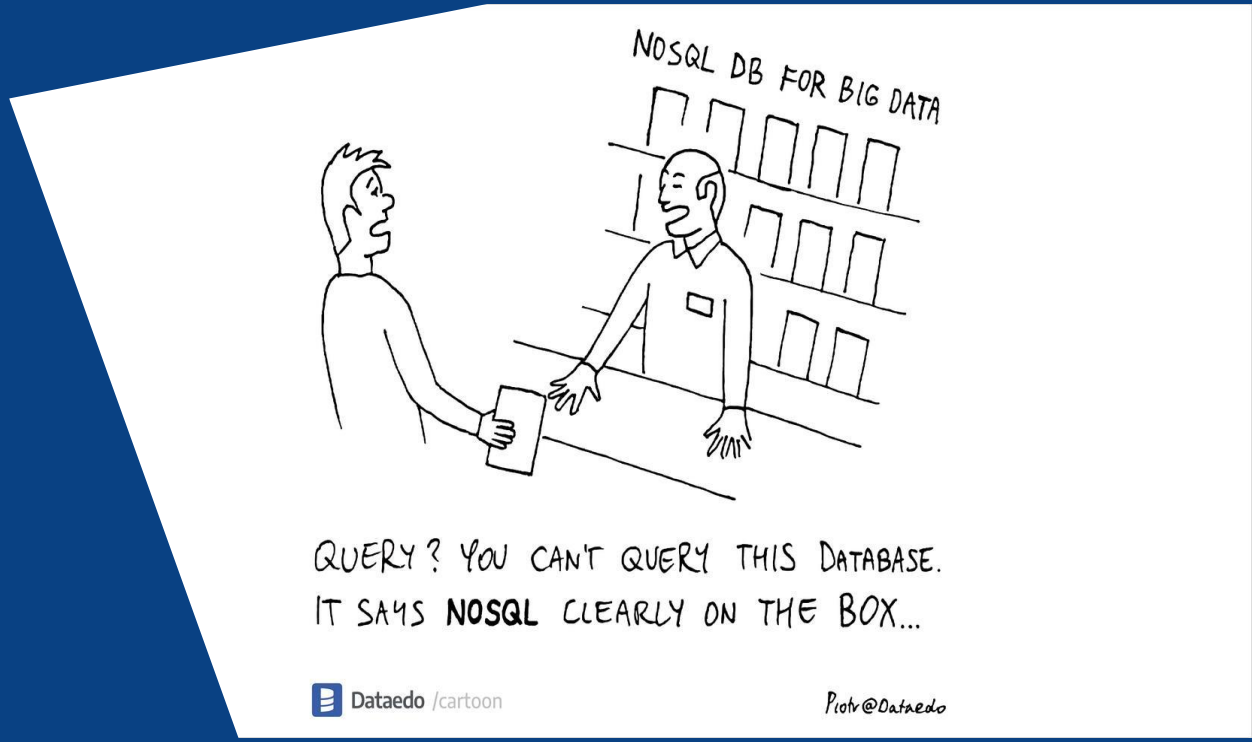
THE UNIVERSITY OF MELBOURNE

NoSQL databases

Database Systems & Information Modelling
INFO90002



Week 10 – Non-relational databases
Dr Tanya Linden
Dr Greg Wadley
Dr Renata Borovica-Gajic
David Eccles

Relational Databases pro's and con's

In a relational database, every **table** has a **schema**.

Every **column** has a pre-defined **size** and **data type.**

Each **row** in the table must **conform** the table **design**.

Tables are **related** to each other using **foreign keys**

Data **duplication** is **eliminated**

One piece of information is ideally stored once in the database

- If a piece of data changes, then ideally it only needs to be changed once.

We can use SQL to search for / retrieve data

However,

not good with "big" data

not good with distributed (partitioned) databases

Problem "Big Data"

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 traditional relational databases

VARIETY – lots of different data types and formats

VELOCTY – data is coming in at a very fast rate (e.g. mobile sensors, data click streams)

Adoption of NoSQL is driven by “cons” of Relational databases

Relational DBMS will not go away

Big Data - characteristics

Schema on *Read*, rather than Schema on Write

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

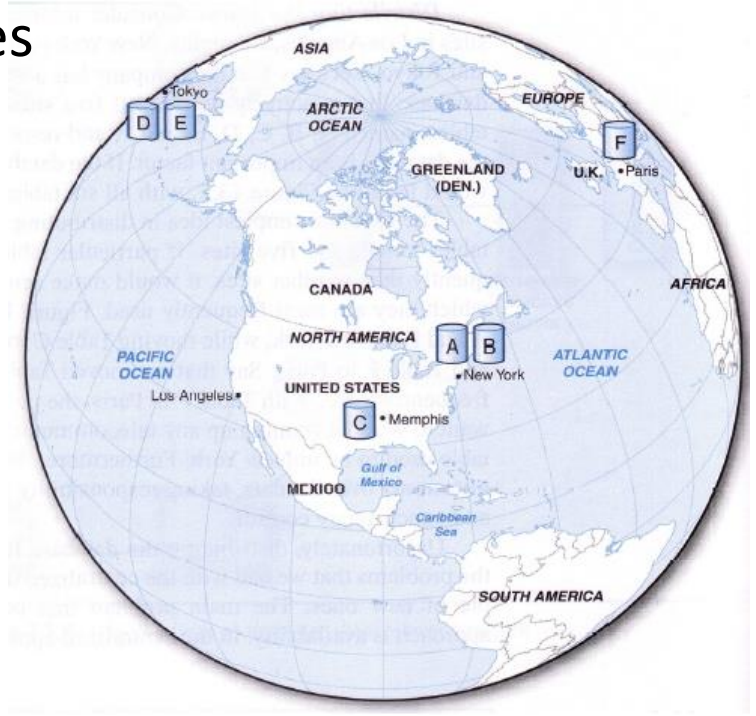
- worry about how you want to use it later

Problem “Distributed data”

Distributed, *especially partitioned*, databases are not a good fit for some relational features, e.g. foreign keys and transactions.

Foreign keys may be stored in remote locations

Network latency – creates bottle necks for resources



Non Relational Databases

A NoSQL database is a non-relational database

There are different types of NoSQL databases. E.g.

- Key-value stores such as Redis, Azure Table Storage
- Column-based stores like Cassandra, Druid
- Graph databases like NEO4, AllegroGraph, ArangoDB
- Document databases like MongoDB, Azure Cosmos DB.

Document Databases

- A document is NOT a word processing document! It is NOT a .pdf document or a spreadsheet!
- A document database stores entities as documents, meaning **JSON** (pronounced Jason) documents

JSON documents are similar to XML files.

Types of NoSQL: key-value store

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

Key = primary key
Value = anything (number, array, image, JSON)
No query language
The *application* is in charge of interpreting what it means.

Popular application - recording sessions in applications that require logins. Data about each session (period from login to logoff) is recorded in a key-value store. Sessions are marked with identifiers and all data recorded about each session (e.g. client IP address, profile, credit card (if transaction), etc).
Session details on shopping (adding items to shopping carts). Note, RDBMS are more suitable to storing payment details.

Key names can range from a simple number to
specific descriptions of the value in the Value field.
Can be compared to a Dictionary where words are keys and their explanations are values.

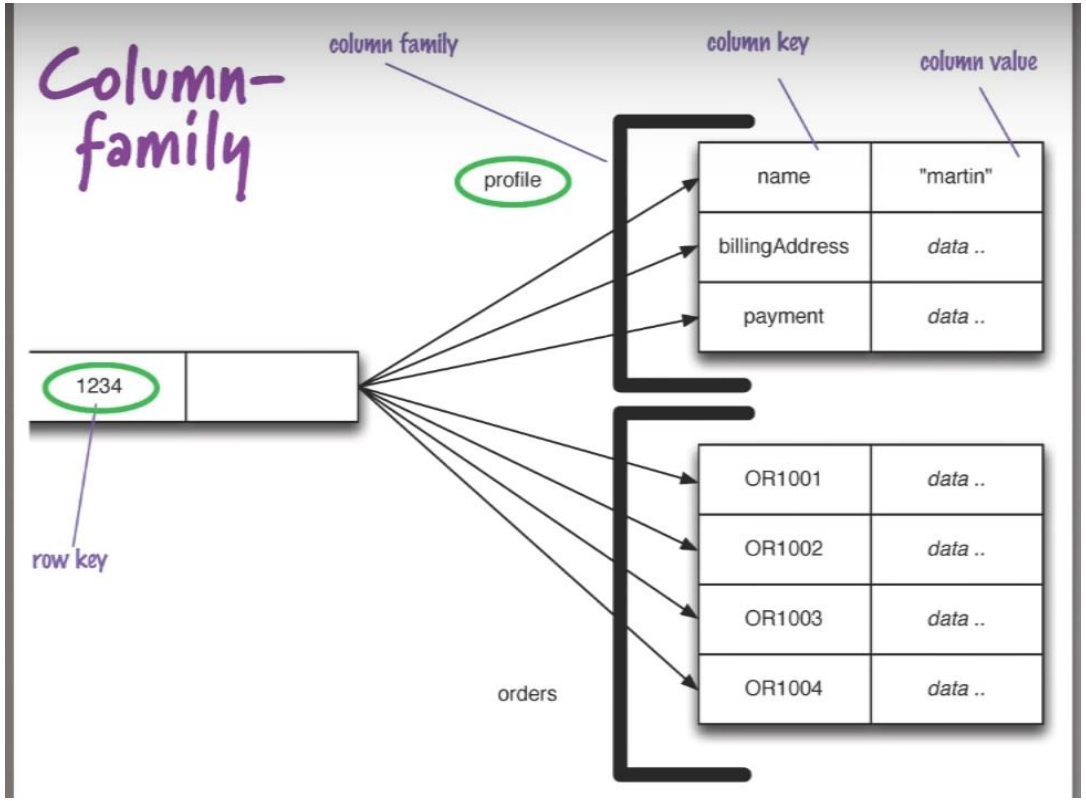
Examples:
Riak, Redis, Memcached, Berkeley DB, Project Voldemort, Couchbase

Types of NoSQL: column family

"Column family" or "wide-column" is like a relational table.
It contains many "rows".
But each row can store a *different set of columns*.

Columns rather than rows are stored together on disk.
Makes analysis by column faster – not for OLTP.

Examples: Cassandra, BigTable, HBase, DynamoDB



Aggregate-oriented databases

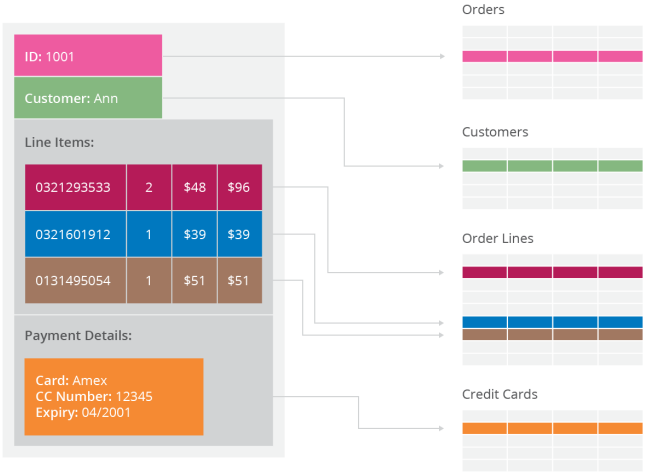
Key-value, *document store* and *column-family* are "aggregate-oriented" databases (in Fowler's terminology)

Pros

- entire aggregate of data is stored together
- less need for transactions
- efficient storage on clusters / distributed databases

Cons

- hard to analyse across subfields of aggregates
  e.g. sum over products instead of orders



Product	revenue	prior revenue
321293533	3083	7043
321601912	5032	4782
131495054	2198	3187
...	...	...

Graphs

A data structure consisting of nodes/vertices and ties/edges/arcs

Nodes/vertices represent entities

Arcs/ties/edges represent relationships

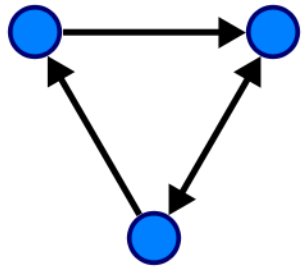
May be directed or undirected

In a graph database:

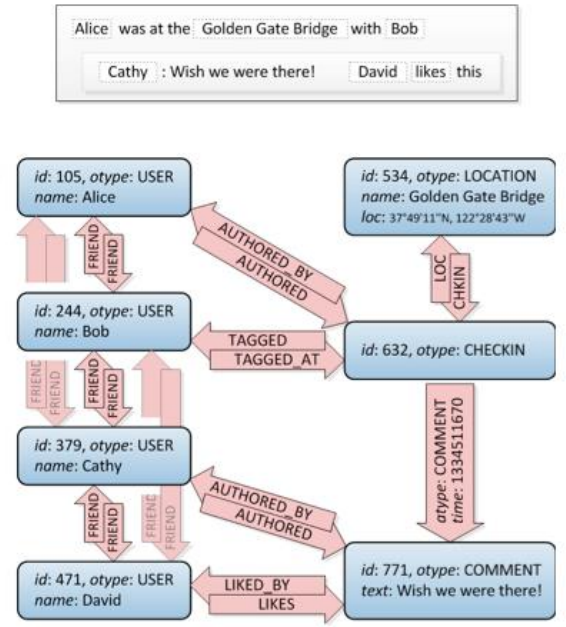
- nodes and ties can have properties and types
- the emphasis is on relationships
- connected nodes physically “point” to each other in the database

Practical examples:

  - Passing messages on social media (who talks to whom)
  - Road networks
  - Networked devices
  - Geo-spatial data (e.g. volcanoes)



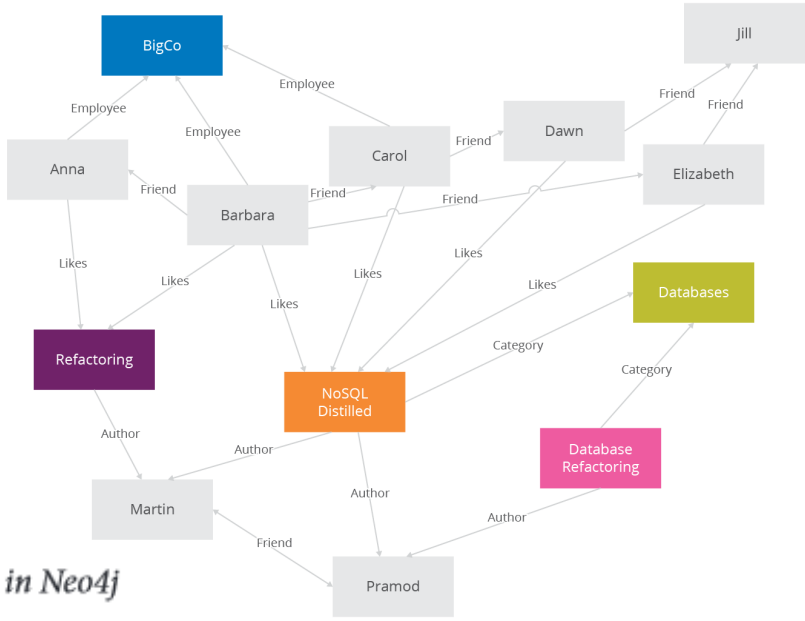
directed graph (source: Wikipedia)



social graph (source: Facebook)

Types of NoSQL: graph database

A ‘graph' is a node-and-tie network
Graphs are difficult to program in relational DB
A *graph DB* stores entities and their relationships
Graph queries deduce knowledge from the graph using node relationships
Often used with OLTP databases/systems



Examples:

- Neo4J
- Infinite Graph
- OrientDB
- FlockDB
- TAO

Table 2-1. Finding extended friends in a relational database versus efficient finding in Neo4j

Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

The *Neo4J* graph database

Nodes

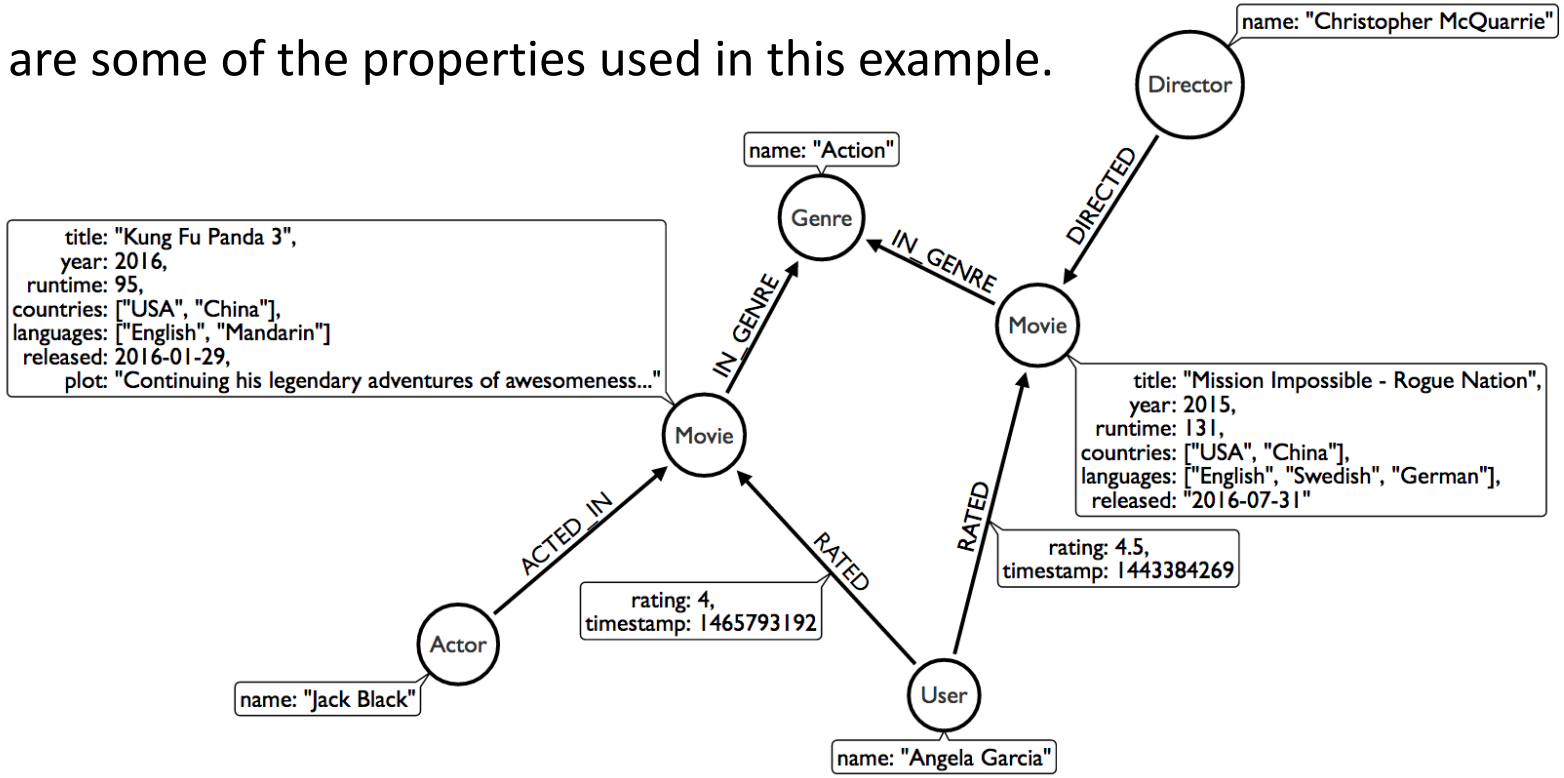
Movie, Actor, Director, User, Genre are the labels used in this example.

Relationships

ACTED_IN, IN_GENRE, DIRECTED, RATED are the relationships used in this example.

Properties

title, name, year, rating are some of the properties used in this example.



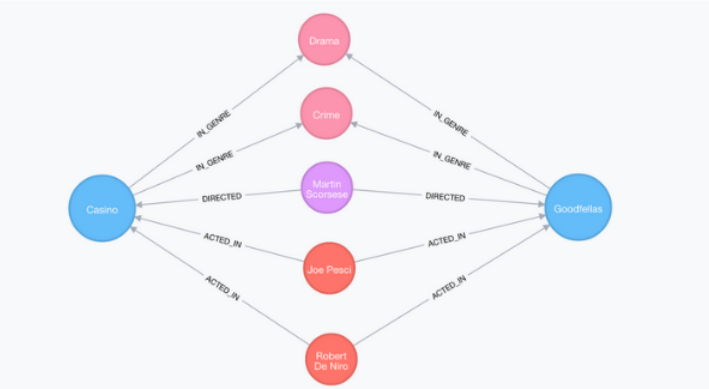
Neo4J queries

Queries are written in the *Cypher* language

find	MATCH (m:Movie)<-[:RATED]-(u:User)	Search for an existing graph pattern
filter	WHERE m.title CONTAINS "Matrix"	Filter matching paths to only those matching a predicate
aggregate	WITH m.title AS movie, COUNT(*) AS reviews	Count number of paths matched for each movie
return	RETURN movie, reviews	Specify columns to be returned by the statement
order	ORDER BY reviews DESC	Order by number of reviews, in descending order
limit	LIMIT 5;	Only return first five records

Content-Based Filtering

Recommend items that are similar to those that a user is viewing, rated highly or purchased previously.

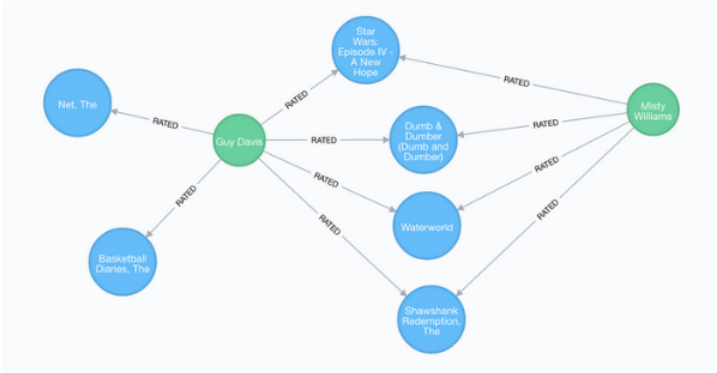


"Products similar to the product you're looking at now"

```
MATCH p=(m:Movie {title: "Net, The"})-[:ACTED_IN|:IN_GENRE|:DIRECTED*2]-()
RETURN p LIMIT 25
```

Collaborative Filtering

Use the preferences, ratings and actions of other users in the network to find items to recommend.



"Users who bought this thing, also bought that other thing."

```
MATCH (m:Movie {title: "Crimson Tide"})<-[:RATED]-(u:User)-[:RATED]->(rec:Movie)
RETURN rec.title AS recommendation, COUNT(*) AS usersWhoAlsoWatched
ORDER BY usersWhoAlsoWatched DESC LIMIT 25
```

Types of NoSQL: document database

Like a key-value database,
except that the "value" (document)
is "examinable" by the database, so
its contents can be queried and
updated

document = object represented by
JSON

Examples:
MongoDB, CouchDB, Terrastore,
OrientDB, RavenDB

Some notable NoSQL users

- Google – BigTable
  - search, gmail, maps, youtube
- Facebook – Cassandra, Tao, Giraph
  - messaging, social graph
- Amazon – SimpleDB, DynamoDB
  - large scale e-commerce and analytics, cloud db
- Instagram - Cassandra
  - social media newsfeed
- LinkedIn – CouchDB, MongoDB
  - monitoring and analysis of operational data
- The Guardian - MongoDB
  - newspaper articles, user identity
- FourSquare - MongoDB
  - venues and user checkins

ACID vs BASE

ACID (Atomic, Consistent, Isolated, Durable)
vs
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 everywhere 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.

Document DB: Replication and Consistency

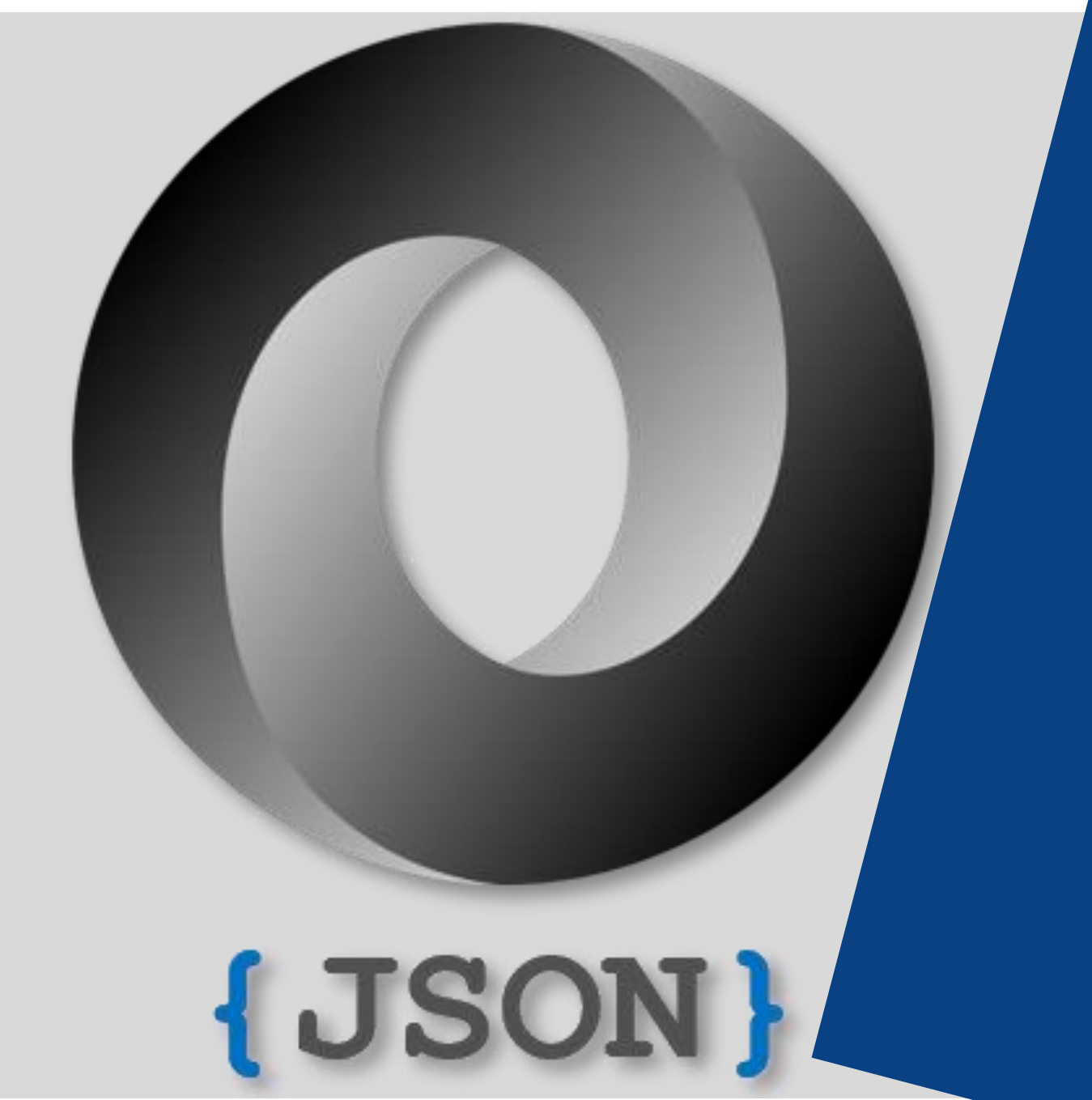
RDBMS: Often a single copy of the database. You wouldn't expect that a bank customer has multiple versions of his/her bank balance throughout multiple databases.

NoSQL Document DB: are often **replicated**. This allows them to satisfy **high volume** of queries and searches. Processes in place for changes to a document to be **propagated** to multiple **replicas** of the database.

RDBMS: Enforce strong consistency.
After data is updated, every user sees new updated values immediately
- E.g. no-one should see an out of date value of the customers bank balance!

DocumentDB: **Eventual consistency**
After data is updated (e.g. customer address)
- changes are made in multiple documents (that store the address)
- changes are propagated through multiple replicated databases

All data is **eventually updated**.
Does it matter if not everyone can see absolutely latest comment?



THE UNIVERSITY OF MELBOURNE

JSON

Documents | Properties | Normalizing | Parent/Child | Queries | Indexes

JSON documents

Java**S**cript **O**bject **N**otation (pronounced Jason)

- represents a (JavaScript)v object and its properties

An object consists of a set of attribute-value pairs, including arrays of objects

- has a ‘tree’ structure

Originally used for transmitting data between computers

Now the storage format for Document databases

Not normalised (not even 1NF)

```
[       // array of students
{
    id: 111111,
    name: "Alan",
    born: 1990,
    address: "1 Smith st",
    subjects: [
        { subject: "Database", result: "H1" },
        { subject: "Programming", result: "H2A" }
    ]
}
,
{
    id: 222222,
    name: "Betty",
    born: 1992,
    address: "2 Two st",
    awards: "Best Student",
    subjects: [
        { subject: "Maths", result: "H1" },
        { subject: "Science", result: "H1" },
        { subject: "History", result: "H1"}
    ]
}
,
{
    id: 333333,
    name: "Chris",
    born: 1990,
    address: "3 Three st",
    subjects: [
        { subject: "Database", result: "H1" }
    ]
}
]
```

What is JSON?

JSON is a syntax for passing around objects that contain

- name/value pairs
- arrays
- other objects.

```
{"my_it_skills": {
            "programming": [
                        {"name": "Python",
                         "yearstarted": "2016"
                        },
                        {"name": "JavaScript",
                         "yearstarted": "2019"
                        } ],
            "database":[
                        {"name": "SQL",
                         "yearstarted": "2019"
                        } ]
}}
```

- Curly braces { } act as containers
- Square brackets [ ] hold arrays
  - Array elements are separated by commas
- Names and values are separated by a colon
- Values surround by double-quotes (in some notations names are also in double quotes)

- Human readable
- Hierarchical (can store values within values)

These terms are interchangeable

- Name – Value pair
- Key – Value pair
- Attribute – Value pair

Another example of a JSON document

```
{
    "id": "WakefieldFamily",
    "parents": [
        { "familyName": "Wakefield", "givenName": "Robin" },
        { "familyName": "Miller", "givenName": "Ben" }
    ],
    "children": [
        {
            "familyName": "Merriam",
             "givenName": "Jesse",
            "gender": "female", "grade": 1,
            "pets": [
                { "givenName": "Goofy" },
                { "givenName": "Shadow" }
            ]
        },
        {
            "familyName": "Miller",
             "givenName": "Lisa",
             "gender": "female",
             "grade": 8 }
    ],
    "address": { "state": "NY", "county": "Manhattan", "city": "NY" },
    "isRegistered": false
}
```

Rows vs Documents

RDBMS: Data is stored in Rows.

- 3 customers requires 3 rows

NoSQL Document DB: Data is stored in multiple **Documents**

- 3 customers require 3 documents *

* This is flexible

Relational Store	Document Store
Rows	Documents

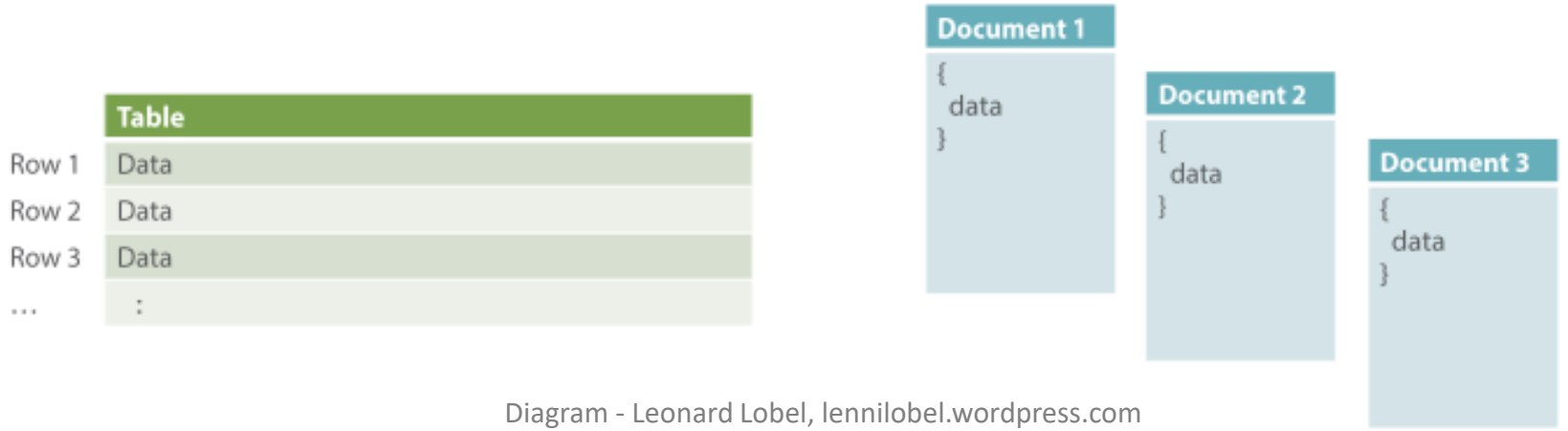


Diagram - Leonard Lobel, lennilobel.wordpress.com

Columns vs Properties

RDBMS: Each row consists of multiple Columns

NoSQL Document DB: Each **document** consists of **multiple Properties**

Relational Store	Document Store
Rows	Documents
Columns	Properties

Col1	Col2	Col3	Col4
Data	Data	Data	Data
Data	Data	Data	Data
Data	Data	Data	Data

Document 1

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```

Document 2

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```

Document 3

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data,
  "prop4": data
}
```

Diagram - Leonard Lobel, lennilobel.wordpress.com

Documents are schema free

RDBMS: Each row must conform to the schema

NoSQL Document DB: Each document may use any **properties** (name/value pairs)

Document 3 has additional properties for the customer's name

Each document must have an **ID property**

Relational Store	Document Store
Rows	Documents
Columns	Properties
Strongly-typed schemas	Schema-free

ID	Name	IsActive	Dob
1	John Smith	True	8/30/1964
2	Sarah Jones	False	2/18/2002
3	Adam Stark	True	7/13/1987

Document 1

```
{
  "id": "1",
  "name": "John Smith",
  "isActive": true,
  "dob": "1964-30-08"
}
```

Document 2

```
{
  "id": "2",
  "fullName": "Sarah Jones",
  "isActive": false,
  "dob": "2002-02-18"
}
```

Document 3

```
{
  "id": "3",
  "fullName":
  {
    "first": "Adam",
    "last": "Stark"
  },
  "isActive": true,
  "dob": "2015-04-19"
}
```

Diagram - Leonard Lobel, lennilobel.wordpress.com

Document properties are indexed

NoSQL Document DB: As each document is added to the database, **every property** is automatically **indexed.**

This means that **retrieval** speeds for any data is very **quick**

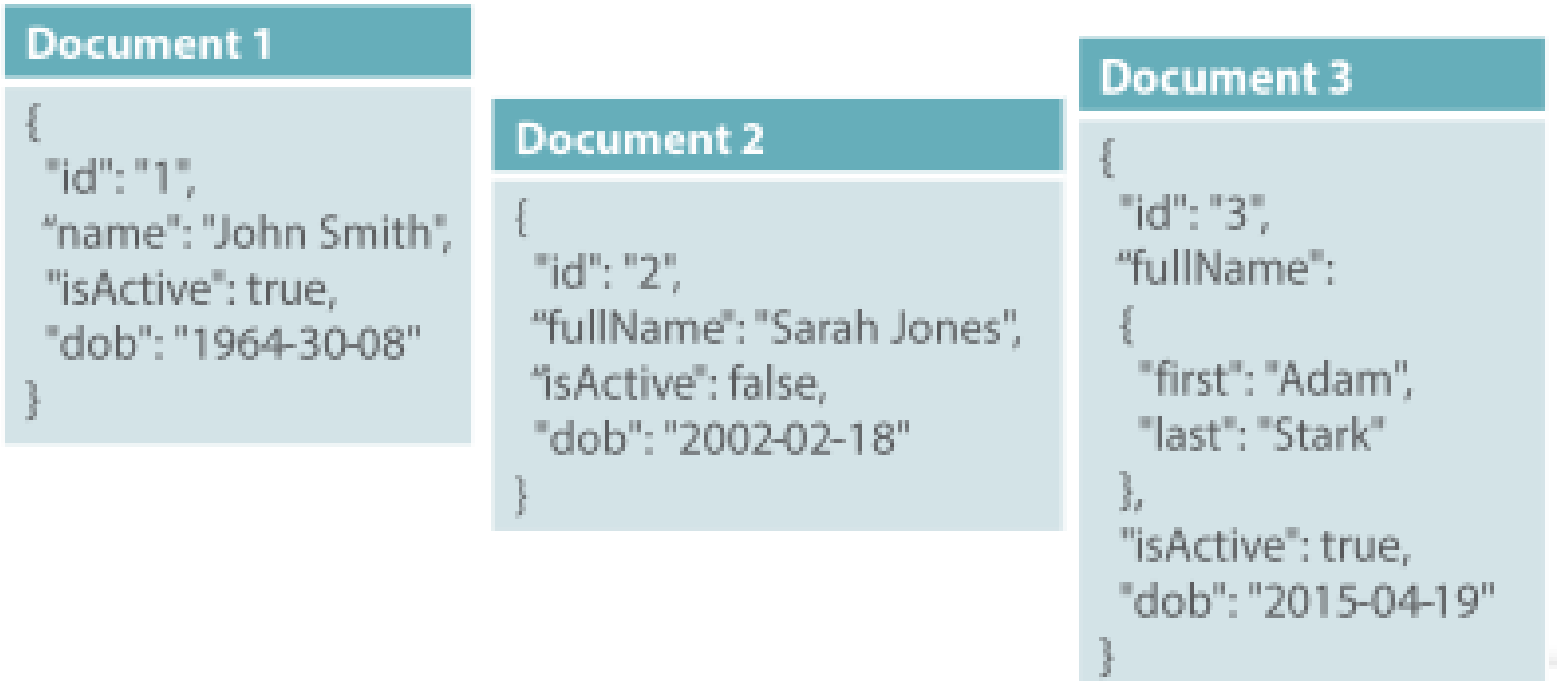


Diagram - Leonard Lobel, lennilobel.wordpress.com

Normalised vs Denormalised

RDBMS: Tables have foreign keys to form relationships between tables

Tables must be joined when queried. There is performance overhead.

NoSQL Document DB: Each document is **denormalised**.

'Holdings' data is stored in the same document as customers data.

Queries are simple to write and fast to execute.

Fewer Joins.

Relational vs. Document

Relational Store	Document Store
Rows	Documents
Columns	Properties
Strongly-typed schemas	Schema-free
Highly normalized	Typically denormalized

User Table

UserID	Name	Dob
1	John Smith	8/30/1964

Holdings Table

StockID	UserID	Qty	Symbol
1	1	100	MSFT
2	1	75	WMT

Document

```
{
  "id": "1",
  "name": "John Smith",
  "dob": "1964-30-08",
  "holdings": [
    { "qty": 100, "symbol": "MSFT" },
    { "qty": 75, "symbol": "WMT" }
  ]
}
```

Diagram - Leonard Lobel, lennilobel.wordpress.com

Redundant Data

RDBMS: Redundant Data is eliminated

NoSQL Document DB: Documents contain **redundant data**

The description of product "BC" is repeated in many documents

Relational Store	Document Store
Rows	Documents
Columns	Properties
Strongly-typed schemas	Schema-free
Highly normalized	Typically denormalized

Document

```
{
  "name": "John Smith",
  "orderDate": "2015-30-03",
  "details": [
    { "qty": 5, "code": "BC", "desc": "Black chair" },
    { "qty": 1, "code": "RT", "desc": "Red table" },
    { "qty": 2, "code": "YC", "desc": "Yellow clock" },
  ]
}
```

Document

```
{
  "name": "Sarah Jones,
  "orderDate": "2015-16-04",
  "details": [
    { "qty": 1, "code": "PL", "desc": "Purple lamp" },
    { "qty": 3, "code": "YC", "desc": "Yellow clock" },
  ]
}
```

Document

```
{
  "name": "Adam Stark",
  "orderDate": "2015-06-05",
  "details": [
    { "qty": 2, "code": "BC", "desc": "Black chair" },
    { "qty": 1, "code": "YC", "desc": "Yellow clock" },
  ]
}
```

Diagram - Leonard Lobel, lennilobel.wordpress.com

Parent / Child relationships

DocumentDB: This document contains a 1:M relationship within a single document
This is reasonable when **there is a limit** to the **maximum** number of comments (children)
It is **impractical** to attempt to store an **infinite** number of children in a single document

Relational Store	Document Store
Rows	Documents
Columns	Properties
Strongly-typed schemas	Schema-free
Highly normalized	Typically denormalized

Document

```
{
  "postid": "1",
  "title": "My blog post",
  "body": "Postcontent…",
  "comments": [
    "comment #1",
    "comment #2",
    "comment #3",
    "comment #4",
```

Document

```
{
  "postid": "1",
  "title": "My blog post",
  "body": "Postcontent…",
  "comments": [
    "comment #1",
    "comment #2",
    "comment #3",
    "comment #4",
        :
    "comment #1598873",
```

Diagram - Leonard Lobel, lennilobel.wordpress.com

Documents can be normalised

NoSQL Document DB: The design of documents is flexible

You may choose to create a 1:M relationship between documents

Here **comments** are stored as **separate documents**

Relational Store	Document Store
Rows	Documents
Columns	Properties
Strongly-typed schemas	Schema-free
Highly normalized	Typically denormalized

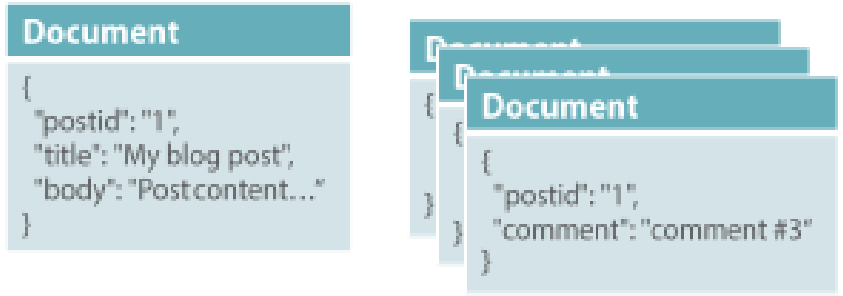


Diagram - Leonard Lobel, lennilobel.wordpress.com

Combine aspects of normalised and denormalised

DocumentDB: This design stores **multiple** 100 **comments** (children) per document

Relational Store	Document Store
Rows	Documents
Columns	Properties
Strongly-typed schemas	Schema-free
Highly normalized	Typically denormalized

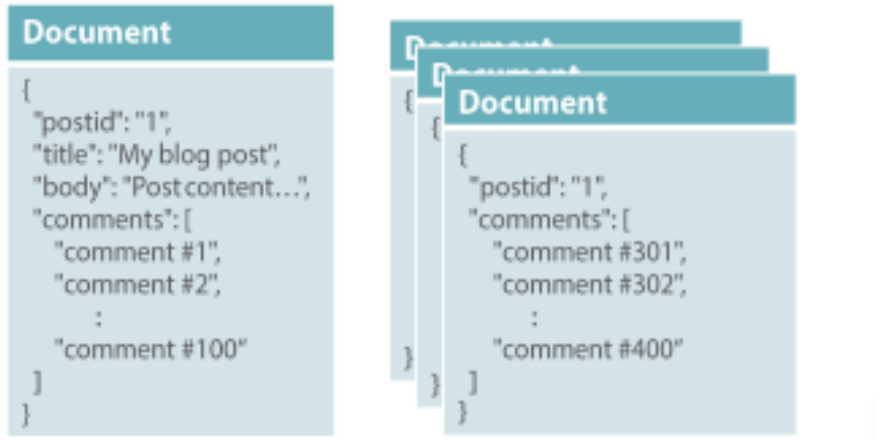


Diagram - Leonard Lobel, lennilobel.wordpress.com

Nested Objects

A JSON document can contain properties, arrays and objects

```
{
  "id": "AndersenFamily",
  "lastName": "Andersen",
  "address": {
    "state": "WA",
    "county": "King",
    "city": "seattle"
  }
}
```

```
[{
  "id": 1,
  "name": {
    "firstname": "Douglas",
    "surname": "Tucker"
  },
  "gender": "Male",
  "email": "dtucker0@imageshack.us",
  "children": [
    {
      "gender": "Female",
      "firstname": "Gloria"
    },
    {
      "gender": "Male",
      "firstname": "Jose"
    }
  ]
}, {
  "id": 2,
  "name": {
    "firstname": "Keith",
    "surname": "Dunn"
  },
  "gender": "Male",
  "email": "kdunn1@example.com",
  "children": [
    {
```

Manipulating data in a NoSQL database

Data in NoSQL document databases is manipulated using JSON

- E.g. db.<collection>.find() in MongoDB is equivalent to SELECT … FROM <table>
  - Replace <collection> with the actual collection name

NoSQL databases have “browsers”, e.g.

- Compass for MongoDB was developed by MongoDB developers
- Studio 3T for MongoDB developed by 3rd party
- NoSQLbooster for MongoDB developed by 3rd party

Some browsers can translate SQL into JSON

Remember:

- A NoSQL database really means a Non Relational database

THE UNIVERSITY OF MELBOURNE

Thank you