



**UNSW**  
SYDNEY

# NoSQL

COMP9311 24T3; Week 9

*By Zhengyi Yang, UNSW*

# Notice

- Additional consultation will be scheduled before the final exam
- Project 1: marks and sample solution will be released on Wednesday
- Assignment 2 will be due at 5pm next Monday

# Acknowledgements

Some parts of this slides are adopted from

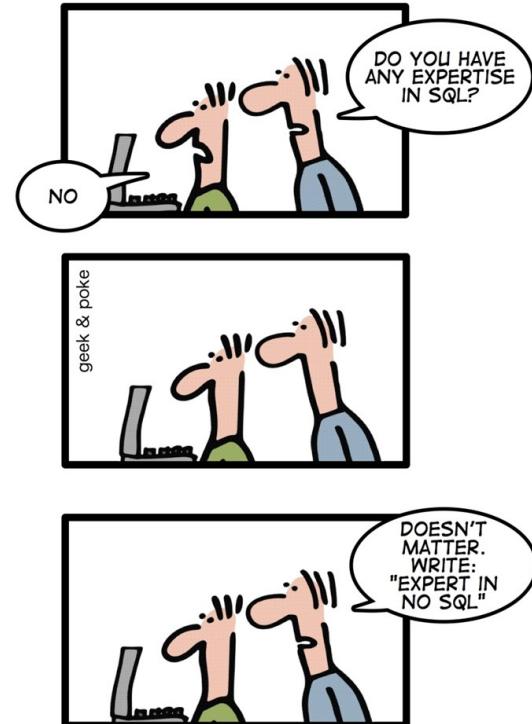
- “Database System Concepts” - 7th Edition
- Slides by Prof. George Kollios (Boston University)
- Slides by Asst Prof. Risa B. Myers (Rice University)
- Slides by Dr. David Novak (Masaryk University)
- Slides by Prof. Ying Zhang (UTS)
- Slides by Dr. Longbin Lai (UNSW)
- Neo4j Educator Resources
- Slides by myself at Rust Meetup Sydeny 2020

# NoSQL is Hot!

NoSQL stands for “**not only SQL**”:

- Non-tabular databases and store data differently than relational tables
- Firstly proposed in 2009

*HOW TO WRITE A CV*



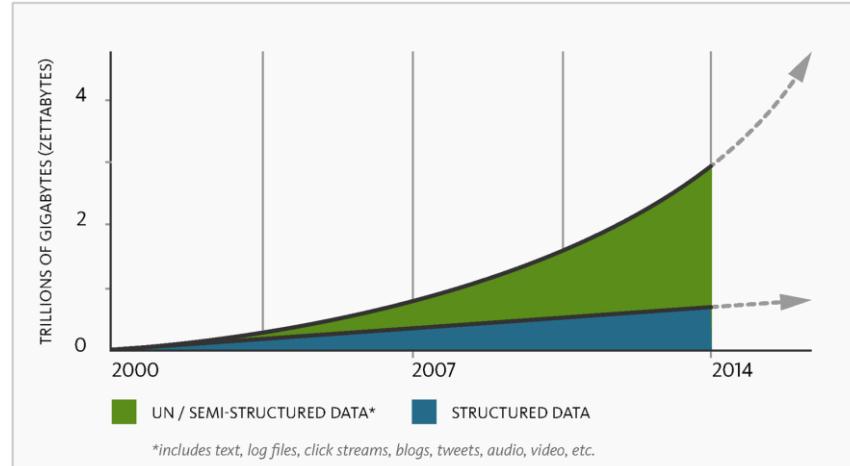
Leverage the NoSQL boom

# Big Data

- Very large volumes of data being collected
  - Driven by growth of web, social media, and more recently internet-of-things
  - Web logs were an early source of data
    - Analytics on web logs has great value for advertisements, web site structuring, what posts to show to a user, etc
- Big Data: differentiated from data handled by earlier generation databases
  - **Volume**: much larger amounts of data stored
  - **Velocity**: much higher rates of insertions
  - **Variety**: many types of data, beyond relational data

# Some Old Numbers

- **Facebook:**
  - 130TB/day: user logs
  - 200-400TB/day: 83 million pictures
- **Google: > 25 PB/day processed data**
- **Gene sequencing: 100M kilobases per day per machine**
  - Sequence 1 human cell costs Illumina \$1k
  - Sequence 1 cell for every infant by 2015?
  - 10 trillion cells / human body
- **Total data created in 2010: 1 ZettaByte (1,000,000 PB)/year**
  - ~60% increase every year



# Big Data is not only Databases

Big data is more about data analytics and on-line querying

Many components:

- Storage systems
- **Database systems**
- Data mining and statistical algorithms
- Visualization

# Features of RDBMS

- Data stored in columns and tables
- Relationships represented by data
- Data Manipulation Language
- Data Definition Language
- Transactions (ACID)
- Abstraction from physical layer
- Applications specify what, not how
- Physical layer can change without modifying applications

# The Value of Relational Databases

- A (mostly) **standard** data model
- Many well **developed** technologies
  - physical organization of the data, search indexes, query optimization, search operator implementations
- Good **concurrency** control (ACID)
  - transactions: atomicity, consistency, isolation, durability
- Many reliable **integration** mechanisms
  - “shared database integration” of applications
- **Well-established**: familiar, mature, support,...

# Data Management: Trends & Requirements

## Trends

- **Volume** of data
- **Cloud** comp. (IaaS)
- **Velocity** of data
- **Big** users
- **Variety** of data

## Requirements

- Real database **scalability** massive
  - database **distribution**
  - **dynamic** resource management
  - **horizontally** scaling systems
- Frequent **update** operations
- Massive **read** throughput
- **Flexible** database schema
  - semi-structured data

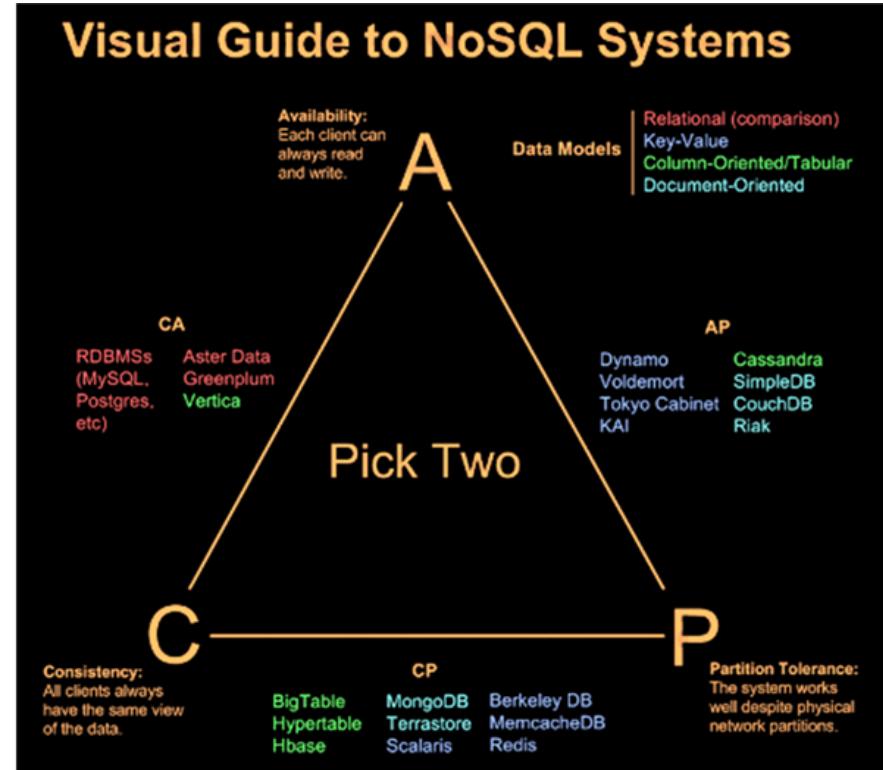
# RDBMS for Big Data

- relational **schema**
  - data in tuples
  - **a priori** known schema
- schema **normalization**
  - data split into tables (3NF)
  - queries merge the data
- **transaction support**
  - trans. management with ACID
  - Atomicity, Consistency, Isolation, Durability
  - safety first
- but current data are naturally **flexible**
- inefficient for large data
- slow in **distributed** environment
- full transactions very inefficient in **distributed** environment.

# CAP Theorem

At most two of the following three can be maximized at one time

- **Consistency**
  - Each client has the same view of the data
- **Availability**
  - Each client can always read and write
- **Partition Tolerance**
  - System works well across distributed physical networks



# Summary: Querying Big Data

- Transaction processing systems that need very high scalability
  - Many applications willing to sacrifice ACID properties and other database features, if they can get very high scalability
- Query processing systems that
  - Need very high scalability, and
  - Need to support non-relation data

# Processing Data - Terms

- **OLTP**: Online Transaction Processing (DBMSs)
  - **Database** applications
  - Storing, querying, multi-user access
- **OLAP**: Online Analytical Processing (Warehousing)
  - Answer multi-dimensional **analytical** queries
  - Financial/marketing reporting, budgeting, forecasting, ...
- **HTAP**: **Hybrid** Transaction/Analytical Processing

# NoSQL Databases

- NoSQL: Database technologies that are (mostly):
  - Not using the **relational** model (nor the SQL language)
  - Designed to run on **large clusters** (horizontally scalable)
  - **No schema** - fields can be freely added to any record
  - Open source
  - Based on the needs of the current big data era
- Other characteristics (often true):
  - easy **replication** support (fault-tolerance, query efficiency)
  - **Simple API**
  - **Eventually** consistent (not ACID)

# Just Another Temporary Trend?

- There have been **other trends** here before
  - object databases, XML databases, etc.
- But NoSQL databases:
  - are answer to **real practical problems** big companies have
  - are often developed by the **biggest players**
  - outside academia but based on **solid theoretical** results
    - e.g. old results on distributed processing
  - widely used

# The End of RDBMS?

- Relational databases are not going away
  - are ideal for a lot of structured data, reliable, mature, etc.
- RDBMS became one option for data storage

Using different data stores in different circumstances

Two trends:

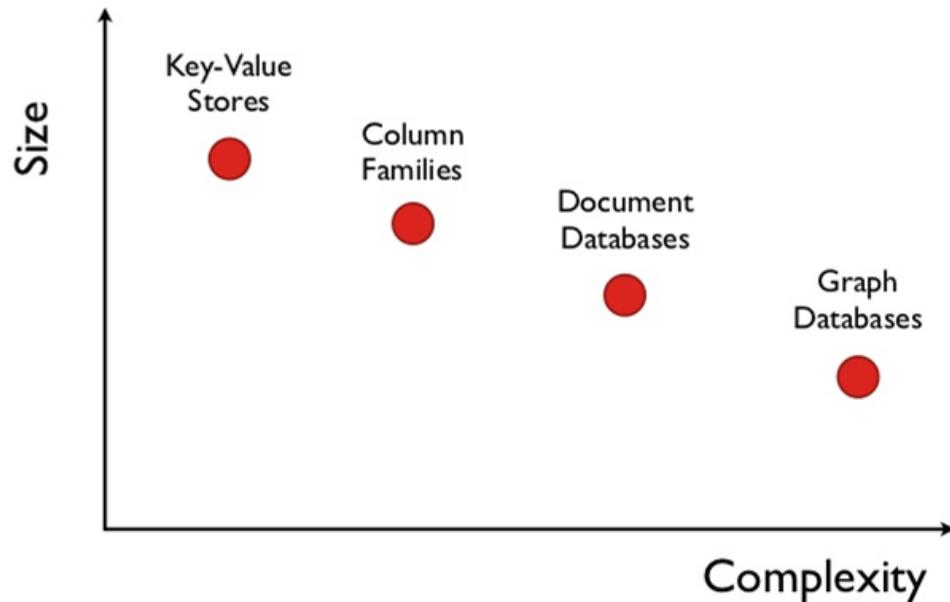
1. NoSQL databases implement standard RDBMS features
2. RDBMS are adopting NoSQL principles

# NoSQL Properties

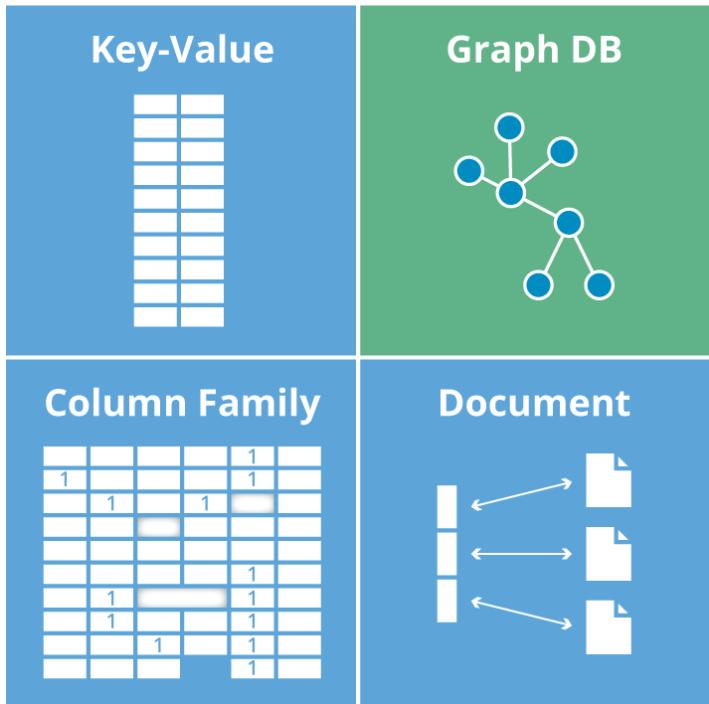
1. Flexible **scalability**
  - horizontal scalability instead of vertical
2. Dynamic **schema** of data
  - different levels of flexibility for **different** types of DB
3. Efficient **reading**
  - spend more time storing the data, but **read fast**
  - keep relevant information together
4. Cost **saving**
  - designed to run on **commodity** hardware
  - typically **open-source** (with a support from a company)

# NoSQL Databases

- Key-value stores
- Document databases
- Column-family stores
- Graph databases



# NoSQL Databases by Data Models



APACHE  
**HBASE**

 **CouchDB** relax

 **Cassandra**

 **riak**



 **mongoDB**

**HYPERTABLE** INC

 **Neo4j**



**redis**

# Key-Value Stores

- Come from a research paper by Amazon (Dynamo)
  - Global Distributed Hash Table (Key-Value Stores)
- A simple **hash table** (map), primarily used when all accesses to the database are via **primary key**
  - **key-value** mapping
- In RDBMS world: A table with two columns:
  - ID column (primary key)
  - DATA column storing the value (unstructured binary large object)

# Key-Value Stores - Operations

- Key-value stores support
  - ***put***(key, value): used to store values with an associated key,
  - ***get***(key): which retrieves the stored value associated with the specified key
  - ***delete***(key): remove the key and its associated value
  - ***scan***(from, to): range scan
- Some systems also support ***range queries*** on key values

# Key-Value Stores - Architecture

## 1. Embedded systems

- a. the system is a **library** and the DB runs **within** your system

## 2. Large-scale **Distributed** stores

- a. **distributed hash table** (DHT)

# Key-Value Stores

- Why?
  - Simple Data Model: Hash Table is well-studied
  - Good Scalability: Small System Cost, via good look-up locality and caching
- Why not?
  - Poor to complex (interconnected) data

# Key-Value Stores - Vendors



Ranked list: <http://db-engines.com/en/ranking/key-value+store>

# Document Stores

- Basic concept of data: **Document**
- Documents are **self-describing** pieces of data
  - Hierarchical tree data **structures**
  - Nested associative arrays (maps), collections, scalars
  - XML, JSON (JavaScript Object Notation), BSON, ...
- Documents in a **collection** should be “similar”
  - Their **schema** can differ
- **Documents** stored in the **value** part of key-value
  - Key-value stores where the values are **examinable**
  - Building search **indexes** on various **keys/fields**

# Document Stores - Example

```
key=3 -> { "personID": 3,  
            "firstname": "Martin",  
            "likes": [ "Biking", "Photography" ],  
            "lastcity": "Boston",  
            "visited": [ "NYC", "Paris" ] }  
  
key=5 -> { "personID": 5,  
            "firstname": "Pramod",  
            "citiesvisited": [ "Chicago", "London", "NYC" ],  
            "addresses": [  
                { "state": "AK",  
                  "city": "DILLINGHAM" },  
                { "state": "MH",  
                  "city": "PUNE" } ],  
            "lastcity": "Chicago" }
```

# MongoDB

- **humongous** => Mongo
- Data are organized in **collections**. A collection stores a set of **documents**.
- Collection like table and document like record
  - but: each document can have a different set of attributes even in the same collection
  - **Semi-structured** schema!
- Only requirement: every document should have an “**\_id**” field

# Example MongoDB

```
{      "_id": ObjectId("4efa8d2b7d284dad101e4bc9") ,  
      "Last Name": " Cousteau" ,  
      "First Name": " Jacques-Yves" ,  
      "Date of Birth": "06-1-1910" } ,  
  
{      "_id": ObjectId("4efa8d2b7d284dad101e4bc7") ,  
      "Last Name": "PELLERIN" ,  
      "First Name": "Franck" ,  
      "Date of Birth": "09-19-1983" ,  
      "Address": "1 chemin des Loges" ,  
      "City": "VERSAILLES" }
```

# MongoDB - Features

- **JSON**-style documents
  - actually uses **BSON** (JSON's binary format)
- replication for high availability
- auto-sharding for scalability
- document-based queries
- can create an index on any attribute
- for faster reads

# MongoDB vs RDBMS

<b>RDBMS</b>	<b>MongoDB Equivalent</b>
database	database
table	collection
row	document
attributes	fields (field-name:value pairs)
primary key	the ' <i>_id</i> ' field, which is the key associated with the document

# Relationships in MongoDB

**Two options:**

1. store references to other documents using their `_id` values
2. embed documents within other documents

# Example

Here is an example of embedded relationship:

```
{  
  "_id": ObjectId("52ffc33cd85242f436000001"),  
  "contact": "987654321",  
  "dob": "01-01-1991",  
  "name": "Tom Benzamin",  
  "address": [  
    {  
      "building": "22 A, Indiana Apt",  
      "pincode": 123456,  
      "city": "Los Angeles",  
      "state": "California"  
    },  
    {  
      "building": "170 A, Acropolis Apt",  
      "pincode": 456789,  
      "city": "Chicago",  
      "state": "Illinois"  
    }  
  ]  
}
```

And here an example of reference based

```
{  
  "_id": ObjectId("52ffc33cd85242f436000001"),  
  "contact": "987654321",  
  "dob": "01-01-1991",  
  "name": "Tom Benzamin",  
  "address_ids": [  
    ObjectId("52ffc4a5d85242602e000000"),  
    ObjectId("52ffc4a5d85242602e000001")  
  ]  
}
```

# MongoDB - Queries

- **Query** language expressed via **JSON**
- clauses: `where`, `sort`, `count`, `sum`, etc.

SQL:        `SELECT * FROM users`

MongoDB: `db.users.find()`

- `SELECT * FROM users WHERE personID = 3`
- `db.users.find( { "personID": 3 } )`
- `SELECT firstname, lastcity FROM users WHERE personID = 5`
- `db.users.find( { "personID": 5 }, {firstname:1, lastcity:1} )`

# Document Stores - Vendors



**MS Azure  
DocumentDB**

Ranked list: <http://db-engines.com/en/ranking/document+store>

# Column-Family Stores

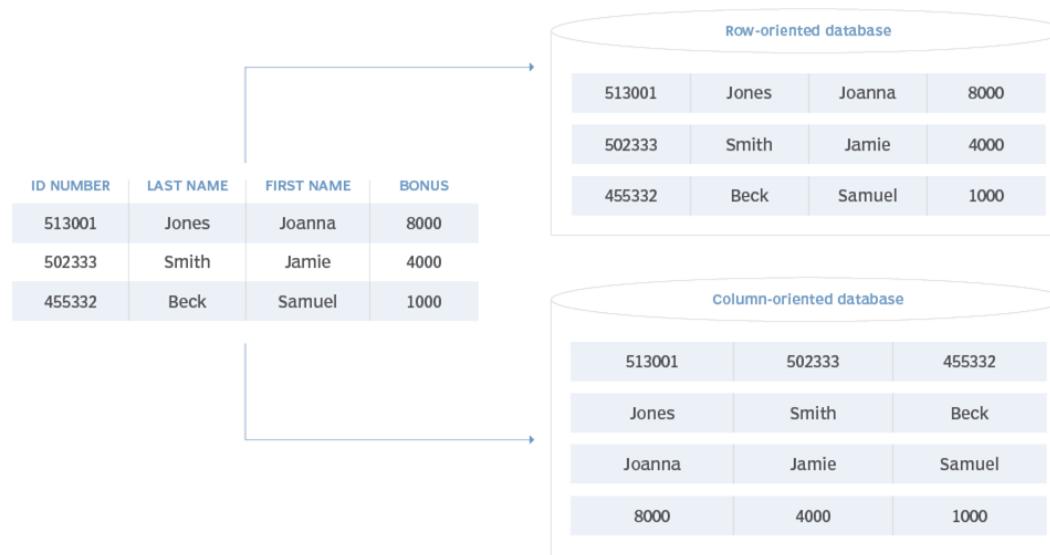
- Origin from Google's BigTable
- Also known as *wide-column* or *columnar*
- Data model: **rows** that have **many columns** associated with a **row key**
- Column families are groups of related data (columns) that are often **accessed together**

# Column-Family Stores - Main Idea

- Each table tends to have many attributes (from thousands ~ millions)
- In most applications (in OLAP) we are only interested in a **few** attributes
- Traditional raw-based
  - Store each record in a sequential file
  - We need to read the **whole** record to access only one attribute
- Column-based
  - Store the data by putting the same attribute in a sequential file
  - Faster access and better compression

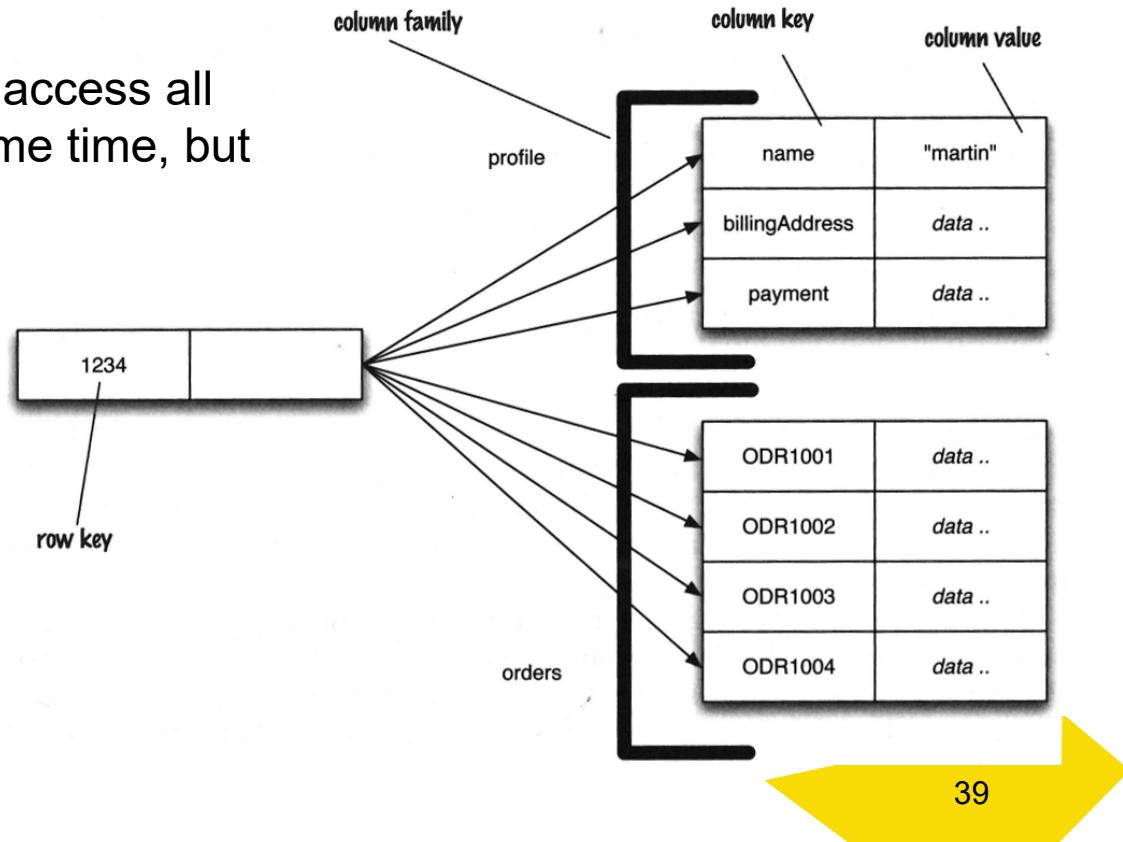
# Column-Family Stores - Example 1

## Column-oriented vs. row-oriented databases



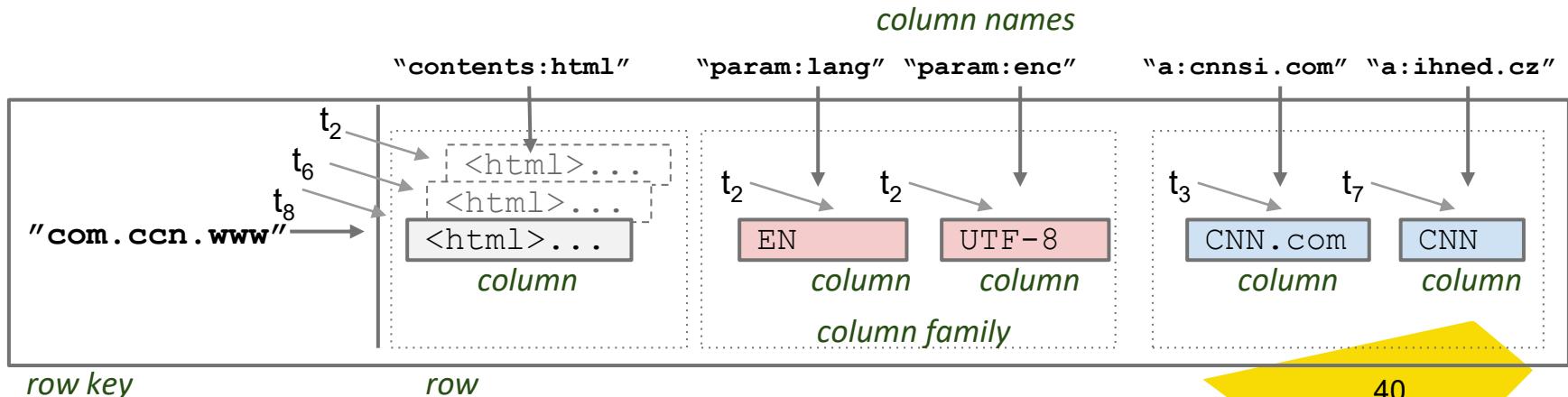
# Column-Family Stores - Example 2

For a **customer** we typically access all **profile** information at the same time, but not customer's **orders**.



# BigTable

- Google's BigTable
  - Drives MapReduce, and the following: Apache Hadoop, Hadoop File System (HDFS), HBase, Apache Cassandra
- 2008: Google published the Bigtable Paper
  - “BigTable = sparse, distributed, persistent, multi-dimensional sorted map indexed by (row\_key, column\_key, timestamp)”



# Column-Family Stores

- Why?
  - Optimized for OLAP
  - Semi-Structured Data: Each column can define its own schema
- Why not?
  - Not good for
    - OLTP
    - Incremental Data Loading
    - Row-specific Queries

# Column-Family Stores - Vendors



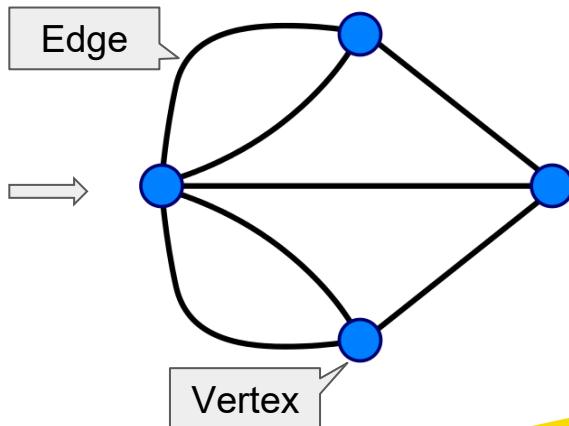
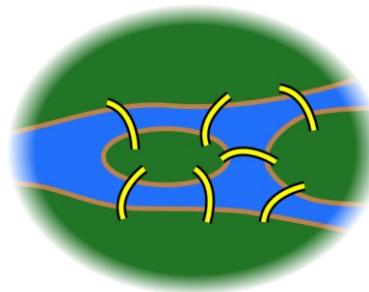
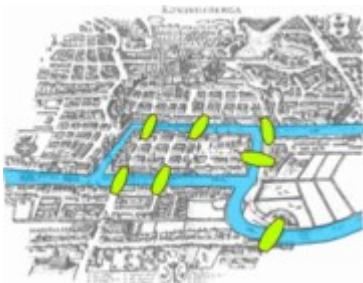
HYPERTABLE



Ranked list: <http://db-engines.com/en/ranking/wide+column+store>

# Graph Data Structure

- A **graph** is a structure in mathematics (graph theory)
- Famous problem: Seven Bridges of Königsberg
- Optimised for handling highly connected data

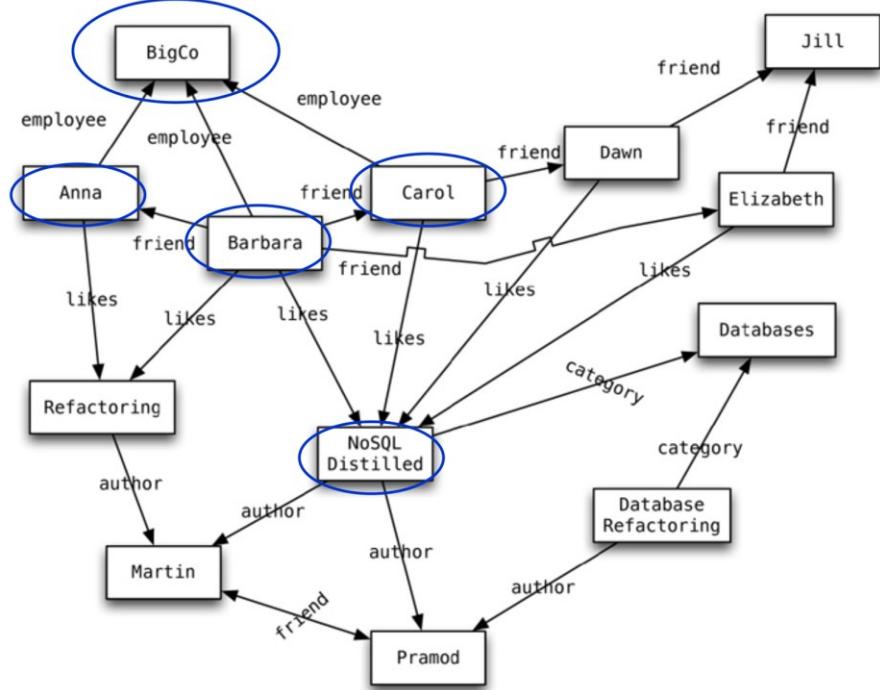


# Graph Database

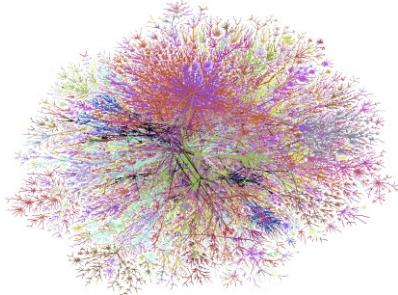
## Data Model

- Vertices (Nodes) -> Entities
- Edges -> Relations

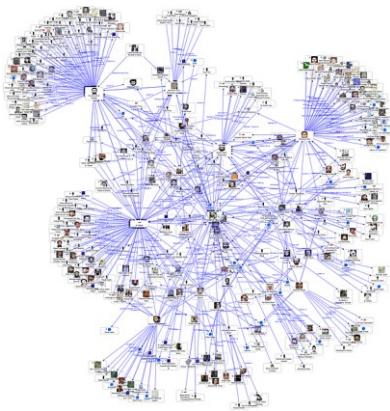
*Are we going to learn ER model again?*



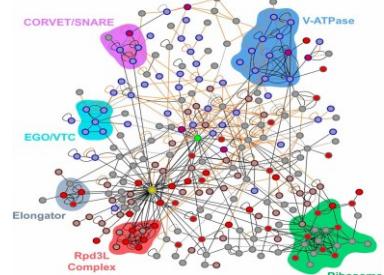
# Graphs are Everywhere!



Internet



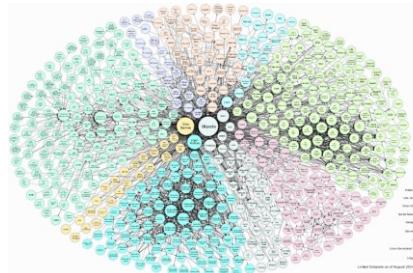
Social Networks



Biological Networks



Road Networks



Knowledge Graphs

# Graphs are Large!

<u>#Vetices</u>	<u>Ratio</u>
<10K	17.3%
10K-100K	17.3%
100K-1M	15.0%
1M-10M	13.4%
10M-100M	15.7%
<b>&gt;100M</b>	<b>21.3%</b>

<u>#Edges</u>	<u>Ratio</u>
<10K	17.8%
10K-100K	17.1%
100K-1M	10.1%
1M-10M	6.9%
10M-100M	16.3%
100M-1B	16.3%
<b>&gt;1B</b>	<b>15.5%</b>

<u>#Bytes</u>	<u>Ratio</u>
<100MB	19.0%
100MB-1G	15.7%
1G-10G	20.7%
10G-100G	14.1%
100G-1T	16.5%
<b>&gt;1T</b>	<b>14.0%</b>



>1 trillion  
connections



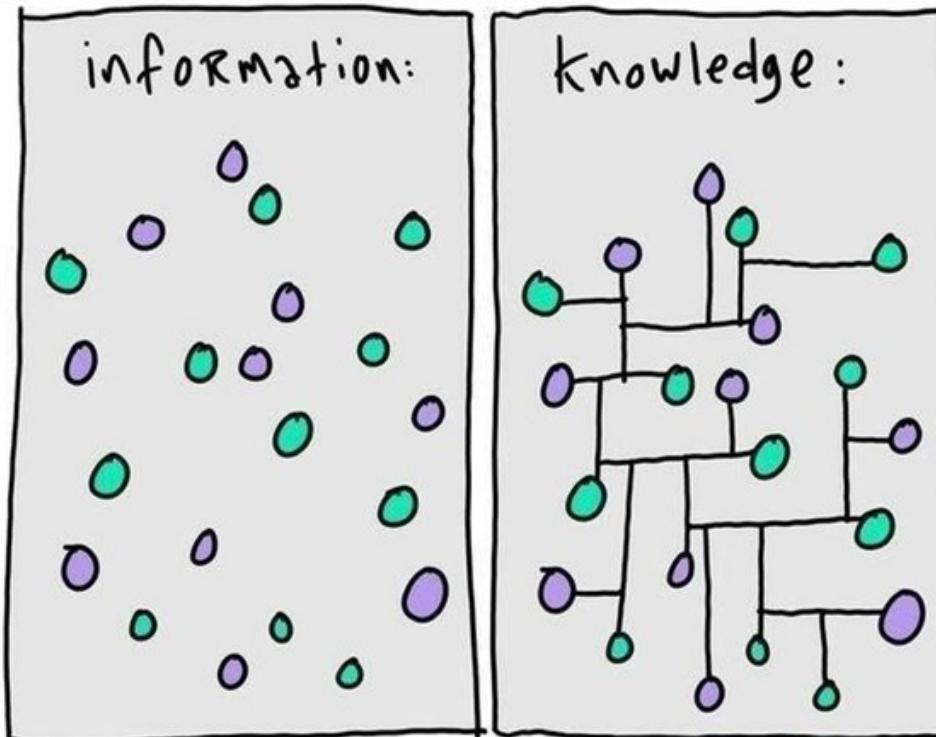
>60 trillion  
URLs



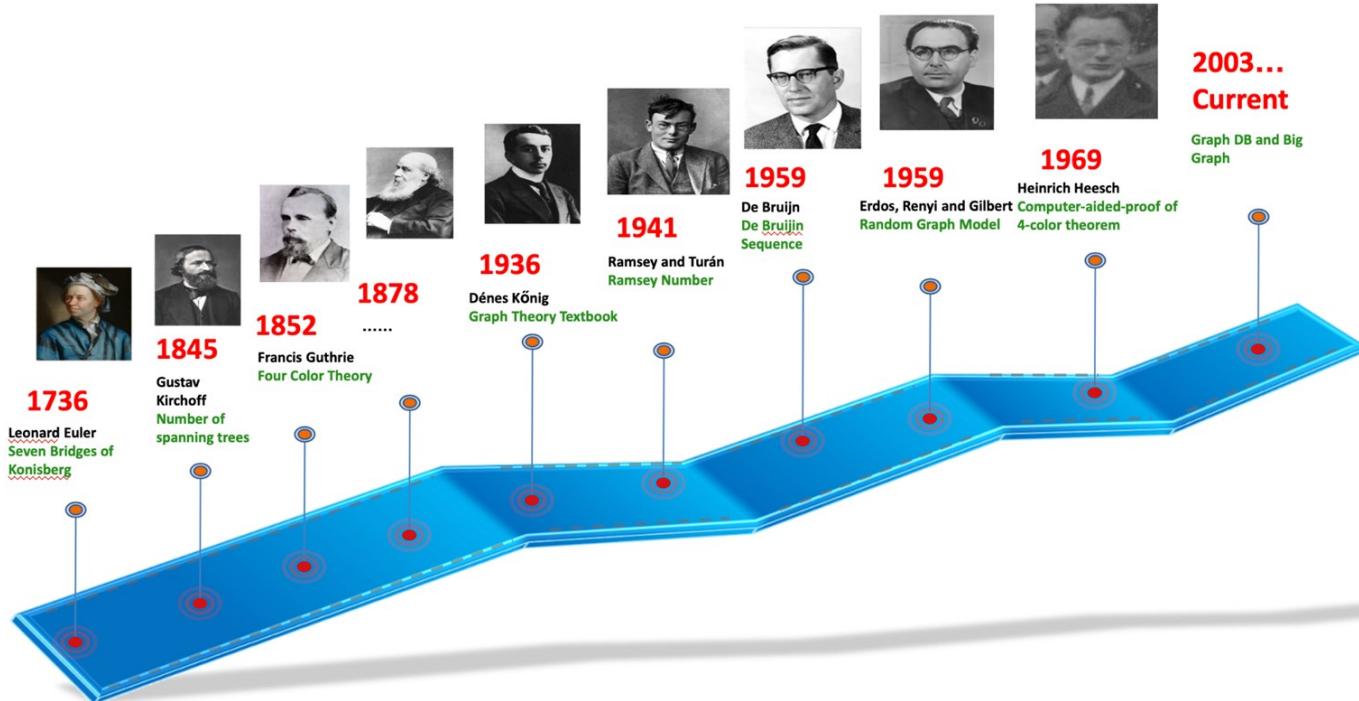
>60 billion  
edges every 30  
days

Small to medium sized companies

# Information vs Knowledge

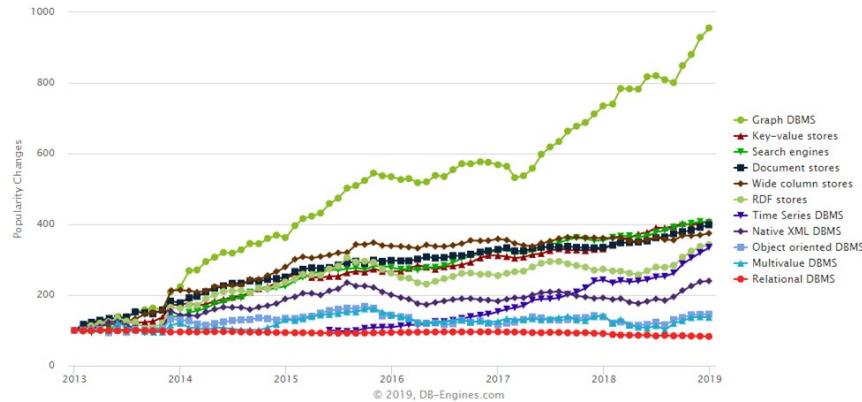


# History



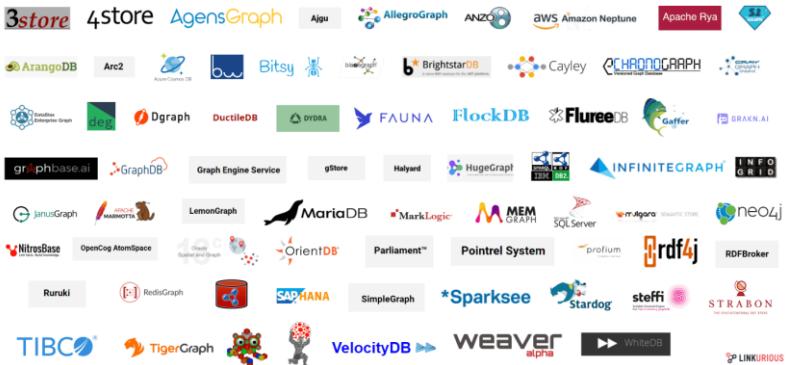
# Graph DBMS Landscape

Complete trend, starting with January 2013



DBMS popularity trend by database model between 2013 and 2019 – DB-Engine

The graph database ecosystem 2019



The graph database landscape in 2019

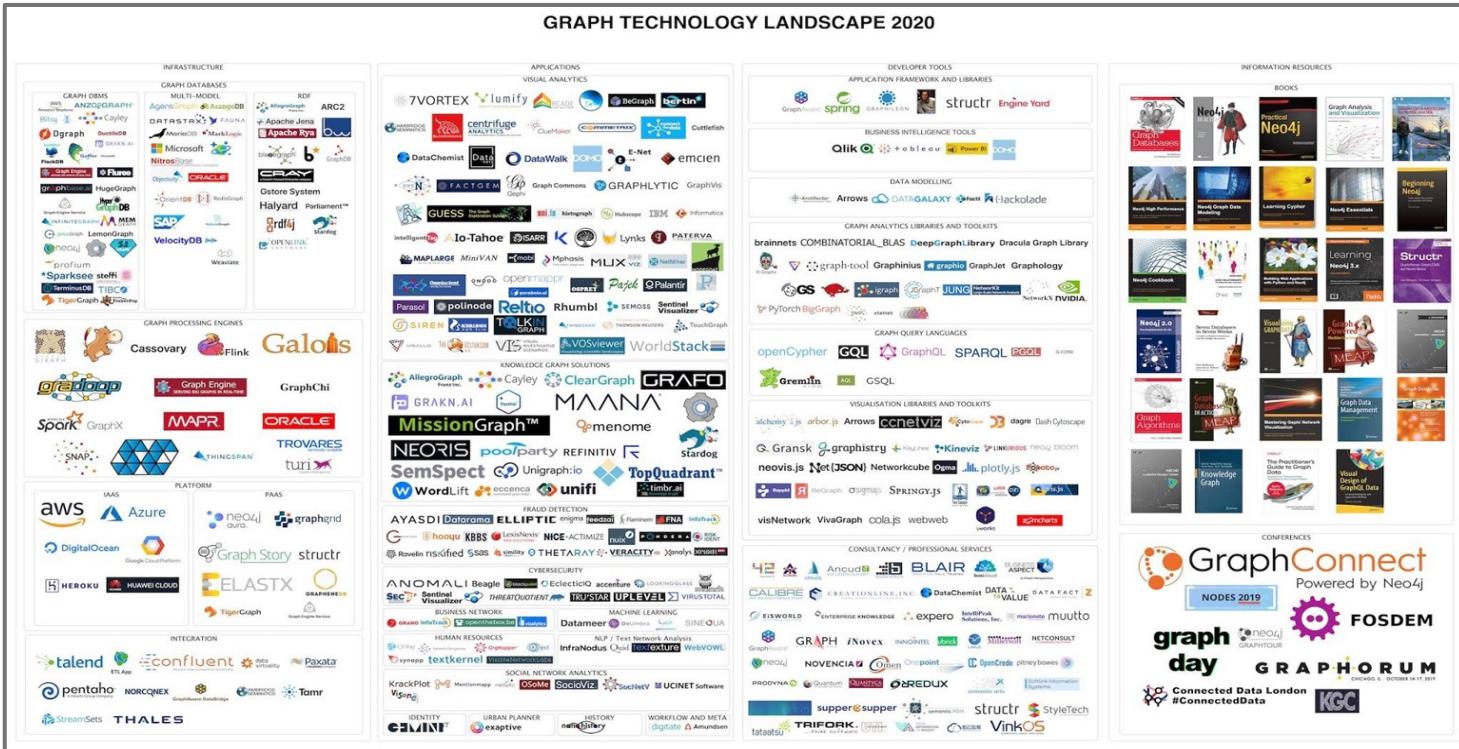
# Advantages

- **Performance**
  - Traditional Joins are inefficient
  - Billion-scale data are common, e.g., Facebook social network , Google web graph
- **Flexibility**
  - Real-world entities may not have a fixed schema. It is not feasible to design 1000 attributes for a table.
  - Relationships among entities can be arbitrary. It is not feasible to use 1000 tables to model 1000 types of relationships.
- **Agility**
  - Business requirements changes over time
  - Today's development practices are agile, test-driven

# Applications

- **Social Network:** Facebook, Twitter, LinkedIn ...
- **E-commerce:** eBay, Amazon, Alibaba ...
- **Banking:** JPMorgan, Citi, UBS ...
- **Telecom:** Verizon, Orange, AT&T ...
- **IoT:** nest
- **Search Engine:** Google
- **Navigation:** Google Maps
- **Bioinformatics:** DNAlexue
- ...

# Graph Technology Landspace



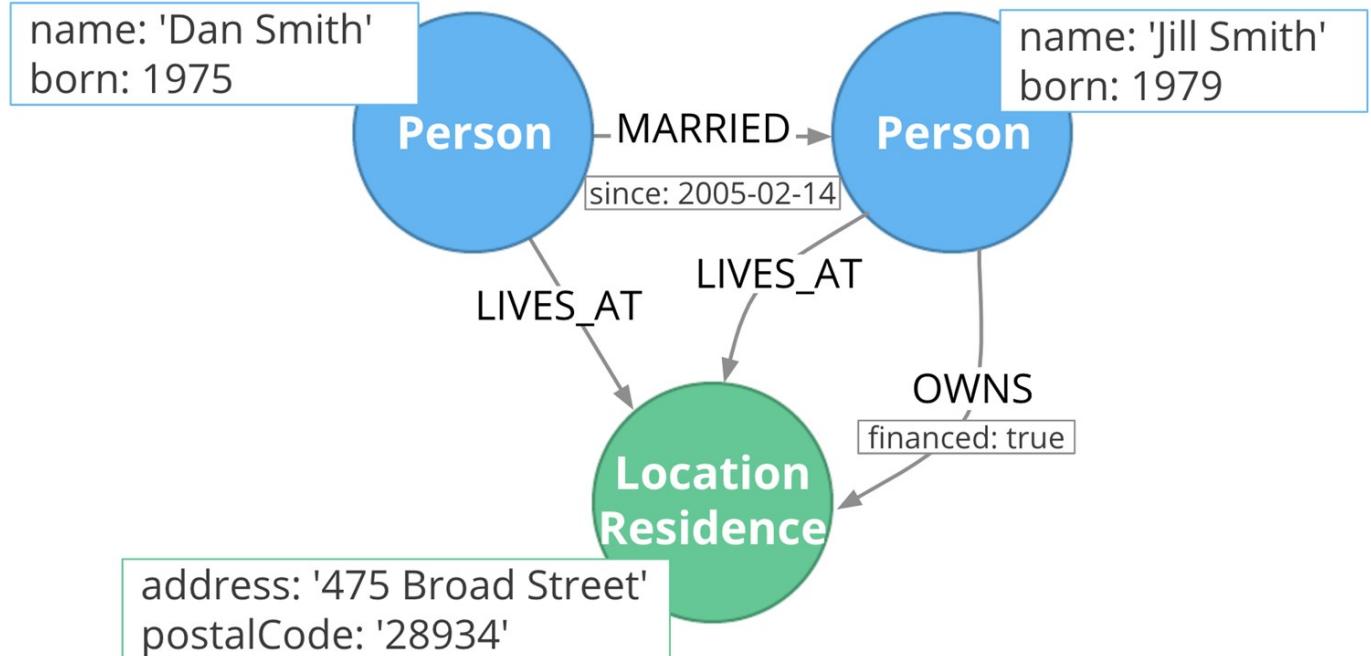
# Neo4j

- The most popular Graph Database at present
- *Cypher* query language
- Developed in Java and open-source
- Resources:
  - [Neo4j Cypher Manual](#)
  - [Neo4j Developer Resources](#)



# Property Graph Model

- **Nodes**  
(Entities)
- **Relationships**
- **Properties**
- **Labels**



# Graph DB Data Modeling

- Flexible & adaptive schema compared to RDBMs
- What you sketch = what you store in the database

# Neo4j vs Relational Model

- Retains **ACID** transaction properties
- Foreign keys not necessary as they are represented as relationships
- Relationships are stored/represented per relational record/row instead of per table

# Neo4j vs Relational Model - Example

Student

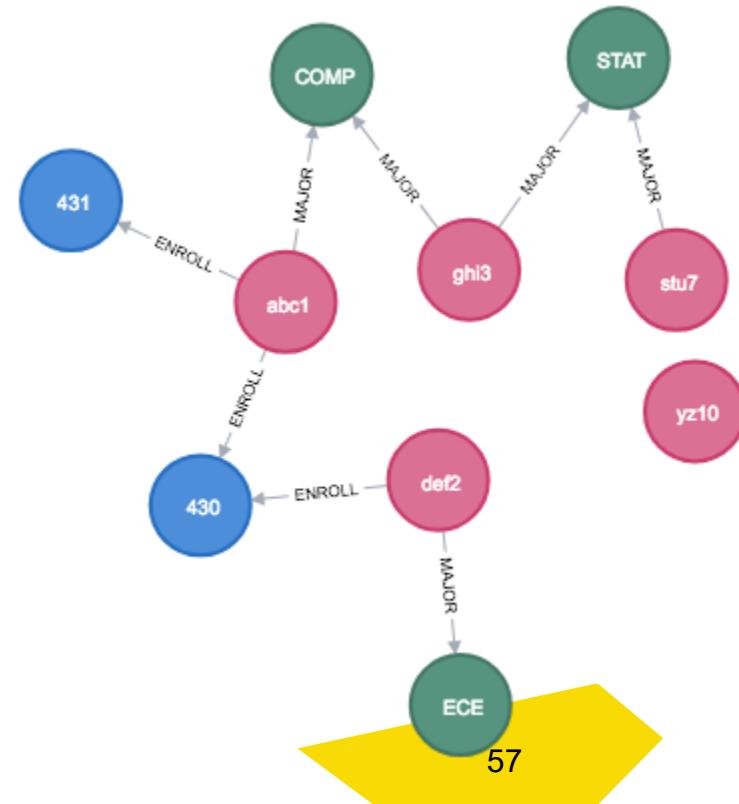
netId	FirstName
abc1	Albert
def2	Danielle
ghi3	Gary
stu7	Sandeep
yz10	Yusin

Enrolls

netId	Course
abc1	COMP 430
def2	COMP 430
abc1	COMP 431

Majors

netId	Major
ghi3	STAT
ghi3	COMP
abc1	COMP
def2	ECE
stu7	STAT



# Property or Node?

- **Dept** for a course could be a property of course or a node
- How to decide? It depends
  - Standardization of terms
    - How often will it be updated?
  - Searchability
  - Readability
  - Reusability

# MongoDB vs Neo4j

```
Student
{
  "_id": 5,
  netId: "def2",
  firstName: "Danielle",
  college: "Baker",
  major: "ECE",
  enrolls: "COMP 430"
}
```

```
Course
{
  "id": 12198,
  num: "430",
  dept: "COMP",
  title: "Intro to Database Systems"
}
```



Student <id>: 19 firstName: Danielle netId: def2

# Choosing a DBMS

- Efficient data storage
  - Structure (e.g. networks)
  - Sparsity
- Performance
  - Types of queries / analysis
  - Need for visualization
  - Built in algorithms or tools
  - Reads vs. Writes
  - Quantity of data
- Importance
  - Objects
  - Relationships

# When to use a graph database?

- If...

- Your data has many M-M relationships

Relational (A)		Graph (B)	
----------------	---	-----------	---

- Care about referential integrity

Relational		Graph	
------------	---	-------	--

- You highly value the relationships of the data

(especially when you may consider the relationship to be more important than the elements themselves)

Relational		Graph	
------------	--	-------	---

- ...case by case question

# Which to use?

1. ...if you were given a set of well-structured data
2. ...if you are scraping data from the Internet
3. ...if you are storing tax info
4. ...if the dataset is extremely large
5. ...if you are trying to build a friend network

A	Relational
B	NoSQL - Mongo
C	NoSQL – Neo4j
D	It depends

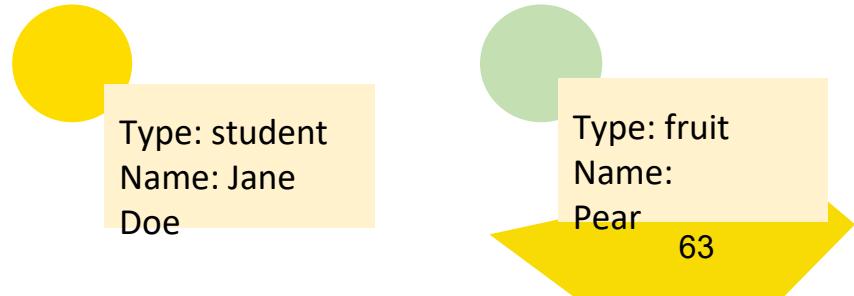
# Nodes and Properties

- Node

- Typically used to represent entities
- Has zero or more relationships
- Each node is an **instance** of an entity

- Property

- Key/value pairs
- Belong to nodes and relationships

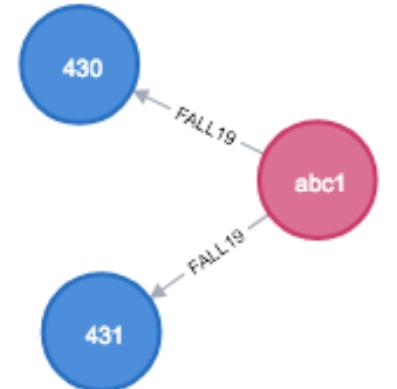
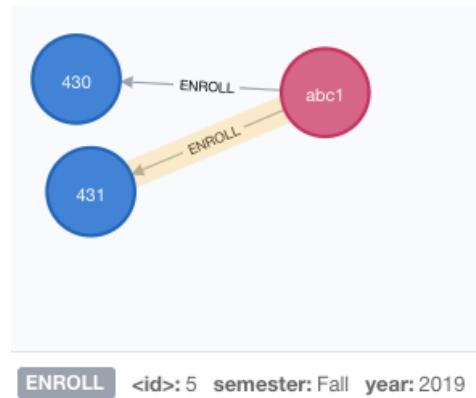


# Property Types

- **Number** (*Integer* and *Float* specifically)
- **String**
- **Boolean**
- **Temporal type** - *Date*, *Time*, *LocalTime*, *DateTime*, *LocalDateTime* and *Duration*

# Relationships

- Edge that connects nodes
- Relationships must have a **direction** and a **type**
- Can have properties



# Questions

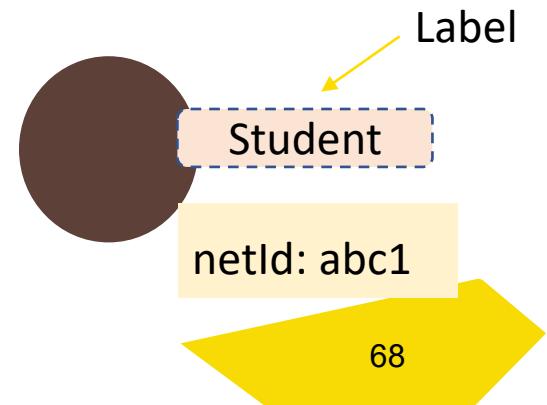
1. What are some examples of relationships that are directional?
2. What are some examples of relationships that are non-directional?

# Questions

1. What are some examples of relationships that are directional?
  - Think of **Twitter**
2. What are some examples of relationships that are non-directional?
  - Thinks of **Facebook**

# Labels

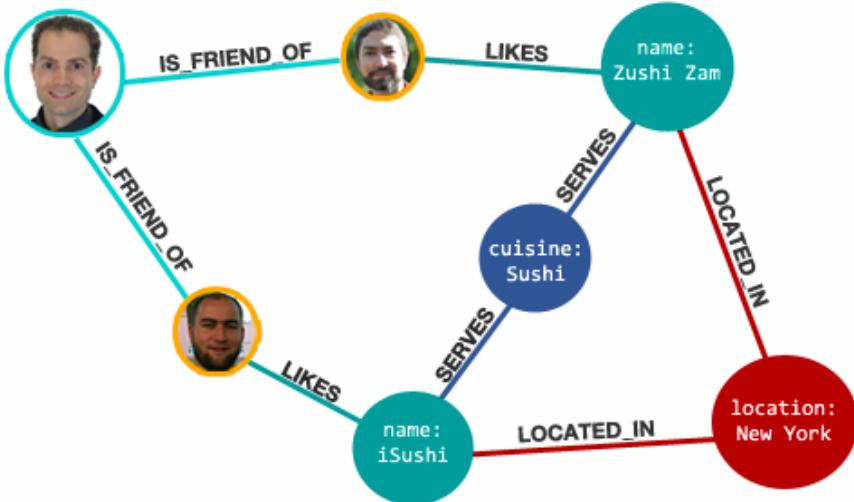
- Grouping mechanism for nodes
- Used to define constraints and/or indexes
- Faster lookup compared to checking a property



# T/F Questions

1. There's a 1-1 mapping from the relational model to a graph database
2. Each node must have the same properties as all other nodes
3. Relationships cannot have properties

# Cypher Query Language



An example of Cypher:

Find Sushi restaurants in New York that Philip's friends like

```
MATCH (person:Person)-[:IS_FRIEND_OF]->(friend),
      (friend)-[:LIKES]->(restaurant:Restaurant),
      (restaurant)-[:LOCATED_IN]->(loc:Location),
      (restaurant)-[:SERVES]->(type:Cuisine)

WHERE person.name = 'Philip'
AND loc.location = 'New York'
AND type.cuisine = 'Sushi'

RETURN restaurant.name, count(*) AS occurrence
ORDER BY occurrence DESC
LIMIT 5
```

# Cypher Introduction

- Declarative graph query language
- Shares many keywords and query structures with SQL
- Comments can be added with “//” or “/\* \*/”
- Case *insensitive* except for
  - Labels
  - Property keys
  - Relationship types

# Cypher Patterns

- Used to describe the shape of what you are looking for
- Node: ( )
- Relationship: -, ->, <-
- Relationship identifier: [ ]
- Labels :<LabelName>
- Variables: n, node, foo
- Not only for querying, also for creating new nodes, relations etc.

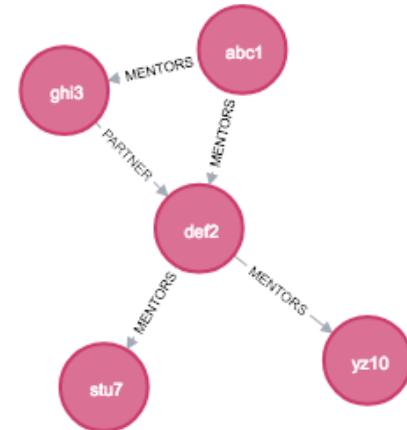
# Cypher Patterns 1

- Any directional relationship

```
MATCH (n:Student) -->(m:Student)  
RETURN n, m;
```

Students with a relation with another Student

n and m are variables  
Student is a label



"n.name"	"m.name"
"Gary"	"Danielle"
"Albert"	"Danielle"
"Albert"	"Gary"
"Danielle"	"Sandeep"
"Danielle"	"Yousef"

# Cypher Patterns 2

- Specific relationships

```
MATCH (n:Student) - [:MENTORS] -> (m:Student)  
RETURN DISTINCT n.netId;
```

netIds of students who mentor other students

- Note the text output
- Can return properties

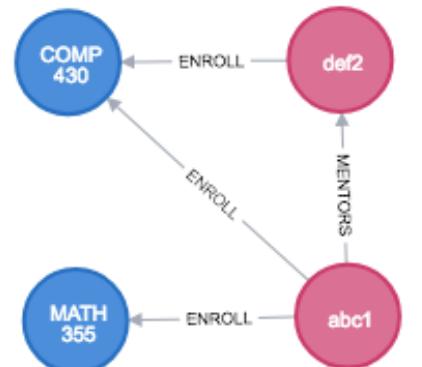
"n.netId"
"def2"
"abc1"

# Cypher Patterns 3

- Nodes with different labels

```
MATCH (n:Student) - [:ENROLL] -> (m:Course)  
RETURN n, m;
```

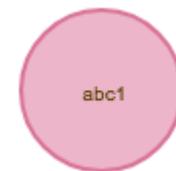
Students enrolled in courses



# Cypher clauses – MATCH/WHERE

- Find pattern (MATCH), then filter results (WHERE)

- WHERE is part of MATCH-WHERE clause
- can be replaced with OPTIONAL MATCH, WITH for future constraints



- Find node with firstname 'Albert'

```
MATCH (n {name: 'Albert'})  
RETURN n;
```

- Also

```
MATCH (n)  
WHERE n.name = 'Albert'  
RETURN n;
```

```
{  
  "name": "Albert",  
  "netId": "abc1"  
}
```

# Cypher clauses – RETURN

- RETURN is equivalent to SELECT in SQL, it returns the specified nodes or properties
- RETURN netIds of students with relationships with other students

```
MATCH (a:Student) -->(otherNode:Student)  
RETURN a.netId, otherNode.netId;
```

"a.netId"	"otherNode.netId"
"jkl4"	"def2"
"ghi3"	"def2"
"abc1"	"def2"
"abc1"	"ghi3"
"def2"	"stu7"
"def2"	"yz10"

# Types of Graph Queries

- **Graph Pattern Matching**
  - Given a graph pattern, find subgraphs in the database graph that match the query.
  - Can be augmented with other (relational-like) features, such as *projection*.

```
MATCH (p:Person)-[:LIKES]->(:Language {name = "SQL"})  
RETURN p.name
```
- **Graph Navigation**
  - A flexible querying mechanism to navigate the topology of the data.
  - Called path queries, since they require to navigate using paths (potentially variable length).

```
MATCH (p:Person)-[:KNOWS*1..2]->(:Person {name = "Alice"})  
RETURN p.name
```

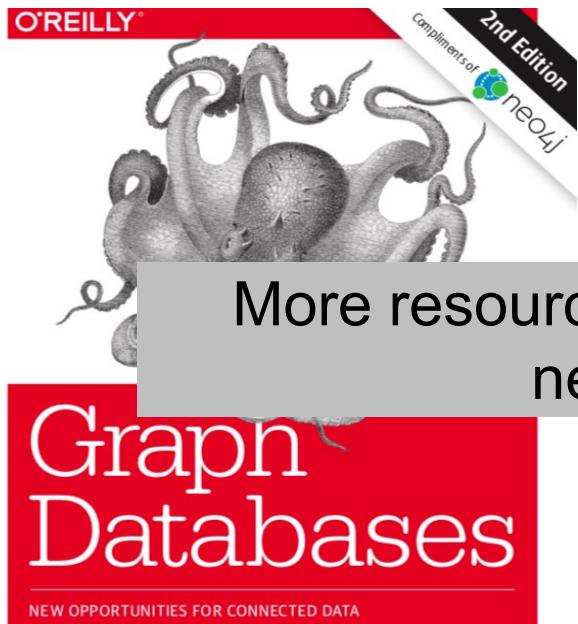
# More Cypher clauses

- CREATE
- DELETE
- SET
- ORDER BY
- LIMIT
- WITH
- Aggregations:
  - COUNT
  - COLLECT
  - SUM
  - ...
- ...

# Graph Algorithms

- The real power of graph databases
- Can save huge amounts of programming effort
- Include
  - Centrality - node importance
  - Community detection – node connectivity and partitions
  - Path finding – routes through the network
  - Similarity – of nodes
  - Link prediction – closeness of nodes
- <https://neo4j.com/docs/graph-data-science/current/>

# Neo4j



More resources can be found on  
[neo4j.com](http://neo4j.com)



Ian Robinson,  
Jim Webber & Emil Eifrem

# Related Courses at UNSW CSE

- COMP9312: [Data Analytics for Graphs](#)
- COMP9313: [Big Data Management](#)

# Learning Outcome

- NoSQL vs RDBMS
- Data Models
  - Key-value
  - Document
  - Column-family
  - Graph