# NoSQL

## Go Beyond Relational Model

**Pros of Relational DBs**

1. Simple, can capture nearly any business use case
2. Can integrate multiple applications via shared data store
3. Standard interface language SQL
4. ad-hoc queries across and within data aggregates
5. Fast, reliable, concurrent, consistent

**Cons of Relational DBs**

1. Object Relational (OR) impedance mismatch
2. not good with big data
3. not good with clustered/replicated servers

Adoption of NoSQL driven by cons of Relational (e.g. a lot of work to dissemble and reassemble the aggregate)

But "Polyglot persistence" => Relational will not go away

# Big Data

## Definition & 3Vs

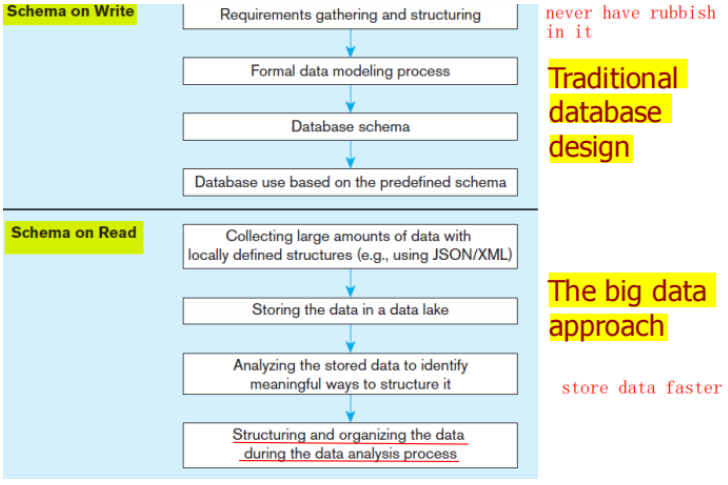
Data that exist in very large volumes and many different varieties (data types) and that need to be processed at a very high velocity (speed)

- **Volume**-Much larger quantity of data than typical for relational DBs
- **Variety**-Lots of different data types and formats
- **Velocity**-data comes at very fast rate (e.g. mobile sensors, web click stream)

## Schema on write vs. schema on read

**Schema on Write**-preexisting data model, how traditional databases are designed (relational databases)

**Schema on Read**-data model determined later, depends on how you want to use it (XML, JSON); capture and store the data, and worry about how you want to use it later.



## Data Lake

A large integrated repository for internal and external data that does not follow a predefined schema.

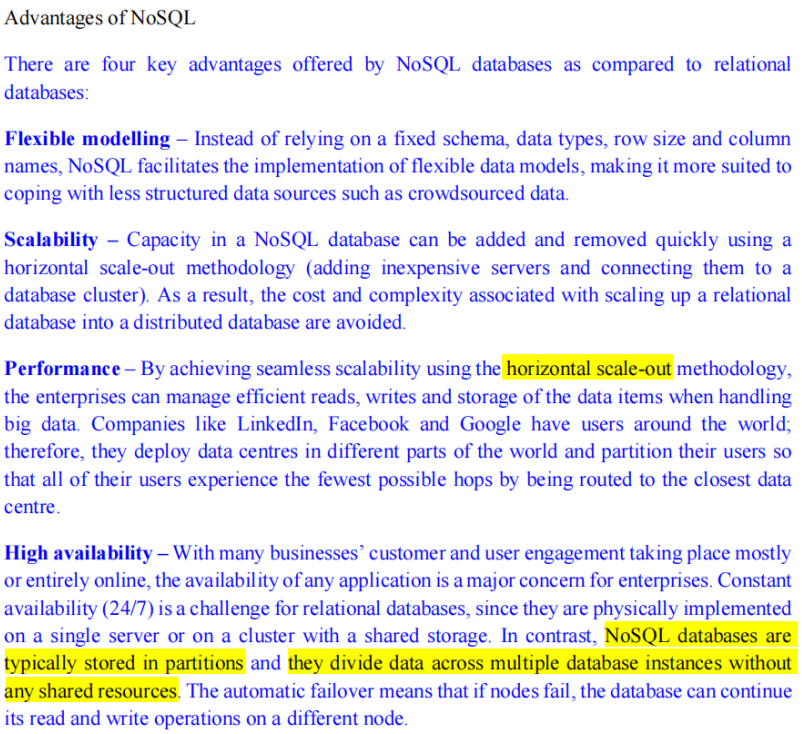
Capture everything, dive in anywhere, flexible access

# NoSQL database

## Properties

1. Doesn't use relational model or SQL language
2. Runs well on distributed servers
3. Most are open-source
4. Built for the modern web
5. Schema-less (though maybe some implicite schema); Supports schema on read
6. Not ACID compliant (can't guarantee instant transfer)
7. Eventually consistent

**Goal:** to improve programer productivity (OR mismatch); to handle larger dta volumes and throughput (big data)



Advantages of NoSQL

There are four key advantages offered by NoSQL databases as compared to relational databases:

**Flexible modelling** – Instead of relying on a fixed schema, data types, row size and column names, NoSQL facilitates the implementation of flexible data models, making it more suited to coping with less structured data sources such as crowdsourced data.

**Scalability** – Capacity in a NoSQL database can be added and removed quickly using a horizontal scale-out methodology (adding inexpensive servers and connecting them to a database cluster). As a result, the cost and complexity associated with scaling up a relational database into a distributed database are avoided.

**Performance** – By achieving seamless scalability using the horizontal scale-out methodology, the enterprises can manage efficient reads, writes and storage of the data items when handling big data. Companies like LinkedIn, Facebook and Google have users around the world; therefore, they deploy data centres in different parts of the world and partition their users so that all of their users experience the fewest possible hops by being routed to the closest data centre.

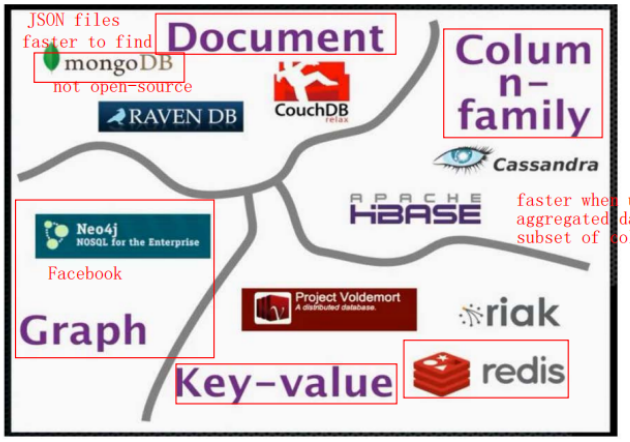
**High availability** – With many businesses' customer and user engagement taking place mostly or entirely online, the availability of any application is a major concern for enterprises. Constant availability (24/7) is a challenge for relational databases, since they are physically implemented on a single server or on a cluster with a shared storage. In contrast, NoSQL databases are typically stored in partitions and they divide data across multiple database instances without any shared resources. The automatic failover means that if nodes fail, the database can continue its read and write operations on a different node.

Comparison Between NoSQL and Relational DBs

NoSQL DBs need to be available all the time. Facebook / Twitter needs to ensure that users can access and post to its site and app all the time, even if the data is not consistent. For example, users expect to always see the number of likes on a post (or retweet count on a tweet), they do not mind if that number is slightly outdated.

Relational DBs sacrifice availability for consistency. This means the DBs need to be consistent all the time for all the users. If there is potential for inconsistencies (such as two people booking the same seats to watch a movie), the database will sometimes be unavailable.

## Types of NoSQL databases




## Key-value stores

**Key**=Primary key

**Value**=anything (number, array, image, JSON); The application is in charge of interpreting what it means (most flexible and least structured)

**Operations**-Put (store), Get, Update

**Examples:** Riak, Redis, Memcached, BerkeleyDB, HamsterDB, Amazon DynamoDB, Project Voldemort, Couchbase

## Document databases

Similar to a key-value store except that the document is examinable by the databases, so its content can be queried, and parts of it updated

**Document**=JSON file

**Examples:** MongoDB, CouchDB, Terrastore, OrientDB, RavenDB

MongoDB example

```
start the mongodb server, then start the mongo shell with "mongo"
show dbs// show a list of all databases
use test// use the database called 'test'
show collections// show all collections in the database 'test'
db.students.insert( {name: "Jack", born: 1992} )// add a doc to collection
db.students.insert( {name: "Jill", born: 1990} )// add a doc to collection
db.students.find()// list all docs in students
db.students.find( {name: "Jill"} )// list all docs where name field = 'Jill'
db.students.update( {name: "Jack"}, {$set: {born: 1990}} ) // change Jack's year
db.students.remove( {born: 1990} ) // delete docs where year = 1990

// now insert complex documents from file –note repeating group, no schema
db.students.find().forEach(printjson)// print all docs in neat JSON format
db.students.find( {born:1990}, {name: true} )// print names for all born in 1990
db.students.update( {id:222222}, {$addToSet:
{subjects: {subject: "English", result: "H1"}}} )        Update data deep in hierarchy
db.students.find( {id:222222}, {_id:false, subjects:true} ).forEach(printjson)

db.students.insert( {name: "John", color: "blue"} )      可能没有结果/blue
// add a new student – different schema but still works
```

### Aggregate-oriented

Key-value, documnet store and column-family are aggregate-oriented-store business object in its entirety databases.

### Pros

1. entire aggregate of data is stored together (no need for transactions)
2. efficient storage on clusters/distributed databases

**Cons**

1. hard to analyse across subfields of aggregates (e.g. sum over products instead of orders)

### Column families

Columns rather than rows are stored together on disk; This is like automatic vertical partitioning

Makes analysis faster, as less data is fetched

Related columns grouped together into families

**Examples:** Cassandra, BigTable, HBase (Facebook, Netflix, Twitter)

### Graph databases

A graph is a node-and-arc network; and Graphs are difficult to program in relational DB

A graph DB stores entities and their relationships; Graph queries deduce knowledge from the graph

**Examples:** Neo4j (Airbnb, Microsoft), Infinite Graph, OrientDBv, FlockDB, TAO

### Summary

1. Key-value stores
   A simple pair of a key and an associated collection of values. Key is usually a string. The database has no knowledge of the structure or meaning of the values
2. Document stores
   Like a key-value store, but document goes further than value. The document is structured, so specific elements can be manipulated separately.
3. Column-family stores
   Data is grouped in column groups/families for efficiency reasons.
4. Graph-oriented databases
   Maintain information regarding the relationships between data items. Nodes with properties

# CAP theorem restate

Fowler's version: If you have a distributed database, when a partition occurs, you must then choose consistency OR availability. (EVERY NoSQL is a distributed database; most NoSQL choose availability)

## ACID vs BASE

**ACID**-Atomic, Consistent, Isolate, Durable

**BASE**-Basically, Available, Soft State, Eventual Consistency

**Basically Available**: This constraint states that the system does guarantee the availability of the data; there will be a response to any request. But data may be in an inconsistent or changing state.

**Soft state**: The state of the system could change over time-even during times without input there may be changes going on due to eventual consistency

**Eventual Consistency**: The system will eventually become consistent once it stops receiving input. The data will propagate to everythwere it needs to, sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one.