NoSQL Databases

Overview

- Relational databases and the need for NoSQL
- Common characteristics of NoSQL databases
- Types of NoSQL database
  - Key-value
  - Document
  - Column-family
  - Graph
- NoSQL and consistency
- The CAP theorem

Resources for this section

- NoSQL Distilled (Sadalage & Fowler, 2013)
    - Key points from this book are summarised here:
        - http://martinfowler.com/articles/nosqlKeyPoints.html
- A 55-minute introduction to NoSQL by Martin Fowler is available to view here:
    - http://www.youtube.com/watch?v=qI_g07C_Q5I

Why is NoSQL needed?

- Relational databases have been a successful technology for twenty years, providing persistence, concurrency control, and an integration mechanism

- However, application developers have been frustrated with the **impedance mismatch** between the relational model and the in-memory data structures

    - Impedance mismatch: the effort required to split up an object (e.g. an order) into separate tables when storing it in a relational database (e.g. an order might have elements in the product table, customer table, order line table etc.) only to have to then put it back together again (using joins)

Why is NoSQL needed?

- The vital factor for a change in data storage was the need to support large volumes of data by running on clusters. Relational databases are not designed to run efficiently on clusters.

- NoSQL is an accidental neologism. There is no prescriptive definition—all you can make is an observation of common characteristics.

NoSQL common characteristics

- The common characteristics of NoSQL databases are:
  - Not using the relational model
  - Running well on clusters
  - Open-source
  - Built for the 21st century web
  - Schemaless
    - No explicit schema is defined; however, the implicit schema of a NoSQL database must still be managed in some way

NoSQL vs. SQL Comparison

	SQL Databases	NoSQL Databases
Types	One type (SQL database) with minor variations	Many different types including key-value stores, document databases, wide-column stores, and graph databases
Development History	Developed in 1970s to deal with first wave of data storage applications	Developed in 2000s to deal with limitations of SQL databases, particularly concerning scale, replication and unstructured data storage
Examples	MySQL, Postgres, Oracle Database	MongoDB, Cassandra, HBase, Neo4j

NoSQL vs. SQL Comparison

	SQL Databases	NoSQL Databases
Data Storage Model	Individual records (e.g., "employees") are stored as rows in tables, with each column storing a specific piece of data about that record (e.g., "manager," "date hired," etc.), much like a spreadsheet. Separate data types are stored in separate tables, and then joined together when more complex queries are executed. For example, "offices" might be stored in one table, and "employees" in another. When a user wants to find the work address of an employee, the database engine joins the "employee" and "office" tables together to get all the information necessary.	Varies based on NoSQL database type. For example, key-value stores function similarly to SQL databases, but have only two columns ("key" and "value"), with more complex information sometimes stored within the "value" columns. Document databases do away with the table-and-row model altogether, storing all relevant data together in single "document" in JSON, XML, or another format, which can nest values hierarchically.

NoSQL vs. SQL Comparison

	SQL Databases	NoSQL Databases
Schemas	Structure and data types are fixed in advance. To store information about a new data item, the entire database must be altered, during which time the database must be taken offline.	Typically dynamic. Records can add new information on the fly, and unlike SQL table rows, dissimilar data can be stored together as necessary. For some databases (e.g., wide-column stores), it is somewhat more challenging to add new fields dynamically.
Scaling	Vertically, meaning a single server must be made increasingly powerful in order to deal with increased demand. It is possible to spread SQL databases over many servers, but significant additional engineering is generally required.	Horizontally, meaning that to add capacity, a database administrator can simply add more commodity servers or cloud instances. The NoSQL database automatically spreads data across servers as necessary

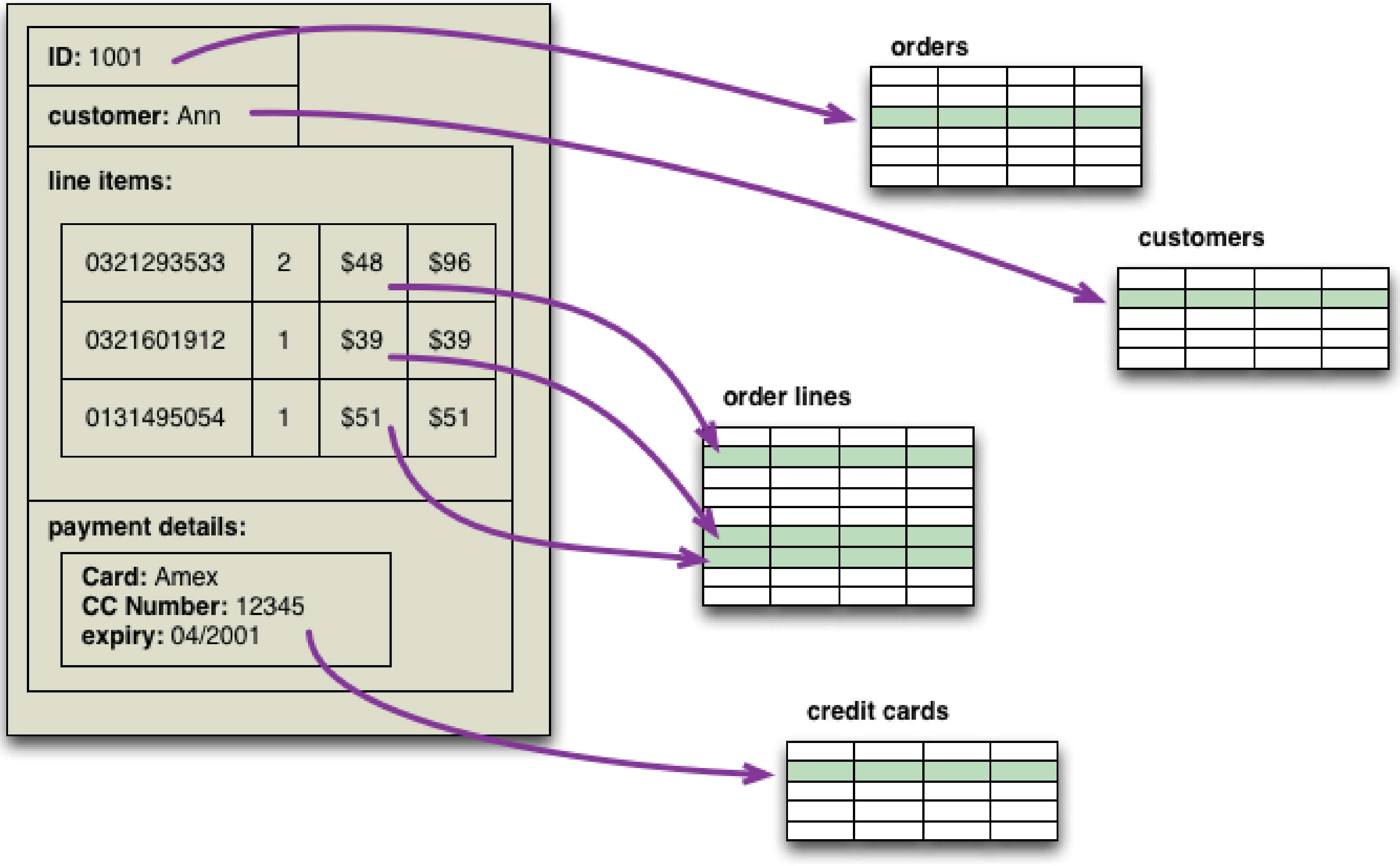
NoSQL vs. SQL Comparison

	SQL Databases	NoSQL Databases
Development Model	Mix of open-source (e.g., Postgres, MySQL) and closed source (e.g., Oracle Database)	Open-source
Supports Transactions	Yes, updates can be configured to complete entirely or not at all	In certain circumstances and at certain levels (e.g., document level vs. database level)
Data Manipulation	Specific language using Select, Insert, and Update statements, e.g. SELECT fields FROM table WHERE…	Through object-oriented APIs
Consistency	Can be configured for strong consistency	Depends on product. Some provide strong consistency (e.g., MongoDB) whereas others offer eventual consistency (e.g., Cassandra)

Four types of NoSQL data model

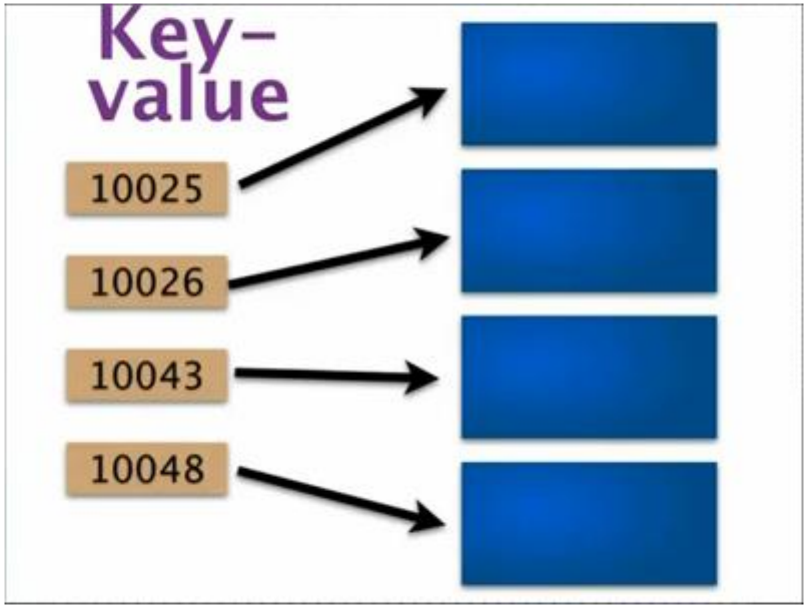
- Key-value
- Document
- Column-family
- Graph

Relational database: order example



Key-value

- Both a key and a value are stored – in the case of the order example, everything to do with that one order is simply stored as one value



Key-value

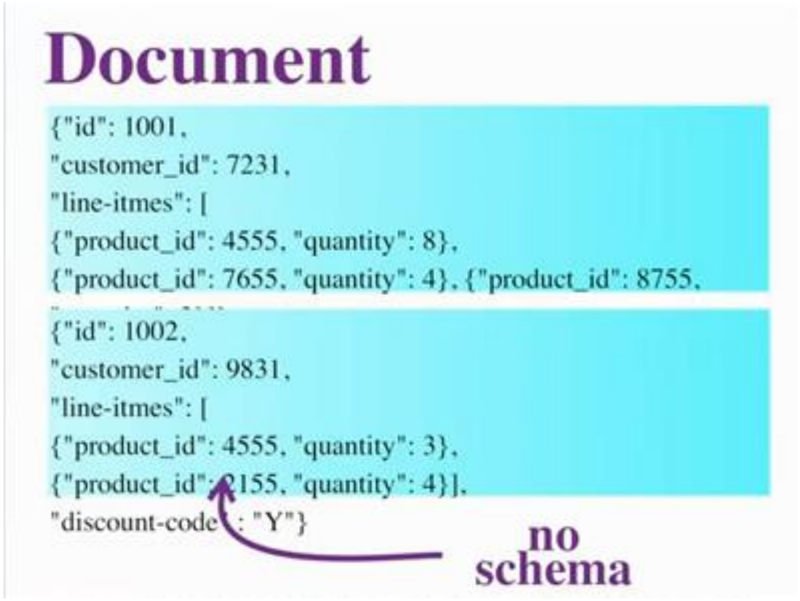
- A key value store is primarily used when all access to the database is via the primary key
- The Key/value model is the simplest and easiest to implement. However, it is inefficient when you are only interested in querying or updating part of a value.

Typical applications:	Storing session information, user profiles/preferences, shopping cart data etc.
Data model:	Collection of key-value pairs
Strengths:	Fast lookups
Weaknesses:	Stored data has no schema

Document

- Data is stored in documents – in the order example, everything to do with one order is stored together in one document

- Data entries are labelled (e.g. customer id, quantity)



Document

```
{"id": 1001,
"customer_id": 7231,
"line-itmes": [
{"product_id": 4555, "quantity": 8},
{"product_id": 7655, "quantity": 4}, {"product_id": 8755,
[illegible]
{"id": 1002,
"customer_id": 9831,
"line-itmes": [
{"product_id": 4555, "quantity": 3},
{"product_id": 2155, "quantity": 4}],
"discount-code": "Y"}
```

no schema

Document

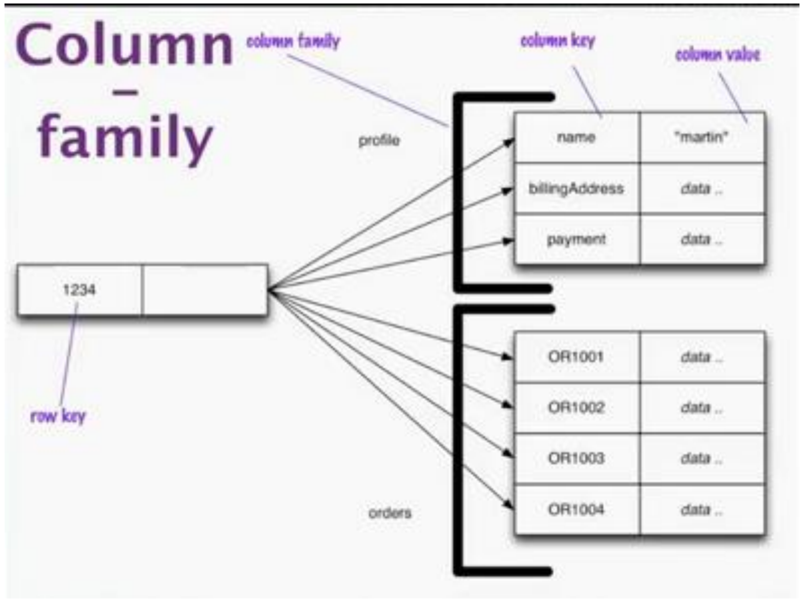
- Documents are self-describing, hierarchical tree structures which allow nested values associated with each key.
- Document databases support querying more efficiently.

Typical applications:	Event logging, content management systems, web analytics or e-commerce applications
Data model:	Documents (with multiple values in a document)
Strengths:	Tolerant of incomplete data
Weaknesses:	Query performance, no standard query syntax

Column-family

- Data is stored with keys that are linked to groups of column (attributes) – for example, a group or column family that stores customer details, another that stores orders for that customer, etc.

- Everything about one order is stored together in this one group of columns



Column-family

- Column family stores allow you to store data with keys mapped to values and the value grouped into multiple column families, each column family being a map of data.

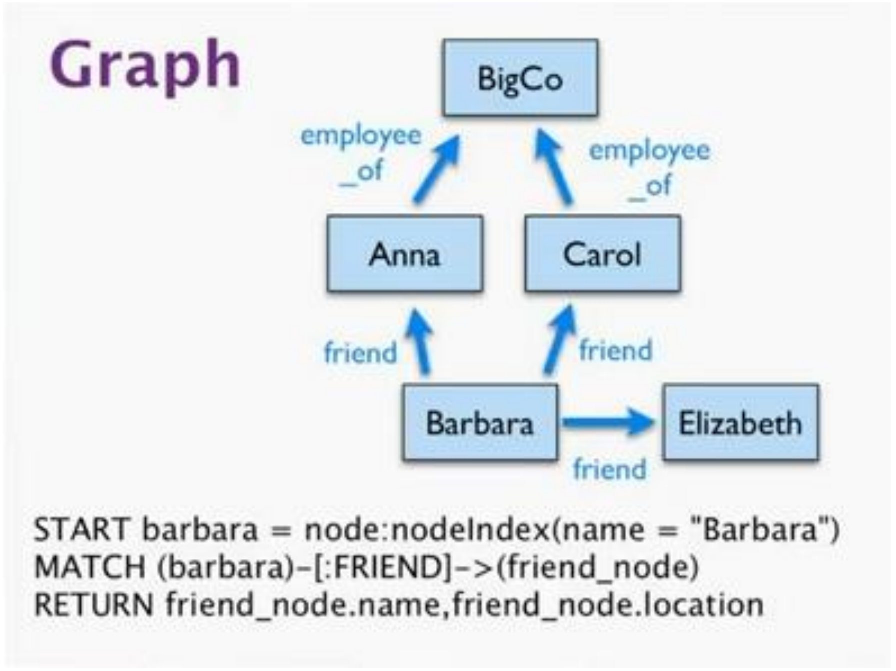
Typical applications:	Event logging, content management systems, blogging platforms, counters
Data model:	Columns – column families
Strengths:	Fast lookups, good distributed storage of data
Weaknesses:	Performing different types of queries can be complex

Aggregate-oriented databases

- Key-value, document, and column-family databases can all be seen as forms of aggregate-oriented database.
- Aggregates make it easier for the database to manage data storage over clusters.
- Aggregate-oriented databases work best when most data interaction is done with the same aggregate
  - i.e. when we always want to view orders as orders – changing the way data is viewed e.g. querying this data to find out which products are selling best is much more difficult than with a relational database

Graph databases

- Work differently than the other, aggregate-oriented types of NoSQL database
- Graph databases organize data into node and edge graphs; they work best for data that has complex relationship structures



```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[:FRIEND]->(friend_node)
RETURN friend_node.name,friend_node.location
```

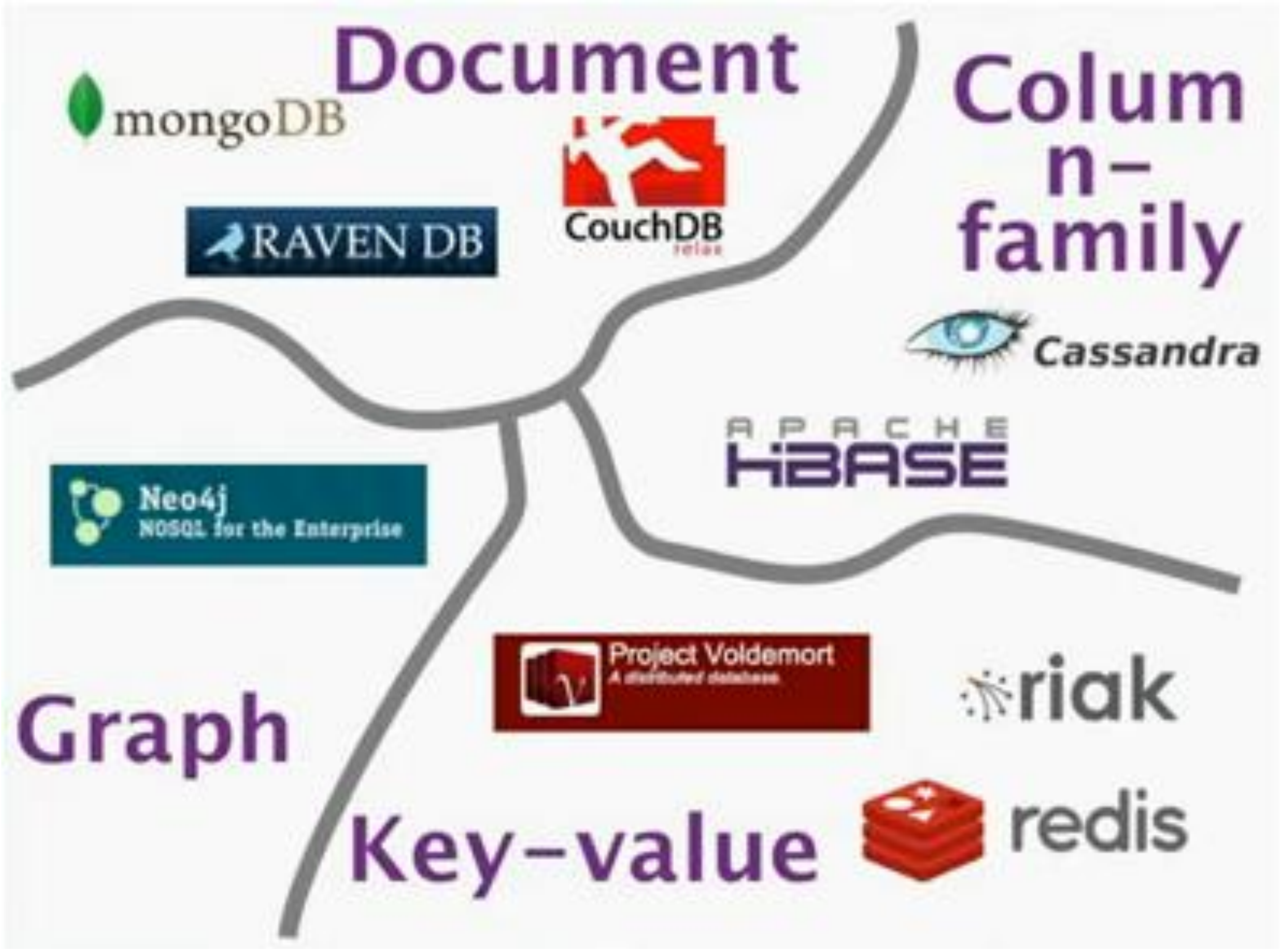
Graph

- Graph databases allow you to store entities and relationships between these entities.
- Entities are also known as nodes, which have properties. Think of a node as an instance of an object in the application.
- Relations are known as edges that can also have properties. Nodes are organised by relationships which allow you to find interesting patterns between the nodes.
- The organisation of the graph lets the data be stored just once and then interpreted in different ways based on relationships.

Graph

Typical applications:	Connected data, routing, dispatch and location-based services, recommendation engines
Data model:	Graph (nodes and edges)
Strengths:	Allows for the type of analysis that aggregate-oriented databases find very difficult
Weaknesses:	Has to traverse the entire graph to achieve a definitive answer.

Types of NoSQL databases



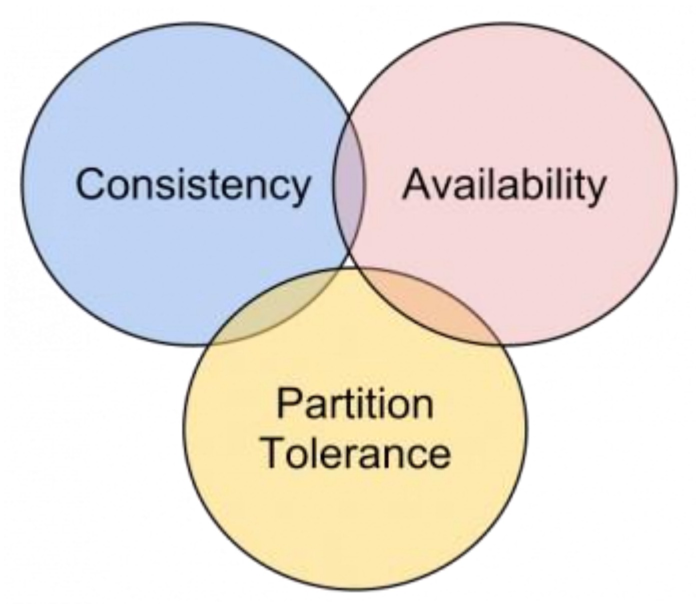
NoSQL and Consistency

- Distributed systems see read-write conflicts due to some nodes having received updates while other nodes have not. Eventual consistency means that at some point the system will become consistent once all the writes have propagated to all the nodes.

- To get good consistency, you need to involve many nodes in data operations, but this increases latency. So you often have to trade off consistency versus latency.

CAP theorem

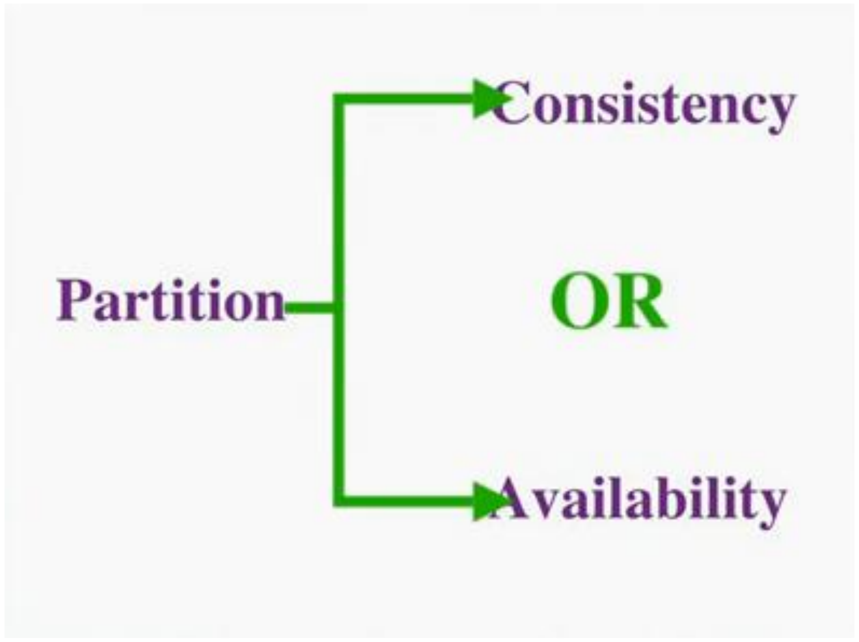
- The CAP theorem states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees:

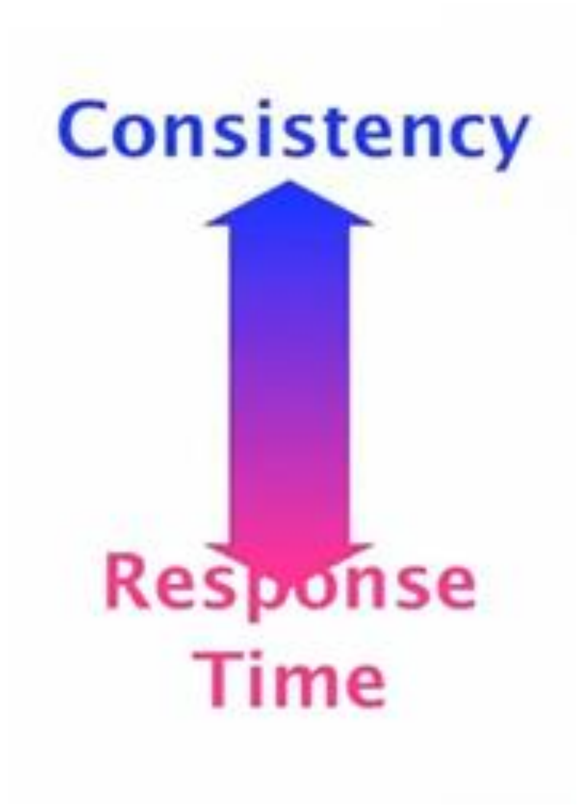


- Therefore if you get a network partition, you have to trade off availability of data versus consistency. In any distributed system, this is a possibility.

CAP theorem

- The trade-off of consistency may not always strictly be about availability; response time is also an issue





Reasons to use NoSQL

1. To improve programmer productivity by using a database that better matches an application's needs.

    - e.g. removing impedance mismatch by storing objects together in aggregates rather than splitting them up into relational tables

2. To improve data access performance via some combination of handling larger data volumes, reducing latency, and improving throughput.

    - When a database is large enough to be split over several database servers, NoSQL may be a good option

Reasons to stick with Relational DBs

- They are well-known, therefore it is easier to find people with experience of using them
- The technology is more mature and less likely to encounter problems
- Many other tools are built on relational technology

"A DBA walks into a NOSQL bar, but turns and leaves because he couldn't find a **table**"

Polyglot persistence

- Polyglot: the ability to speak multiple languages
- It is predicted that in the future, developers will make use of a range of different technologies for the persistence (storage) of data
- Relational databases and types of NoSQL databases can be utilised as necessary to solve the particular problems to which they are best suited