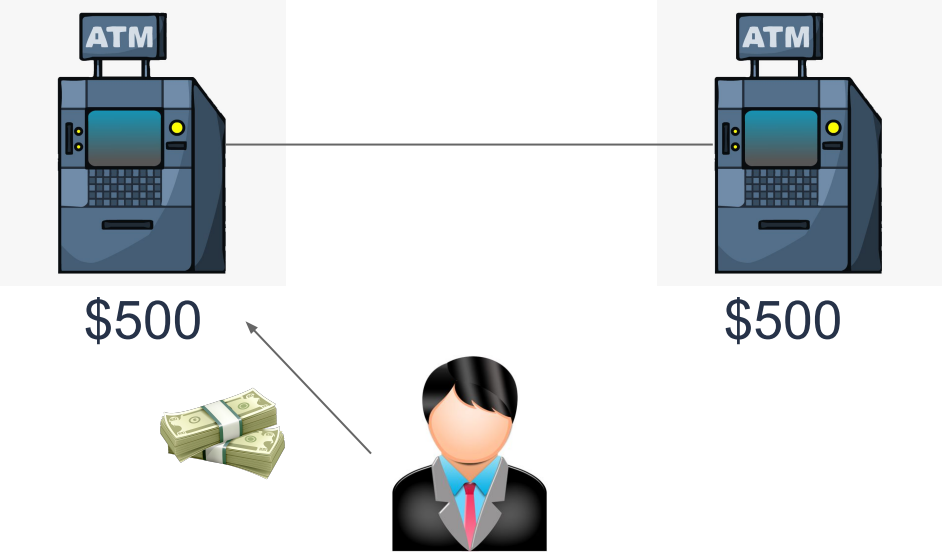
- Consistency means that all clients see the same data at the same time, no matter which node they connect to.
- In order for this to take place, every time data is written to one node, it must immediately be sent or duplicated to all of the other nodes in the system before the write can be considered to have been “successfully completed.”



- Store the account balance on the ATMs themselves
- Each ATM has a copy of the account balance
- When the customer deposits/withdraws money update the account balance figures on both the ATMs
- The two ATMs always have a consistent view of the account balance

CAP Theorem - Availability

- Availability means users can do whatever they want whenever they want
- Any client that makes a request for data will get a response, even if one or more of the nodes in the network are unavailable.

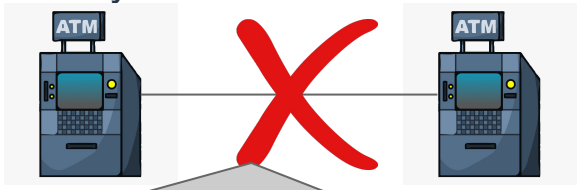


Each and every one of the operational nodes in the distributed system will, without fail, provide a legitimate answer to every request.

If there never are "Partitions" we can make the system both "Consistent" and "Available"

CAP Theorem - Partition Tolerance

- If one or more nodes goes out of sync, how well do they recover from that?
- the cluster's functionality must be maintained despite any number of failures in communication between the individual nodes that make up the system



Network Partition : a link between two nodes in the system that is lost or temporally delayed

Most complicated scenario

The system has to make a choice in its operation. It can be either "Consistent" or "Available" but not both.

CAP Theorem - Consistent Design

Consistent designs are simpler to understand, build, and use at the cost of availability.

- Consistent Design - Will not accepts withdrawal or deposit requests since it can't update the balance in the other ATM because its not safe to do so.
  - It will guarantee safety
  - But the users will be unhappy

CAP Theorem - Available Design

- Available Design - Will allow deposits/withdrawals and keep track of what happened. Later when the partition heals it will talk to other ATMs and exchange the state with them.
  - More available
  - But the balances maintained at the ATMs will be inconsistent

CAP Theorem - Partially Available Design 1

Partially available designs add more complexity to achieve higher availability

- Deposits - OK
- Withdrawals - No
- Balance information : Tentative (Not sure if the balance is correct)

CAP Theorem - Partially Available Design 2

- Deposits - OK
- Withdrawals - Small and rate limited
- Balance information : Tentative (Not sure if the balance is correct)

If the balance goes negative it will not go negative quickly.
If the balance goes negative we can hit them with huge fees (worst case sue them to get the money back)

CAP Theorem

- Sacrificing consistency is not the only way of increasing availability
  - Increase battery backups of ATMs
  - Armour Plated physical networks, redundant connections, so the network failures are less likely to happen
  - Test the software better so that there are less bugs resulting less system failures

CAP Theorem - Situations where trade off unnecessary

- Modern datacenter networks
  - Network does not fail very often
  - If all the machines are located in the same data center there is very less likely network failures
  - We need not care about the tradeoff. Have a very simple consistent system "Whenever a network failure happens our system goes down". This ever happens at all.

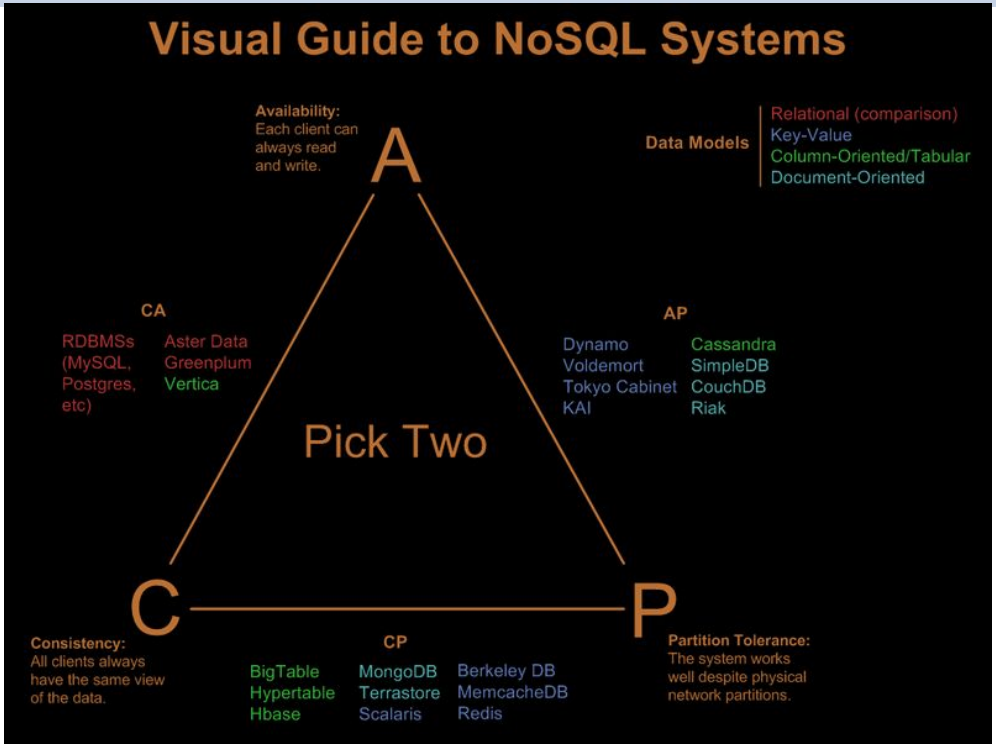
CAP Theorem - Situations where trade off unnecessary

- Disconnected offline operation
  - Google docs wants people to edit documents offline
  - Totally need to design a system that allows higher availability at the cost of consistency and figure out how to sync things backup later

SQL v NoSQL

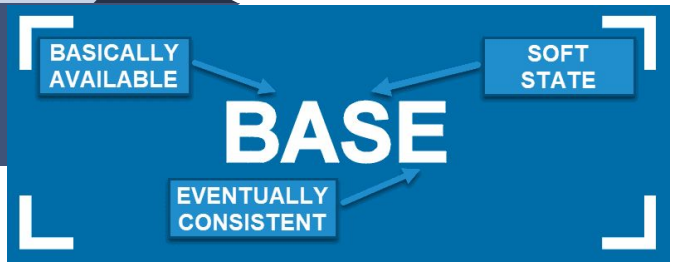
SQL	NoSQL
Relational	Non-relational
Vertically Scale	Horizontally Scale
Table based	Documents/Key value pairs/Graphs
SQL query language	Dynamic
Structured data	Unstructured (JSON)
Best for e-commerce platforms	Best for social media

CAP Theorem Applied on Different Databases



https://blog.nahurst.com/visual-guide-to-nosql-systems

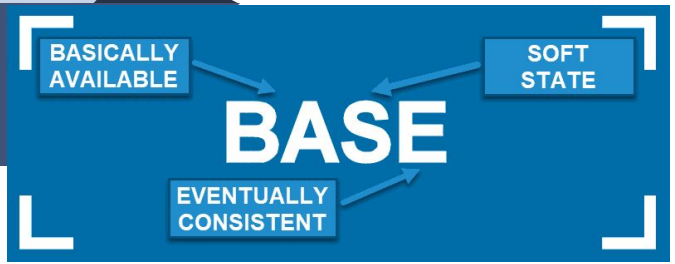
BASE Model



- BASE is a relatively new database model which reflects the characteristics of NoSQL databases
- Just as SQL databases are almost uniformly ACID compliant, NoSQL databases tend to conform to BASE principles
- Basically Available
  - 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.

https://phoenixnap.com/kb/acid-vs-base

BASE Model (Contd.)



- Soft State
  - Data values may change over time due to the lack of immediate consistency
  - Traditionally databases enforced their own consistency, but BASE model transferred that responsibility to application developers
- Eventually Consistent
  - The only requirement that NoSQL databases have regarding consistency is to require that at some point in the future, data will converge to a consistent state. No guarantees are made, however, about when this will occur.
  - The fact that BASE does not enforce immediate consistency does not mean that it never achieves it. However, until it does, data reads are still possible (even though they might not reflect the reality).

https://phoenixnap.com/kb/acid-vs-base

BASE Model - Use case Example

- Marketing and customer service companies who deal with sentiment analysis will prefer the elasticity of BASE when conducting their social network research.
  - Social network feeds are not well structured but contain huge amounts of data which a BASE-modeled database can easily store.

NoSQL Data Models

- Documents Oriented
- Column Oriented/Tabular
- Key-Value
- Network
- Geospatial

NoSQL Data Models - Documents Oriented

- A document database is a database that stores information in documents
- Also known as a document-oriented database or document store
- Type of general purpose databases which can be used in various industries and use cases
- Document databases are the most popular alternative to tabular and relational databases

Key Features of Documents Databases

- **Document model** : Documents map to objects in most popular programming languages, which allows developers to rapidly develop their applications.
- **Distributed and resilient** : Document databases are distributed, which allows for horizontal scaling (typically cheaper than vertical scaling) and data distribution. Document databases provide resiliency through replication.
- **Flexible schema** : Document databases have a flexible schema, meaning that not all documents in a collection need to have the same fields. The schema of the data can differ across documents, but these documents can still belong to the same collection
- **Querying through an API or query language**: Document databases have an API or query language that allows developers to execute the CRUD operations on the database.

Documents Oriented Databases - Features

- Documents stored are similar to each other but do not have to be exactly the same
- Document databases store documents in the value part of the key-value store
  - Document databases can be thought of as key-value stores where value is examinable

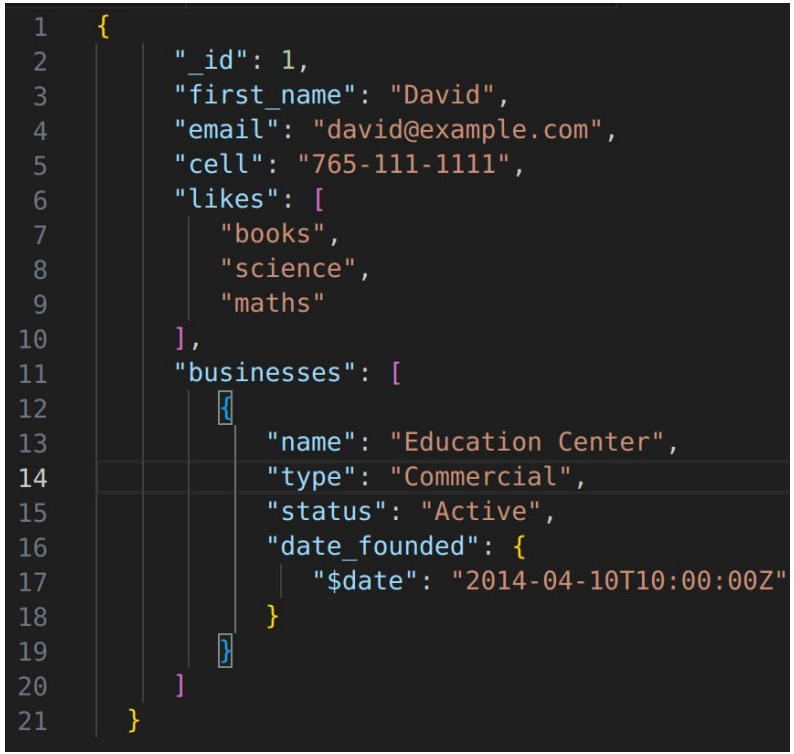
Documents Oriented Databases - Document

- A record in a document database
- Typically a document stores information on one object as well as any of its related metadata
- A document stores data in field-value pairs.
- The values can be a variety of types and structures such as,
  - Strings
  - Numbers
  - Dates
  - Arrays
  - Objects
- Can be stored in formats such as JSON, BSON, and XML

Documents Oriented Databases - Document (Contd.)

- In documents there are no empty attributes
  - If a given attribute was not found that implies that it was not set or irrelevant to the document
- Documents allow for new attributes to be created without the need to define them or to change the existing documents.

Documents Oriented Databases - Document - Example 1



```
{
    "_id": 1,
    "first_name": "David",
    "email": "david@example.com",
    "cell": "765-111-1111",
    "likes": [
        "books",
        "science",
        "maths"
    ],
    "businesses": [
        {
            "name": "Education Center",
            "type": "Commercial",
            "status": "Active",
            "date_founded": {
                "$date": "2014-04-10T10:00:00Z"
            }
        }
    ]
}
```

A sample document which stores information of a person named David.

Documents Oriented Databases - Collections

- A collection is a group of documents
- In general a collection stores documents that have similar contents
- Since document databases have a flexible schema not all documents in a collection are required to have the same fields
  - Some databases provide schema validation so that the schema can be optionally locked down when needed

Documents Oriented Databases - Document - Example 2

```
{
    "_id": 2,
    "first_name": "Donna",
    "email": "donna@example.com",
    "spouse": "Joe",
    "likes": [
        "spas",
        "shopping",
        "live tweeting"
    ],
    "businesses": [
        {
            "name": "Castle Realty",
            "status": "Thriving",
            "date_founded": {
                "$date": "2013-11-21T04:00:00Z"
            }
        }
    ]
}
```

- A sample document which stores information of a person named Donna.
- Note that the document for Donna does not contain the same fields as the document for David.
- The users collection is leveraging a flexible schema to store the information that exists for each user.

Documents Oriented Databases - CRUD Operations

- Document oriented databases typically have an API or a query language which allows developers to execute the CRUD operations
  - **Create** - Documents can be created in the database with each having a unique identifier.
  - **Read** - Documents can be read from the database. Indexes can be added to the database in order to increase read performance.
  - **Update** - Existing documents can be updated — either in whole or in part.
  - **Delete** - Documents can be deleted from the database.

Document Databases vs Relational Databases

- **The intuitiveness of the data model:**
  - Documents map to the objects in code, so they are much more natural to work with.
  - There is no need to decompose data across tables, run expensive joins, or integrate a separate Object Relational Mapping (ORM) layer.
  - Data that is accessed together is stored together, so developers have less code to write and end users get higher performance.

Document Databases vs Relational Databases

- **The ubiquity of JSON documents:**
  - JSON has become an established standard for data interchange and storage.
  - JSON documents are lightweight, language-independent, and human-readable.
  - Documents are a superset of all other data models so developers can structure data in the way their applications need — rich objects, key-value pairs, tables, geospatial and time-series data, or the nodes and edges of a graph.

Document Databases vs Relational Databases

- **The flexibility of the schema:**
  - A document's schema is dynamic and self-describing, so developers don't need to first pre-define it in the database.
  - Fields can vary from document to document.
  - Developers can modify the structure at any time, avoiding disruptive schema migrations.

Document Databases vs Tables

- Working with data in documents is more intuitive than working with data in tables.
- Developers need not worry about manual splitting of related data across multiple tables when storing or joining it back when accessing it
- No need to use an Object Relational Mapper (ORM) to handle manipulating the data for them.
- Can easily work with the data directly in their applications.

Document Databases vs Tables (Contd.)

```
{
    "_id": 1,
    "first_name": "David",
    "email": "david@example.com",
    "cell": "765-111-1111",
    "likes": [
        "books",
        "science",
        "maths"
    ],
    "businesses": [
        {
            "name": "Education Center",
            "type": "Commercial",
            "status": "Active",
            "date_founded": {
                "$date": "2014-04-10T10:00:00Z"
            }
        }
    ]
}
```

users

ID	first_name	email	cell
1	David	david@example.com	765-111-1111

likes

ID	user_id	like
1	1	books
2	1	science
3	1	maths

business

ID	user_id	name	type	status	date_founded
10	1	Education Center	commercial	Active	2014-04-10T10:00:00Z

Document Databases vs Tables (Contd.)

- Information on a user can be stored in a single document in a document database or three tables in a relational database.
- Retrieving or updating information about a user in the database is straightforward, and modeling the data in the database is intuitive

Relationship between document databases and other databases

- Document model is a superset of other data models
- **Key-value pairs** : Can be modeled with fields and values in a document. Any field in a document can be indexed providing additional flexibility in how to query the data.
- **Relational data** :
  - keep related data together in a single document using embedded documents and arrays.
  - Related data can also be stored in separate documents

Relationship between document databases and other databases (Contd.)

- Documents map to objects in most popular programming languages.
- Graph nodes and/or edges can be modeled as documents.
- Geospatial data can be modeled as arrays in documents.

Document Oriented Database Pros.

- A flexible schema which allows the data model to evolve as application needs change.
- Ability to scale horizontally
- Document model is an intuitive data model
  - Easy and fast for developers to work with
- Document databases have rich APIs and query languages that allow developers to easily interact with their data.
- Document databases are distributed (allowing for horizontal scaling as well as global data distribution) and resilient

Document Oriented Database Cons.

- Many of the document databases do not support multi-document ACID transactions (80-90% use cases involving document databases do not require this feature)
  - There are document databases which support this such as RavenDB, MongoDB
- Queries against varying aggregate structure
  - Flexible schema means that the database does not enforce any restrictions on the schema. Data is saved in the form of application entities. If you need to query these entities ad hoc, your queries will be changing
  - Since the data is saved as an aggregate, if the design of the aggregate is constantly changing, you need to save the aggregates at the lowest level of granularity—basically, you need to normalize the data

Document Databases - Use cases

- Single view or data hub
- Customer data management and personalization
- Internet of Things (IoT) and time-series data
- Product catalogs and content management
- Payment processing
- Mobile apps
- Mainframe offload
- Operational analytics
- Real-time analytics
- Event logging
- Blogging platforms
- Content management systems (CMS)

Document Databases - Popular examples

- MongoDB
- CouchDB
- Terrastore
- OrientDB
- RavenDB

Column-Oriented Databases

- Also known as Column-stores
- Each attribute of a table is stored in a separate file or region on storage.
- Allows to store data with keys mapped to values and the values grouped into multiple column families, each column family being a map of data

Column-Oriented Databases (Contd.)

- Column-oriented database have increasing interest in recent times with analytic queries that perform scans and aggregates over large portions of a few columns of a table.
- Main advantage of column store is that it can access just the columns needed to answer the analytic queries.
- Database system performance is directly related to the efficiency of the system at storing data on primary storage (e.g., disk) and moving it into CPU registers for processing.

Column-Oriented Databases (Contd.)

- Column-oriented database have increasing interest in recent times with analytic queries that perform scans and aggregates over large portions of a few columns of a table.
- Main advantage of column store is that it can access just the columns needed to answer the analytic queries.
- Database system performance is directly related to the efficiency of the system at storing data on primary storage (e.g., disk) and moving it into CPU registers for processing.
  - Various research done on topics of indexing, materialized views, vertical and horizontal partitioning.

Column-Oriented Databases (Contd.)

- Early influential efforts,
  - Academic
    - MonetDB
    - VectorWise
    - C-Store
  - Commercial
    - SybaseIQ
    - Ingres VectorWise
    - Vertica

Column-Oriented Databases (Contd.)

- Column-oriented databases completely vertically partition a database into a collection of individual columns that are stored separately.

- Storing each column separately on disk, these column-based systems enable queries to read just the attributes they need, rather than having to read entire rows from disk and discard unneeded attributes once they are in memory.
  - A similar benefits happens while transferring data from main memory to CPU registers improving the overall utilization of the available I/O and memory bandwidth.

Column-Oriented Databases - When not to use?

- Applications requiring ACID transactions
- Aggregation of the data using queries
  - SUM, AVG, etc.
  - Needs to be done at the client side instead
- For early prototypes
  - When it is unclear what kind of query pattern will be present in the application it's wise not to use Column stores
  - As the query pattern changes, we have to change the design of the column family design

Column-Oriented Databases - Popular examples

- Google's BigTable
- HBASE
- Cassandra
- HyperTable
- SimpleDB

Google’s Bigtable

- Google’s Bigtable is really the parent of the modern columnar databases.
- There are a few published papers on its design, and each of the other columnar databases discussed are implementations that closely follow Bigtable’s design

Apache Cassandra - Introduction

- Developed at Facebook
  - Initial release: 2008
  - Stable release: 2013
    - Apache Licence
- Written in: Java
- OS: cross-platform
- Operations:
  - CQL (Cassandra Query Language)
  - MapReduce support
    - Can cooperate with Hadoop (data storage instead of HDFS)

Apache Cassandra - in 50 Words or less

Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tuneably consistent, column-oriented 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.

Eben Hewitt, Cassandra, The Definitive Guide, O'Reilly

Apache Cassandra - Introduction (Contd.)

- Cassandra provides eventual (weak) consistency, Availability, Partition-tolerance

- Some of the myriad users

Apache Cassandra - Terminology

RDBMS	Cassandra
database instance	cluster
database	keyspace (usually one per application)
table	column family
row	row
column (same for all rows)	column (can be different per row)

Apache Cassandra - Terminology

- **Column** - This is the basic unit, consists of a name-value pair
  - Name serves as a key
  - Stored with a timestamp (expired data, resolving conflicts, …)
- **Row** - This is a collection of columns attached or linked to a key
  - Columns can be added to any row at any time without having to add it to other rows
- **Column family** - A collection of similar rows
  - Rows do not have the same columns

Apache Cassandra - Sample

docker run --name cassandra -d cassandra:4.1

docker exec -it cassandra bash

https://hub.docker.com/_/cassandra/#!

Apache Cassandra - Sample

docker run --name cassandra -d cassandra:4.1

docker exec -it cassandra bash

https://hub.docker.com/_/cassandra/#!

Apache Cassandra - Sample (Contd.)

```
CREATE KEYSPACE Excelsior
WITH replication = {'class': 'SimpleStrategy',
'replication_factor' : 1};

ALTER KEYSPACE Excelsior2
WITH replication = {'class': 'SimpleStrategy',
'replication_factor' : 0};

DROP KEYSPACE Excelsior2;
```

- Data Model - Example

{ name: "firstName", value: "Martin", timestamp: 12345667890 }

- Column key of firstName and the value of Martin

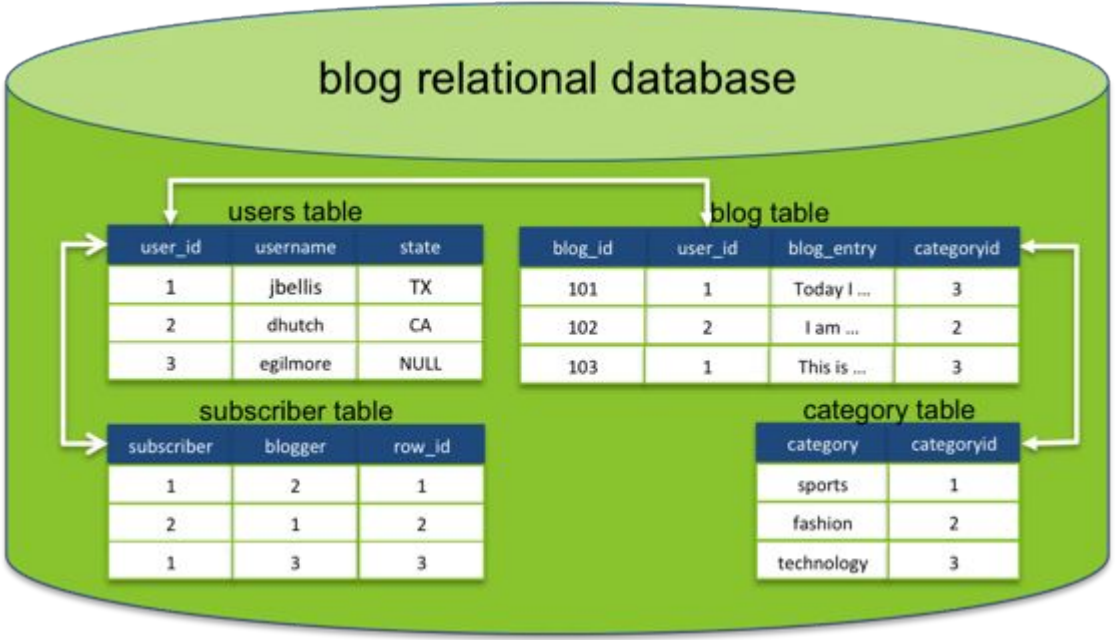
{ "pramod-sadalage" : {
firstName: "Pramod",
lastName: "Sadalage",
lastVisit: "2012/12/12" }
"martin-fowler" : {
firstName: "Martin",
lastName: "Fowler",
location: "Boston" } }

pramod-sadalage row and martin-fowler row with different columns;
both rows are a part of a column family

Cassandra Column-families vs. Relations

- Do not need to model all of the columns up front
  - Each row is not required to have the same set of columns
  - Usually we assume similar sets of columns
    - Related data
    - Can be extended when needed
- No formal foreign keys
  - Joining column families at query time is not supported
  - need to pre-compute the query / use a secondary index

Cassandra Column-families vs. Relations



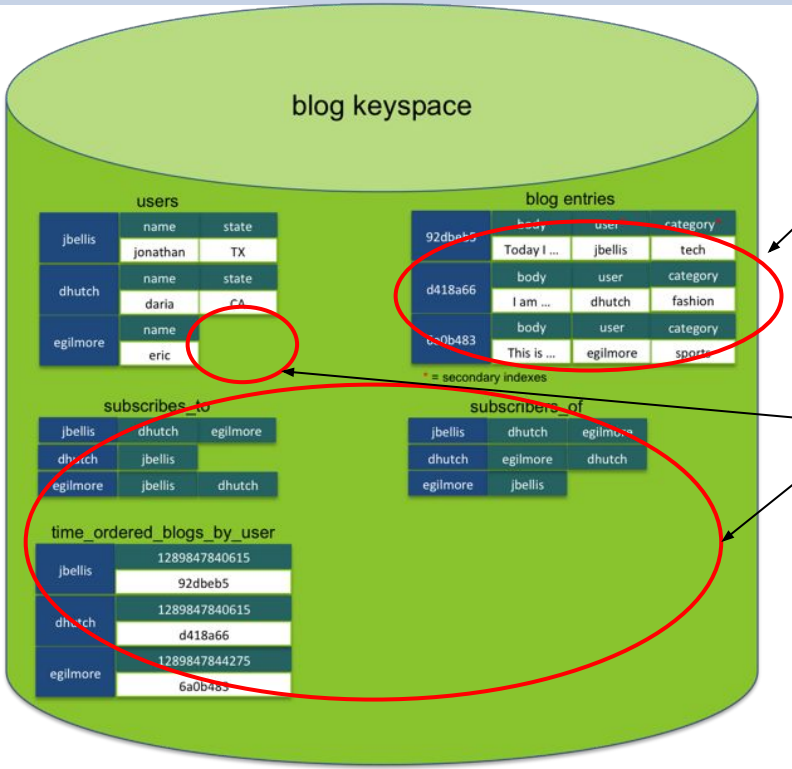
- Data stored in tables
- Schema-based (Structured tables)
- Queried using SQL

Typical SQL query

```
SELECT user_id
from users WHERE username = “jbellis”
```

source

Cassandra Column-families vs. Relations



- Like SQL tables
- but may be unstructured (client-specified)
- Can have index tables

- No schemas
- Some columns missing from some entries
- “Not Only SQL”
- Supports get(key) and put(key, value) operations
- Often write-heavy workloads

Cassandra Column families

- Can define metadata about columns
  - Actual columns of a row are determined by client application
  - Each row can have a different set of columns
- **Static** – similar to a relational database table
  - Rows have the same set of columns
  - Not required to have all of the columns defined
- **Dynamic** – takes advantage of Cassandra's ability to use arbitrary application-supplied column names
  - Pre-computed result sets
  - Stored in a single row for efficient data retrieval
  - Row = a snapshot of data that satisfy a given query
    - Like a materialized view

Cassandra Column families

static

row key	columns …			
jbellis	name	email	address	state
	jonathan	jb@ds.com	123 main	TX
dhutch	name	email	address	state
	daria	dh@ds.com	45 2nd St.	CA
egilmore	name	email		
	eric	eg@ds.com		

dynamic

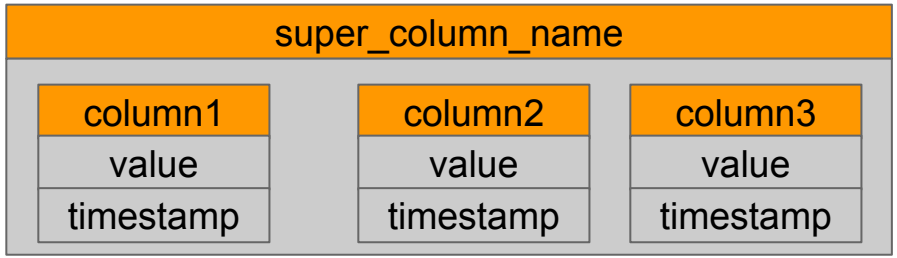
row key	columns …			
jbellis	dhutch	egilmore	datastax	mzcassie
dhutch	egilmore			
egilmore	datastax	mzcassie		

Cassandra Columns

- Column is the smallest increment of data
  - Name + value + timestamp
  - Value can be empty (e.g., materialized views)
- Can be indexed on their name
  - Using a secondary index
  - Primary index = row key
    - Ensure uniqueness, speeds up access, can influence storage order
- Types:
  - Expiring - with optional expiration date called TTL
    - Can be queried
  - Counter – to store a number that incrementally counts the occurrences of a particular event or process
    - E.g., to count the number of times a page is viewed
    - Operation increment/decrement with a specified value
    - Internally ensures consistency across all replicas
  - Super – add another level of nesting
    - To group multiple columns based on a common lookup value

Cassandra Super Columns

```
{name: "book:978-0767905923",
value: { author: "Mitch Albon",
title: "Tuesdays with Morrie",
isbn: "978-0767905923" } }
```



- Super column - a column consisting of a map of columns
  - It has a name and value involving the map of columns
- super column family – a column family consisting of super columns

Cassandra Column Family Definition

- A key must be specified
- Data types for columns can be specified
- Options can be specified

```
CREATE COLUMNFAMILY Fish (key blob PRIMARY KEY);
CREATE COLUMNFAMILY FastFoodEatings (user text PRIMARY KEY)
WITH comparator=timestamp AND default_validation=int;
CREATE COLUMNFAMILY MonkeyTypes (
KEY uuid PRIMARY KEY,
species text,
alias text,
population varint
) WITH comment='Important biological records'
AND read_repair_chance = 1.0;
```

Cassandra Column Families

- Comparator : data type for a column name
- Validator : data type for a column (or row key) value
- Data types do not need to be defined
  - Default: ByteType which represents arbitrary hexadecimal bytes
- Basic operations : GET, SET, DEL
- New versions of Cassandra and CQL follow new strategy but the capabilities remain the same
  - Still we can create tables with arbitrary columns

Cassandra Column Families – Example

```
CREATE COLUMNFAMILY users
with key_validation_class = 'UTF8Type'
and comparator = 'UTF8Type'
and column_metadata = [
{column_name : 'name', validation_class : UTF8Type},
{column_name : 'birth_year', validation_class : Int32Type}];
SET users['jbellis']['name'] = 'Jonathan Ellis';
SET users['jbellis']['birth_year'] = 1976;
SET users['jbellis']['home'] = long(1112223333);
SET users['jbellis']['work'] = long(2223334444);
GET users['jbellis'];
GET users['jbellis']['home'];
DEL users['jbellis']['home'];
DEL users['jbellis'];
```

Column Families - Best Practice

- Validators
  - Define a default row key validator using property key_validation_class
  - Static column families define each column and its associated type
  - Dynamic column families
    - Column names are not known ahead
    - Specify default_validation_class

Column Families - Best Practice

- Comparators
  - Within a row, columns are stored in sorted order by their column name
  - Static column families
    - Typically Strings
    - Order is unimportant
  - Dynamic column families
    - Order is usually important (e.g., timestamps)

Cassandra Query Language (CQL)

- A query language for Cassandra with SQL-like commands
  - CREATE, ALTER, UPDATE, DROP, DELETE, TRUNCATE, INSERT, …
- Much simpler than SQL
  - WHERE clauses are simple
  - Does not allow joins or subqueries
- More general. different approach than column families
  - Closer to key/value and document databases

CQL Data Types

ascii	ASCII character string
bigint	64-bit signed long
blob	Arbitrary bytes (no validation)
boolean	true or false
counter	Counter column (64-bit long)
decimal	Variable-precision decimal
double	64-bit IEEE-754 floating point

float	32-bit IEEE-754 floating point
int	32-bit signed int
text	UTF8 encoded string
timestamp	A timestamp
uuid	Type 1 or type 4 UUID
varchar	UTF8 encoded string
varint	Arbitrary-precision integer

Cassandra - Key Space

- Create a key space with a specified replication strategy and parameters

```
CREATE KEYSPACE Excelsior
WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};
```

- Set a keyspace as the current working keyspace

```
USE Excelsior;
```

- Alter the properties of an existing keyspace

```
ALTER KEYSPACE Excelsior
WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 4};
```

- Drop a keyspace

```
DROP KEYSPACE Excelsior;
```

Cassandra Working with a Table - Primary Key

- Creating a table with name, columns and other options

```
CREATE TABLE timeline (
userid uuid,
posted_month int,
posted_time uuid,
body text,
posted_by text,
PRIMARY KEY (userid, posted_month, posted_time) )
WITH compaction = { 'class' : 'LeveledCompactionStrategy' };
```

- Primary key is compulsory
  - Partition key: The first column (or set of columns if parenthesized)
    - Records are stored on the same node
  - Clustering columns
    - Determine per-partition clustering, i.e., the order for physical storing rows

Cassandra Working with a Table - Column Expiration

- When the data will expire

```
CREATE TABLE excelsior.clicks (
userid uuid,
url text,
date timestamp,
name text,
PRIMARY KEY (userid, url) );
INSERT INTO excelsior.clicks (userid, url, date, name)
VALUES (3715e600-2eb0-11e2-81c1-0800200c9a66,
'http://apache.org', '2013-10-09', 'Mary')
USING TTL 86400;
```

- Determine how much longer the data has to live

```
SELECT TTL (name) from excelsior.clicks
WHERE url = 'http://apache.org' ALLOW FILTERING;
```

Cassandra Working with a Table - Collections

- Collection types
  - set – unordered unique values
    - Returned in alphabetical order, when queried
  - list – ordered list of elements
    - Can store the same value multiple times
    - Returned sorted according to index value in the list
  - map – name + value pairs
    - Each element is internally stored as one Cassandra column
    - Each element can have an individual time-to-live

Cassandra
Working with a Table – Set

```
CREATE TABLE users (
user_id text PRIMARY KEY,
first_name text,
last_name text,
emails set<text> );

INSERT INTO users (user_id, first_name, last_name, emails)
VALUES('frodo', 'Frodo', 'Baggins', {'f@baggins.com', 'baggins@gmail.com'});

UPDATE users SET emails = emails + {'fb@friendsofmordor.org'}
WHERE user_id = 'frodo';

SELECT user_id, emails FROM users WHERE user_id = 'frodo';

UPDATE users SET emails = emails – {'fb@friendsofmordor.org'}
WHERE user_id = 'frodo';

UPDATE users SET emails = {} WHERE user_id = 'frodo';
```

Cassandra
Working with a Table – List

```
ALTER TABLE users ADD todo map<timestamp, text>;

UPDATE users SET todo = { '2012-9-24' : 'enter mordor',
'2012-10-2 12:00' : 'throw ring into mount doom' }
WHERE user_id = 'frodo';

UPDATE users SET todo['2012-10-2 12:00'] =
'throw my precious into mount doom‘
WHERE user_id = 'frodo';

INSERT INTO users (todo) VALUES ( {
'2013-9-22 12:01' : 'birthday wishes to Bilbo',
'2013-10-1 18:00' : 'Check into Inn of Prancing Pony' });

DELETE todo['2012-9-24'] FROM users
WHERE user_id = 'frodo';
```

Cassandra
Working with a Table

- Delete a table including all data

```
DROP TABLE timeline;
```

- Remove all data from a table

```
TRUNCATE timeline;
```

- Create a (secondary) index
- Allow efficient querying of other columns than key

```
CREATE INDEX userIndex ON timeline (posted_by);
```

- Drop an index

```
DROP INDEX userIndex;
```

Cassandra Querying

- Simplified queries without joins
- Filtering is done with (WHERE) Clause

```
SELECT * FROM users
WHERE first_name = 'jane' and last_name='smith';
```

- Ordering is done with (ORDER BY) Clause

```
SELECT * FROM emp
WHERE empID IN (130,104)
ORDER BY deptID DESC;
```

Cassandra Querying (Contd.)

SELECT select_expression
FROM keyspace_name.table_name
WHERE relation **AND** relation ...
ORDER BY (clustering_key (ASC | DESC)...)
LIMIT n
ALLOW FILTERING

- select_expression
  - List of columns
  - DISTINCT
  - COUNT
  - Aliases (AS)
  - TTL (column_name)
  - WRITETIME(column_name)

Cassandra Querying (Contd.)

- relation:
  - column_name (= | < | > | <= | >=) key_value
  - column_name IN ((key_value,...))
  - TOKEN (column_name, ...) (= | < | > | <= | >=)
  - (term | TOKEN (term, ...))
- term:
  - set/list/map
  - constant

Cassandra
Querying – ALLOW FILTERING

- Two types of filtering queries
  - Non-filtering queries
    - Have predictable performance
    - Queries where it is know that all records read will be returned (maybe partly) in the result set
  - Filtering queries
    - Attempting a potentially expensive query may result in Bad Request error
    - Use the ALLOW FILTERING to denote that "We know what we are doing"
    - Usually used together with LIMIT n

Cassandra
Querying – ALLOW FILTERING

```
CREATE TABLE users (
username text PRIMARY KEY,
firstname text,
lastname text,
birth_year int,
country text
);
CREATE INDEX ON users(birth_year);

SELECT * FROM users;

SELECT firstname, lastname FROM users
WHERE birth_year = 1981;
```

Query performance is proportional to the amount of data returned

Cassandra
Querying – ALLOW FILTERING (Contd.)

```
SELECT firstname, lastname
FROM users
WHERE birth_year = 1981 AND country = 'FR';
```

There is no guarantee that Cassandra won't have to scan large amount of data even if the result is small

```
SELECT firstname, lastname
FROM users
WHERE birth_year = 1981 AND country = 'FR'
ALLOW FILTERING;
```

Thank you!