DSECL ZG 522: Big Data Systems

Session 10: NoSQL Databases

Janardhanan PS

Professor

janardhanan.ps@wilp.bits-pilani.ac.in

Topics for today

- **NoSQL Introduction**
- Classification
- Examples
  - MongoDB
  - Cassandra
  - GraphDBs: Neo4J and Tinkerpop

Getting Started – Watch the webinars by Janardhanan PS

1. TechGig webinar – Evolution of NoSQL Databases (2317 Views - 15 June, 2021)
   https://www.techgig.com/webinar/Evolution-of-NoSQL-Databases-2001
2. CSI Weekly Talk Series - Introduction to Neo4j, the graph database (18 Feb 2021)
   https://www.youtube.com/watch?v=ImYDZkRrctw

What is NoSQL Database ?

- NoSQL databases, also known as "Not Only SQL" databases, are a type of database that do not use traditional SQL (Structured Query Language) for storing and manipulating data
- They are designed to handle large amounts of unstructured, semi-structured, or polymorphic data and are often used for big data, real-time data processing, and cloud-based applications
- NoSQL databases use a distributed architecture, allowing them to scale horizontally across multiple servers or nodes, making them ideal for handling high levels of concurrency and data volume

NoSQL Use Cases

- **Big data:** NoSQL databases are perfect for handling large amounts of data since they can scale horizontally across multiple servers or nodes and handle high levels of concurrency
- **Real-time data processing:** They are often used for real-time data processing since they can handle high levels of concurrency and support low latency
- **Cloud-based applications:** NoSQL databases are perfect for cloud-based applications since they can easily scale and handle large amounts of data in a distributed environment
- **Content management:** NoSQL databases are often used for content management systems since they can handle large amounts of data and support flexible data models
- **Social media:** NoSQL databases are often used for social media applications since they can handle high levels of concurrency and support flexible data models
- **Internet of Things (IoT):** These databases are often used for IoT applications since they can handle large amounts of data from a large number of devices and handle high levels of concurrency
- **E-commerce:** They are often used for e-commerce applications since they can handle high levels of concurrency and support flexible data models

Why NoSQL (1)

- RDBMS meant for OLTP systems / Systems of Record
  - Strict consistency and durability guarantees (ACID) over multiple data items involved in a transaction
  - But they have scale and cost issues with large volumes of data, distributed geo-scale applications, very high transaction volumes
- Typical web scale systems do not need strict consistency and durability for every use case
  - Social networking
  - Real-time applications
  - Log analysis
  - Browsing retail catalogs
  - Reviews and blogs
  - …

Why NoSQL (2)

- RDBMS ensure uniform structure and modelling of relationships between entities
- A class of emerging applications need granular and extreme connectivity information modelled between individual semi-structured data items. This information needs to be also queried at scale without large expensive joins.
  - Connectivity between users in a social media application: How many friends do you have between 2 hops ?
  - Connectivity between companies in terms of domain, technology, people skills, hiring : Useful for skills acquisition, M&A etc.
  - Connectivity between IT network devices: Useful for troubleshooting incidents

What is NoSQL ?

- Coined by Carlo Strozzi in 1998
  - ✓Lightweight, open source database without standard SQL interface
- Reintroduced by Johan Oskarsson in 2009
  - ✓Non-relational databases
- Characteristics
  - ✓Not Only SQL
  - ✓Non-relational
  - ✓Schema-less
  - ✓Loosen consistency to address scalability and availability requirements in large scale applications
  - ✓Open source movement born out of web-scale applications
  - ✓Distributed for scale
  - ✓Cluster Friendly

Data model

- Supports rich variety of data : structured, semi-structured and unstructured
- No fixed schema, i.e. each record could have different attributes
- Non-relational - no join operations are typically supported
- Transaction semantics for multiple data items are typically not supported
- Relaxed consistency semantics - no support for ACID as in RDBMS
- In some cases can model data as graphs and queries as graph traversals

Choice between consistency and availability

- In a distributed database
  - ✓Scalability and fault tolerance can be improved through additional nodes, although this puts challenges on maintaining consistency (C).
  - ✓The addition of nodes can also cause availability (A) to suffer due to the latency caused by increased communication between nodes.
    - May have to update all replicas before sending success to client . so longer takes time and system may not be available during this period to service reads on same data item.
- Large scale distributed systems cannot be 100% partition tolerant (P).
  - ✓Although communication outages are rare and temporary, partition tolerance (P) must always be supported by distributed database

- In NoSQL, generally a choice between choosing either CP or AP of CAP
- RDBMS systems mainly provide CA for single data items and then on top of that provide ACID for transactions that touch multiple data items.

Classification of NoSQL DBs

- Key – value
  - ✓ Maintains a big hash table of keys and values
  - ✓ Example : Dynamo, Redis, Riak etc
- Document
  - ✓ Maintains data in collections of documents
  - ✓ Example : MongoDB, CouchDB etc
- Column
  - ✓ Each storage block has data from only one column
  - ✓ Example : Cassandra, HBase
- Graph
  - ✓ Network databases
  - ✓ Graph stores data in nodes
  - ✓ Example : Neo4j, HyperGraphDB, Apache Tinkerpop

NoSQL Databases - http://nosql-database.org/

DynamoDB

amazon web services™

cassandra

Characteristics

- Scale out architecture instead of monolithic architecture of relational databases
  - Cluster scale - distribution across 100+ nodes across DCs
  - Performance scale - 100K+ DB reads and writes per sec
  - Data scale - 1B+ docs in DB
- House large amount of structured, semi-structured and unstructured data
- Dynamic schemas
  - ✓ allows insertion of data without pre-defined schema
- Auto sharding
  - ✓ automatically spreads data across the number of servers
  - ✓ applications are not aware about it
  - ✓ helps in data balancing and failure from recovery
- Replication
  - ✓ Good support for replication of data which offers high availability, fault tolerance

Pros and Cons

Pros

- Cost effective for large data sets
- Easy to implement
- Easy to distribute esp across DCs
- Easier to scale up/down
- Relaxes data consistency when required
- No pre-defined schema
- Easier to model semi-structured data or connectivity data
- Easy to support data replication

Cons

- Joins between data sets / tables
- Group by operations
- ACID properties for transactions
- SQL interface
- Lack of standardisation in this space
  - Makes it difficult to port from SQL and across NoSQL stores
- Less skills compared to SQL
- Lesser BI tools compared to mature SQL BI space

SQL vs NoSQL

SQL	NoSQL
Relational database	Non relational, distributed databases
Pre-defined schema	Schema less
Table based databases	Multiple options: Key-Value, Document, Column, Graph
Vertically scalable	Horizontally scalable
Supports ACID properties	Supports CAP theorem
Supports complex querying	Relatively simpler querying
Excellent support from vendors	Relies heavily on community support

Vendors

- Amazon
- Facebook
- Google
- Oracle

DynamoDB

amazon web services

Amazon DocumentDB

Amazon Neptune

cassandra

APACHE HBASE

redis

ORACLE

NOSQL DATABASE

Topics for today

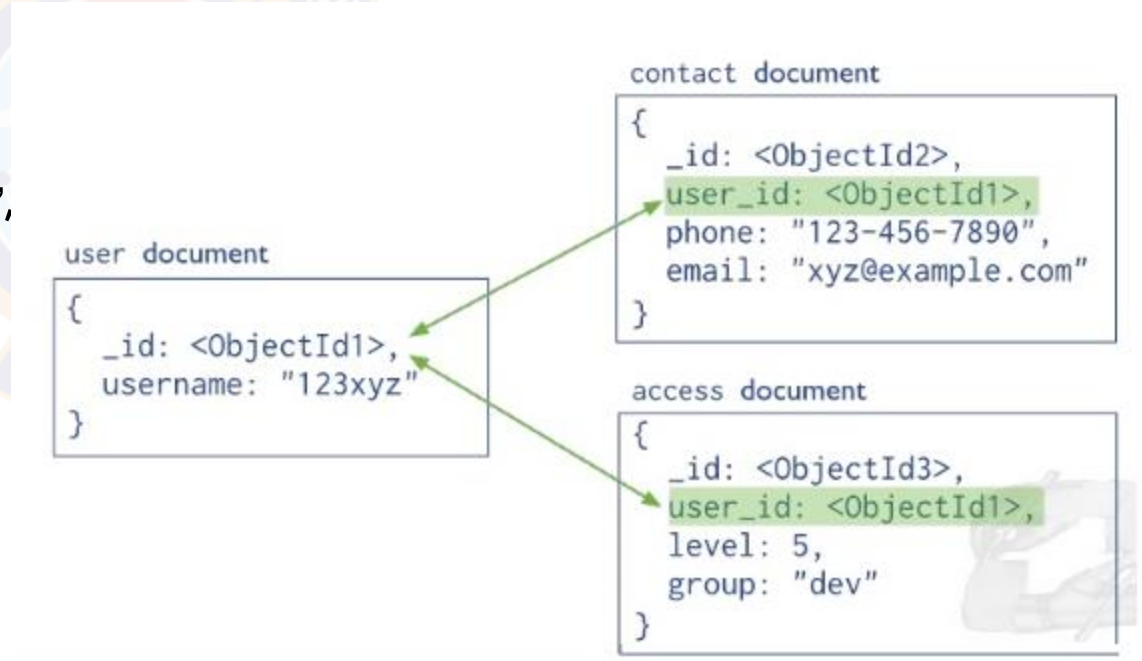
- NoSQL Introduction
- **Classification**
- Examples
  - Cassandra
  - Mongo
  - GraphDBs: Neo4J and Tinkerpop

Classification: Document-based

- Store data in form of documents using well known formats like JSON
- Documents accessible via their id, but can be accessed through other index as well
- Maintains data in collections of documents
- Example,
  - MongoDB, CouchDB, CouchBase

- Book document :

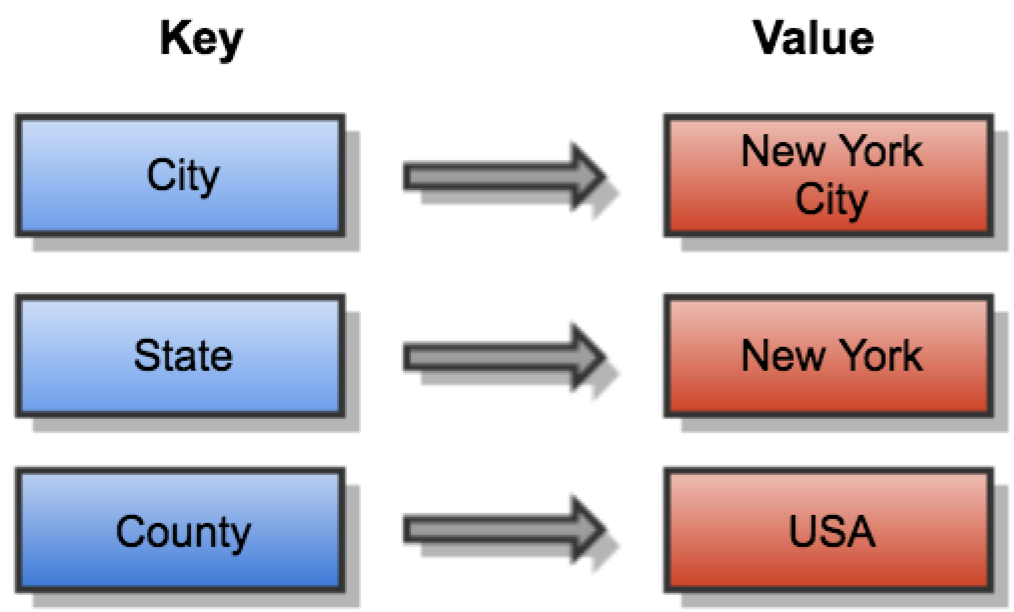
{
    “Book Title” : “Fundamentals of Database Systems”,
    “Publisher” : “Addison-Wesley”,
    “Authors”   : “Elmasri & Navathe”
    “Year of Publication” : “2011”
}



Classification: Key-Value store

- Simple data model based on fast access by the key to the value associated with the key
- Value can be a record or object or document or even complex data structure
- Maintains a big hash table of keys and values
- For example,
  - ✓ Dynamo, Redis, Riak

Key	Value
2014HW112220	{ Santosh,Sharma,Pilani}
2018HW123123	{Eshwar,Pillai,Hyd}



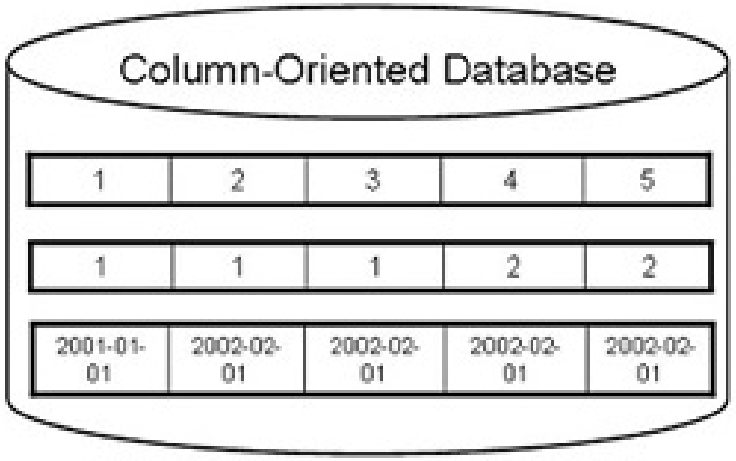
Classification: Column-based

- Partition a table by column into column families
- A part of vertical partitioning where each column family is stored in its own files
- Allows versioning of data values
- Each storage block has data from only one column
- Example,
  - ✓ Cassandra, Hbase

COLUMN FAMILIES

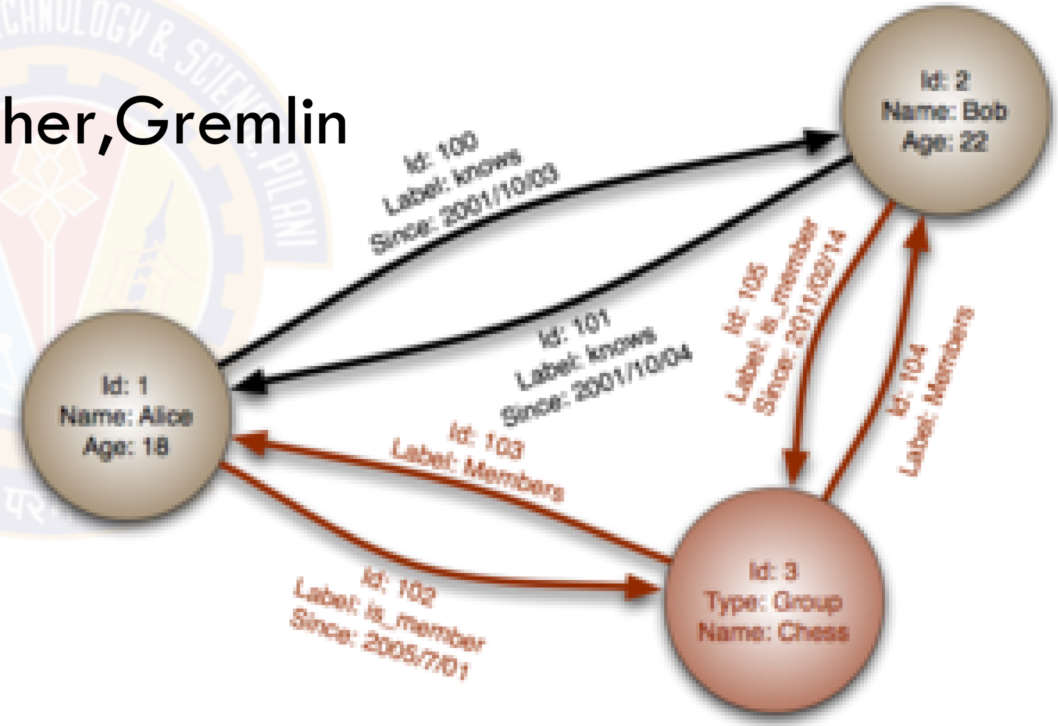
Row key	personal data		professional data	
empid	name	city	designation	salary
1	raju	hyderabad	manager	50,000
2	ravi	chennai	sr.engineer	30,000
3	rajesh	delhi	jr.engineer	25,000

Emp_no	Dept_id	Hire_date	Emp_ln	Emp_fn
1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue
4	2	2003-02-01	Stemle	Bill
5	2	1999-06-15	Aurora	Jack
6	3	2000-08-15	Jung	Laura



Classification: Graph based

- Data is represented as graphs and related nodes can be found by traversing the edges using the path expression
- aka network database
- Graph query languages, e.g. Cypher,Gremlin
- Example
  - ✓ Neo4J
  - ✓ HyperGraphDB
  - ✓ Apache TinkerPop



Topics for today

- NoSQL Introduction
- Classification
- Examples
    - ✓**MongoDB**
    - ✓Cassandra
    - ✓GraphDBs: Neo4J and Tinkerpop

MongoDB

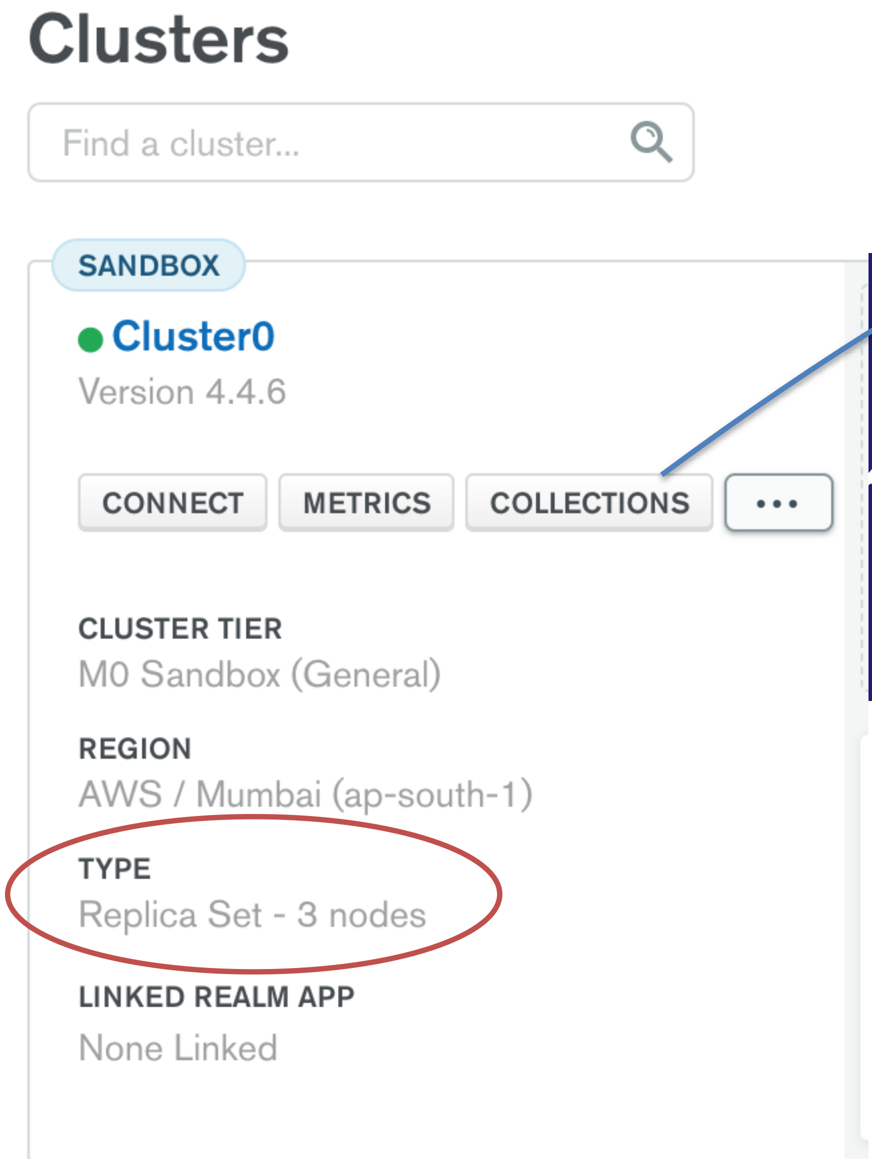
- Database is a set of collections
- A collection is like a table in RDBMS
- A collection stores documents
  - ✓ BSON or Binary JSON with hierarchical key-value pairs
  - ✓ Similar to rows in a table
  - ✓ Max 16MB documents stored in WiredTiger storage engine
- For larger than 16MB documents uses GridFS
  - ✓ Support for binary data
  - ✓ Large objects can be stored in ‘chunks’ of 255KB
  - ✓ Stores Meta-data in a separate collection
  - ✓ Does not support multi-document transactions
  - ✓ WiredTiger storage engine*

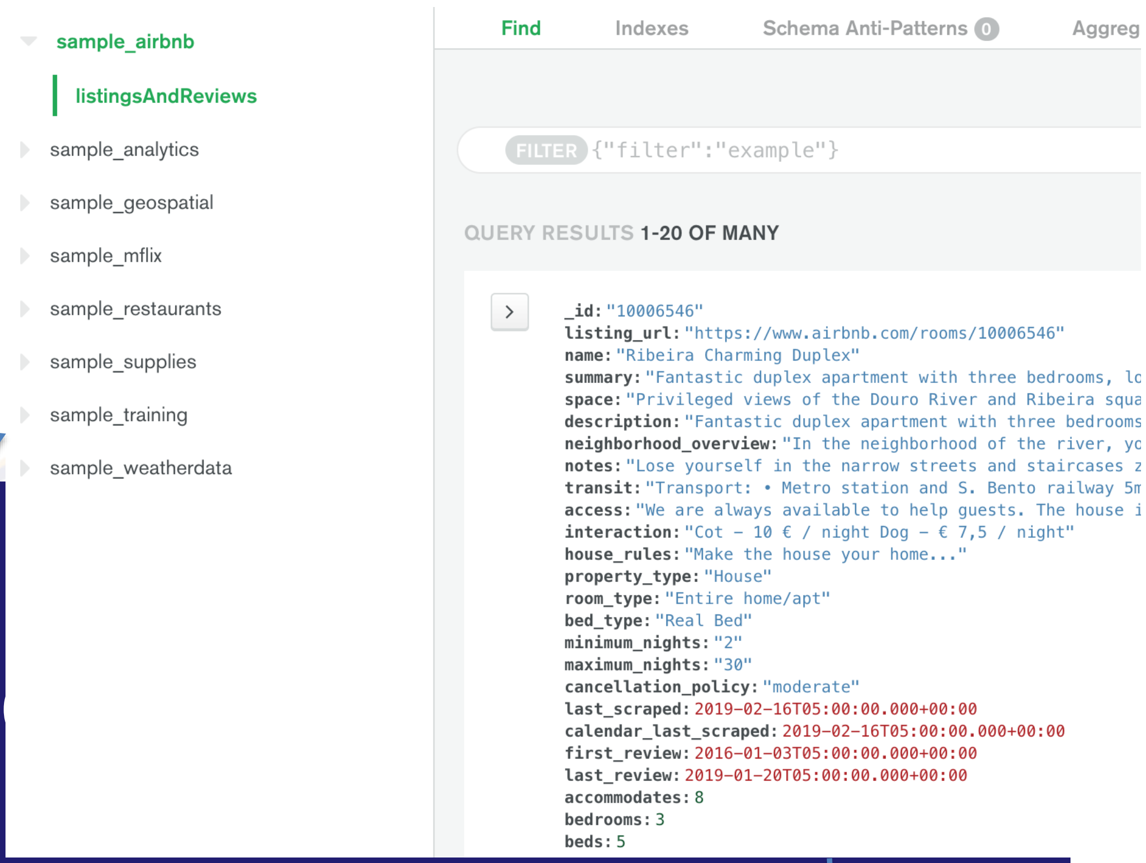
* https://docs.mongodb.com/manual/core/wiredtiger/

MongoDB

- Data is partitioned in shards
  - ✓ For horizontal scaling
  - ✓ Reduces amount of data each shard handles as the cluster grows
  - ✓ Reduces number of operations on each shard
- Data is replicated
  - ✓ Writes to primary in oplog. “write-concern” setting used to tweak write consistency.
  - ✓ Secondaries use oplog to get local copies updated
  - ✓ Clients usually read from primary but “read-preference” setting can tweak read consistency
- Data updates happen in place and not versioned / timestamped

cloud.mongodb.com





k to e

Get me top 10 beach front homes

```
# sort search results by score
s = collection.aggregate([

                { "$match": { "$text": { "$search": "beach front" } } },
                { "$project": { "name": 1, "_id": 0, "score": { "$meta": "textScore" } } },
                { "$match": { "score": { "$gt": 1.0 } } },
                { "$sort": { "score": -1}},
                { "$limit": 10}
        ])
```

MongoDB Data Example

Collection inventory

```
{
    item: "ABC2",
    details: { model: "14Q3", manufacturer: "M1 Corporation" },
    stock: [ { size: "M", qty: 50 } ],
    category: "clothing"
}

{
    item: "MNO2",
    details: { model: "14Q3", manufacturer: "ABC Compar
    stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 }, { size: "
    category: "clothing"
}
```

Document insertion

```
db.inventory.insert(
  {
    item: "ABC1",
    details: {model: "14Q3",manufacturer: "XYZ Company"},
    stock: [ { size: "S", qty: 25 }, { size: "M", qty: 50 } ],
    category: "clothing"
  }
)
```

Example of Simple Query

Collection **orders**

```
{
    _id: "a",
    cust_id: "abc123",
    status: "A",
    price: 25,
    items: [ { sku: "mmm", qty: 5, price: 3 },
             { sku: "nnn", qty: 5, price: 2 } ]
}
{
    _id: "b",
    cust_id: "abc124",
    status: "B",
    price: 12,
    items: [ { sku: "nnn", qty: 2, price: 2 },
             { sku: "ppp", qty: 2, price: 4 } ]
}
```

```
db.users.find(
    { status: "A" },
    { cust_id: 1, price: 1, _id: 0 }
)
```

selection

projection

In SQL it would look like this:

```
SELECT cust_id, price
  FROM orders
WHERE status="A"
```

Results

```
{
    cust_id: "abc123",
    price: 25
}
```

MongoDB Data Model

- JavaScript Object Notation (JSON) model
- *Database* = set of named *collections*
- *Collection* = sequence of *documents*
- *Document* = {attribute$_1$:value$_1$,...,attribute$_k$:value$_k$}
- *Attribute* = string (attribute$_i \neq$attribute$_j$)
- *Value* = primitive value (string, number, date, ...), or a document, or an *array*
- *Array* = [value$_1$,...,value$_n$]

- Key properties: hierarchical (like XML), no schema
  - ✓ Collection docs may have different attributes

MongoDB: MapReduce

```
> db.collection.mapReduce(
   function() {emit(key,value);},  //map function
   function(key,values) {return reduceFunction}, {   //reduce function
      out: collection,
      query: document,
      sort: document,
      limit: number
  }
```

create a collection reviews with each document as : {“name": “abc”, “review”:”…“, “publish”:”true”}

now count the number of published comments per user name

```
>db.posts.mapReduce(
   function() { emit(this.name,1); },
   function(key, values) {return Array.sum(values)}, {
      query:{publish:"true"},
      out:”total_reviews"
   }
).find()
```

MongoDB: Indexing

- Can create index on any field of a collection or a sub-document fields
- e.g. document in a collection

```
{
    "address": {
        "city": “New Delhi",
        "state": "Delhi",
        "pincode": "110001"
    },
    "tags": [
        "football",
        "cricket",
        "badminton"
    ],
    "name": "Ravi"
}
```

- indexing a field in ascending order and find

```
> db.users.createIndex({“tags":1})
> db.users.find({tags:"cricket"}).pretty()
```

- indexing a sub-document field in ascending order and find

```
> db.users.createIndex({"address.city":1,"address.state":1,"address.pincode":1})
> db.users.find({“address.city":"New Delhi”}).pretty()
```

MongoDB: Joins

- Mongo 3.2+ it is possible to join data from 2 collections using aggregate
- Collection books (isbn, title, author) and books_selling_data(isbn, copies_sold)

```
db.books.aggregate([{ $lookup: {
          from: "books_selling_data",
          localField: "isbn",
          foreignField: "isbn",
          as: "copies_sold"
      }
 }])
```

- Sample joined document:

```
{
  "isbn": "978-3-16-148410-0",
  "title": "Some cool book",
  "author": "John Doe",
  "copies_sold": [
      {
          "isbn": "978-3-16-148410-0",
          "copies_sold": 12500
      }
   ]
}
```

If you have two collections (users , comments) and want to pull all the comments with pid=444 along with the user info for each comments

```
 { uid:12345, pid:444, comment="blah" }
 { uid:12345, pid:888, comment="asdf" }
 { uid:99999, pid:444, comment="qwer" }
```

users

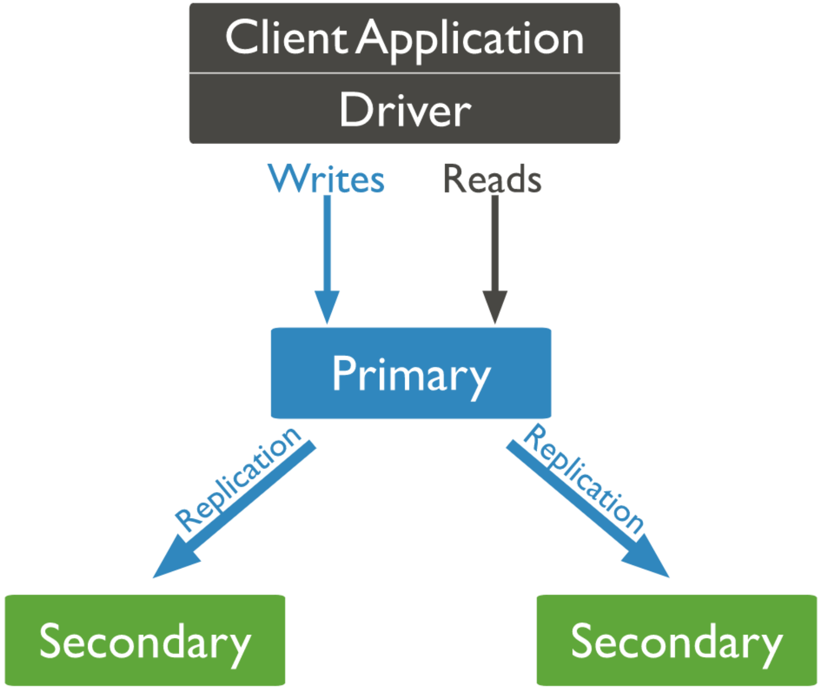
```
 { uid:12345, name:"john" }
 { uid:99999, name:"mia"  }
```

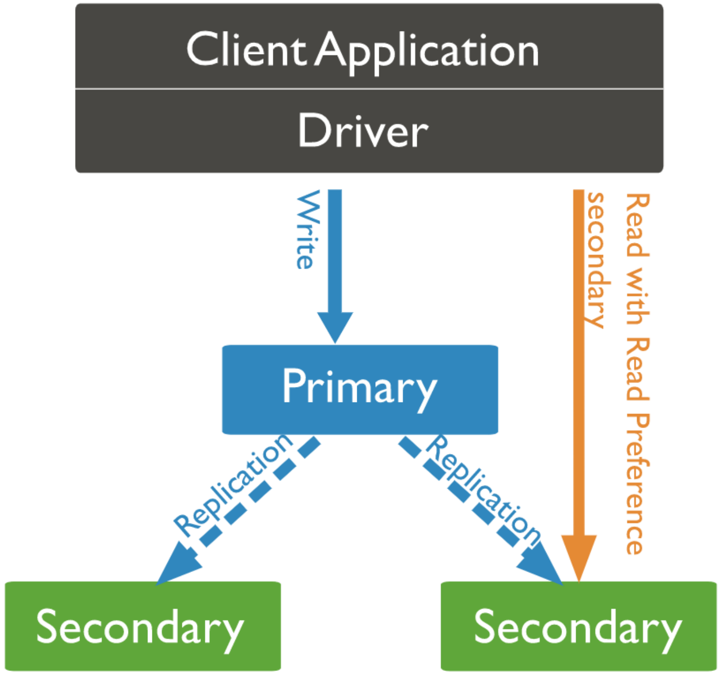
Join command - Join using $lookup

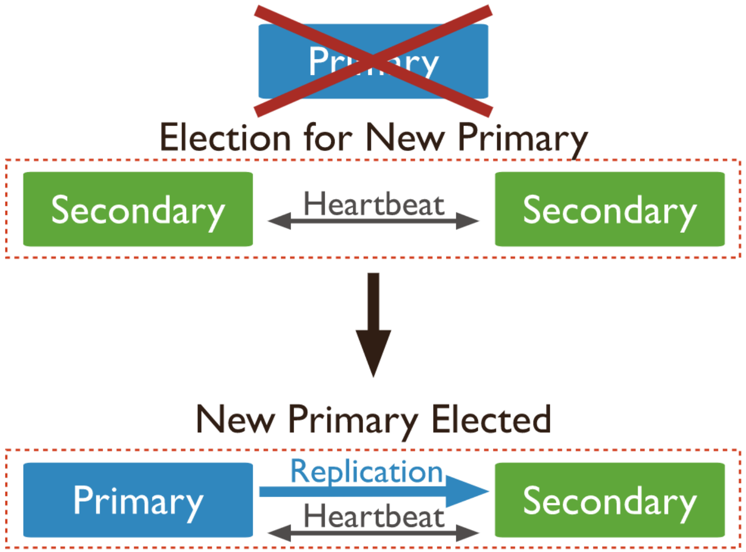
```
db.users.aggregate({
   $lookup:{
      from:"comments",
      localField:"uid",
      foreignField:"uid",
      as:"users_comments"
   }
})
```

MongoDB – Writes and Reads

- Document oriented DB
- Various read and write choices for flexible consistency tradeoff with scale / performance and durability
- Automatic primary re-election on primary failure and/or network partition



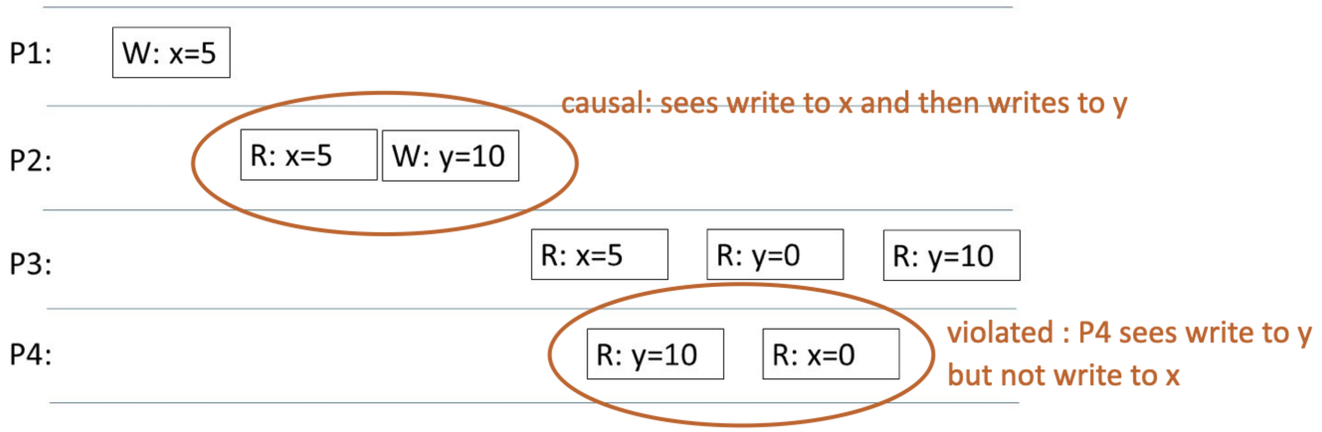




What is Causal Consistency (recap)

Read your writes	Read operations reflect the results of write operations that precede them.
Monotonic reads	Read operations do not return results that correspond to an earlier state of the data than a preceding read operation.
Monotonic writes	Write operations that must precede other writes are executed before those other writes.
Writes follow reads	Write operations that must occur after read operations are executed after those read operations.

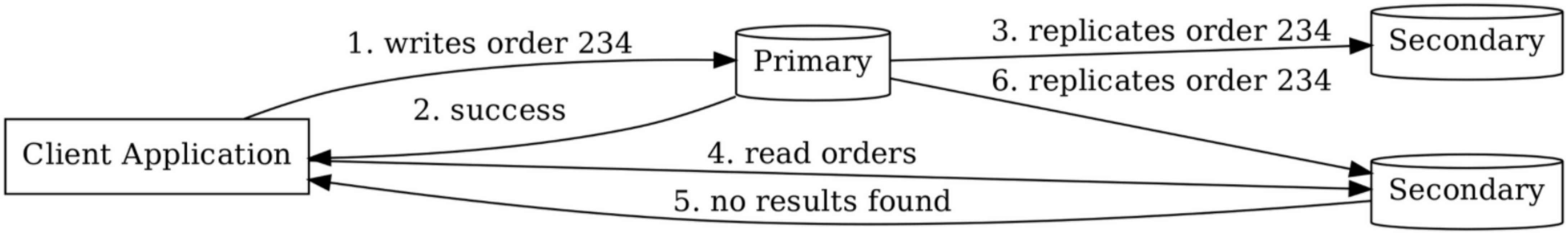
(Initial value of X and Y are 0)



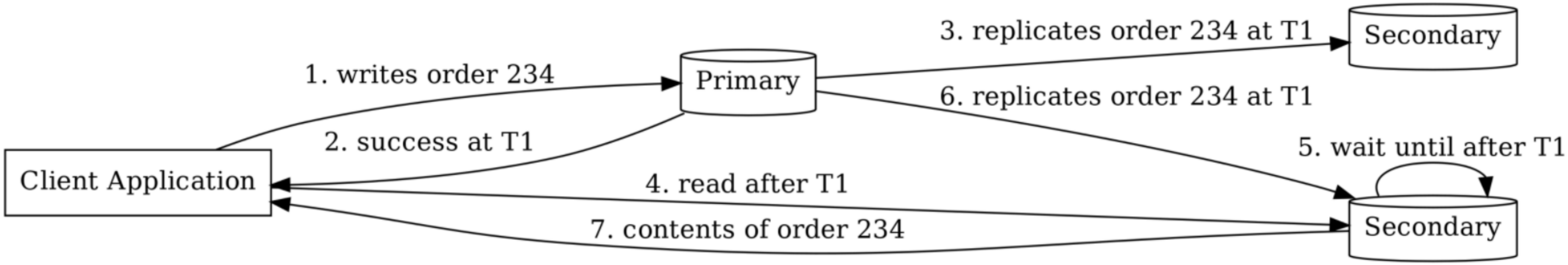
This schedule is **not** causally consistent, nor linearizable or strictly consistent or sequentially consistent

Example in MongoDB

- Case 1 : No causal consistency



- Case 2: Causal consistency by making read to secondary wait



https://engineering.mongodb.com/post/ryp0ohr2w9pvv0fks88kq6qkz9k9p3

MongoDB “read concerns”

- local :
  - ✓Client reads primary replica
  - ✓Client reads from secondary in causally consistent sessions
- available:
  - ✓Read on secondary but causal consistency not required
- majority :
  - ✓If client wants to read what majority of nodes have. Best option for fault tolerance and durability.
- linearizable :
  - ✓If client wants to read what has been written to majority of nodes before the read started.
  - ✓Has to be read on primary
  - ✓Only single document can be read

https://docs.mongodb.com/v3.4/core/read-preference-mechanics/

MongoDB “write concerns”

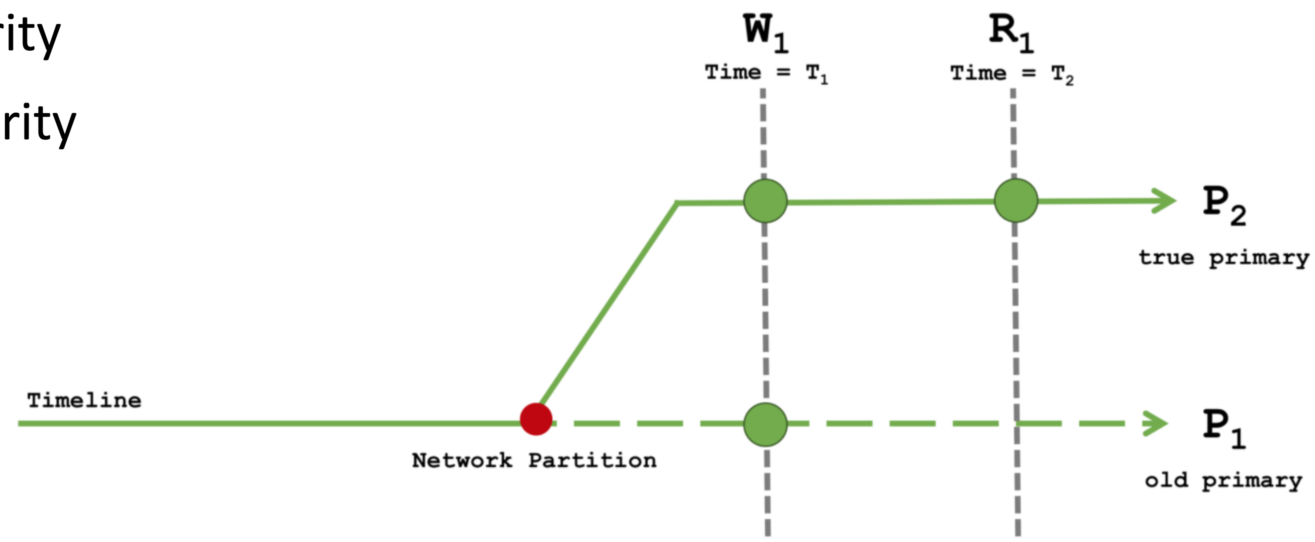
- how many replicas should ack
  - ✓1 - primary only
  - ✓0 - none
  - ✓n - how many including primary
  - ✓majority - a majority of nodes (preferred for durability)
- journaling - If True then nodes need to write to disk journal before ack else ack after writing to memory (less durable)
- timeout for write operation

https://docs.mongodb.com/manual/reference/write-concern/

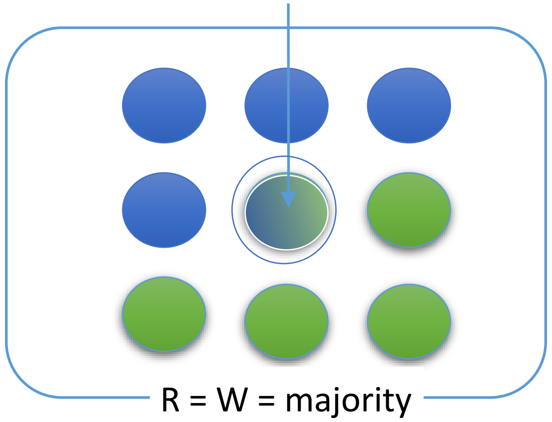
Consistency scenarios - causally consistent and durable

read=majority

write=majority



Read latest written value from common node



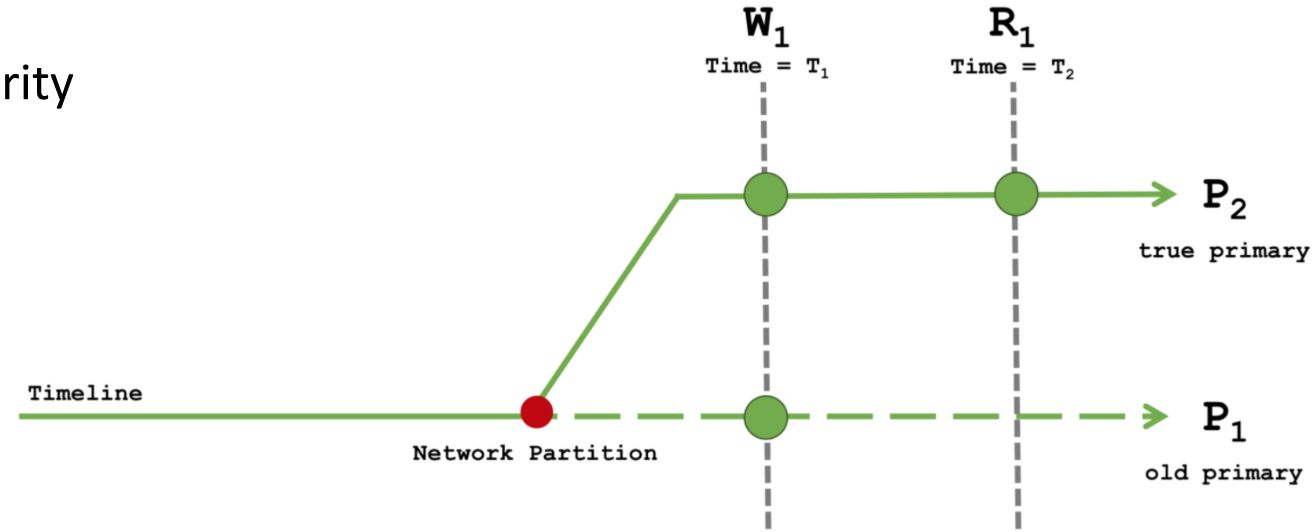
- W1 and R1 for P1 will fail and will succeed in P2
- So causally consistent, durable even with network partition sacrificing performance
- *Example*: Used in critical transaction oriented applications, e.g. stock trading

https://engineering.mongodb.com/post/ryp0ohr2w9pvv0fks88kq6qkz9k9p3

Consistency scenarios - causally consistent but not durable

read=majority

write=1



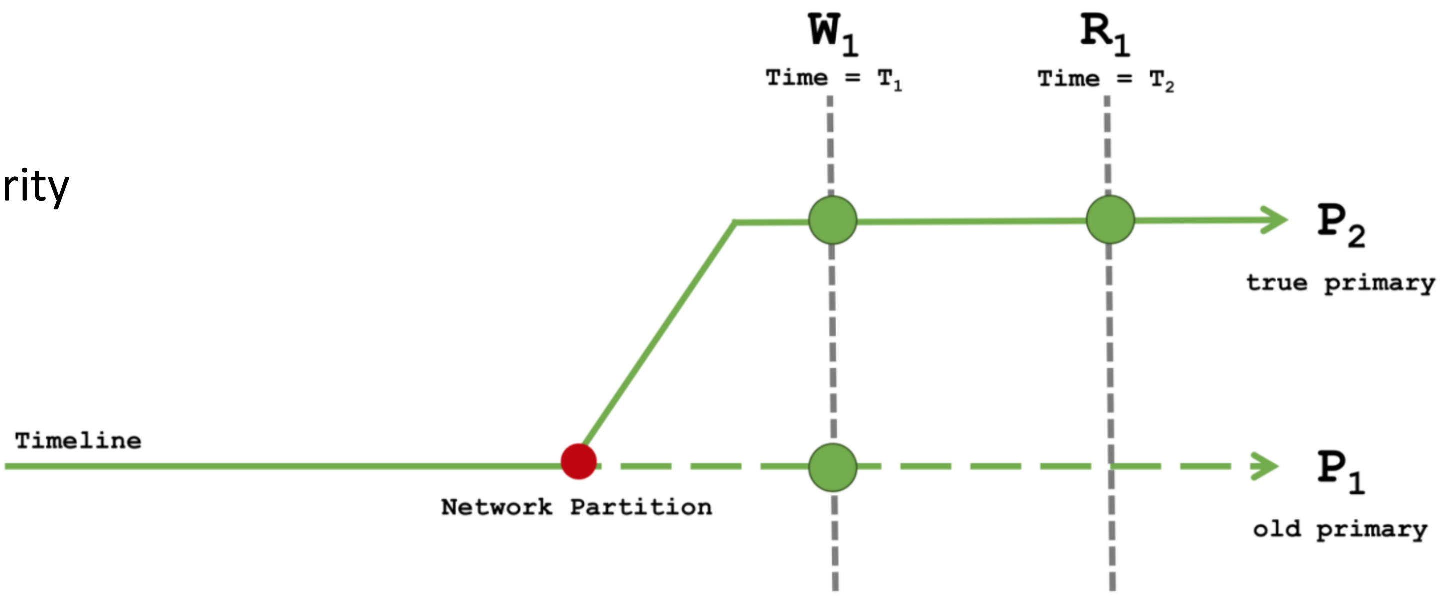
- W1 may succeed on P1 and P2. R1 will succeed only on P2. W1 on P1 may roll back.
- So causally consistent but not durable with network partition. Fast writes, slower reads.
- *Example*: Twitter - a post may disappear but if on refresh you see it then it should be durable, else repost.

https://engineering.mongodb.com/post/ryp0ohr2w9pvv0fks88kq6qkz9k9p3

Consistency scenarios - eventual consistency with durable writes

read=local

write=majority



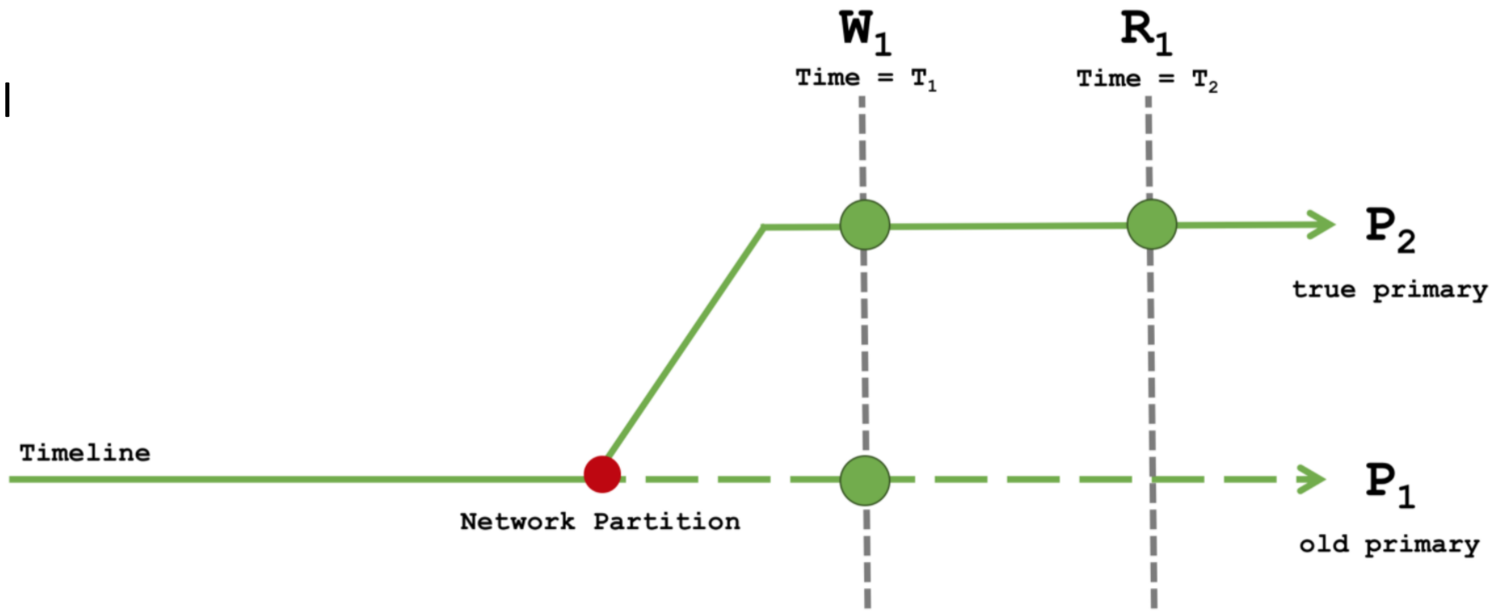
- W1 will succeed only for P2 and will not be accepted on P1 after failure. Reads may not succeed to see the last write on P1. Slow durable writes and fast non-causal reads.
- *Example:* Review site where write should be durable if committed but reads don't need causal guarantee as long as it appears some time (eventual consistency).

https://engineering.mongodb.com/post/ryp0ohr2w9pvv0fks88kq6qkz9k9p3

Consistency scenarios - eventual consistency but no durability

read=local

write=1



- Same as previous scenario and not writes are also not durable and may be rolled back.
- *Example:* Real-time sensor data feed that needs fast writes to keep up with the rate and reads should get as much recent real-time data as possible. Data may be dropped on failures.

https://engineering.mongodb.com/post/ryp0ohr2w9pvv0fks88kq6qkz9k9p3

How applications deal with eventual consistency

- Application must be ready to deal with multiple versions of data
- Application must handle stale data
- Application must be able to fix inconsistent state
- Read again, if you do not get the data during the first read

MongoDB on Cloud

- MongoDB Atlas on AWS Cloud
- Automated MongoDB Service on Microsoft Azure
- MongoDB Atlas on Google Cloud

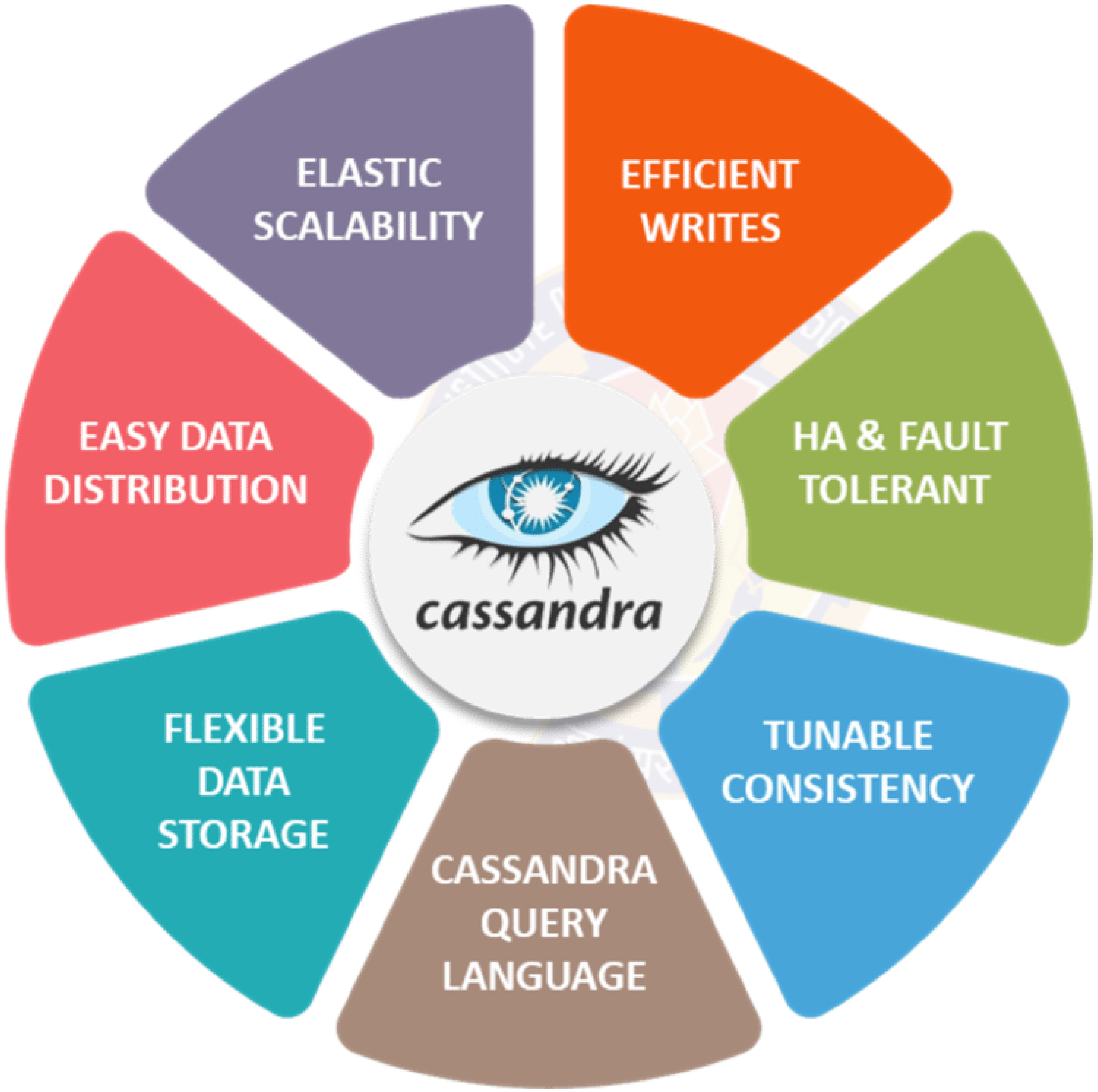
Topics for today

- NoSQL Introduction
- Classification
- Examples
    - MongoDB
    - **Cassandra**
    - GraphDBs: Neo4J and Tinkerpop

Cassandra

- Born in Facebook and built on Amazon Dynamo and Google Big Table concepts
- AP design in CAP context
- High performance, high availability applications that can sacrifice consistency
  - ✓ Hence built for peer-to-peer symmetric nodes instead of primary-secondary architecture (as in MongoDB)
- Column oriented DB
  - ✓Create keyspace (like a DB)
  - ✓ Within keyspace create column family (like a table)
  - ✓ Within CF create attributes / columns with their types

Cassandra features



Cassandra on Cloud

- Amazon Keyspaces (for Apache Cassandra)
- Azure Managed Instance for Apache Cassandra
- DataStax Astra DB for Apache Cassandra (Google Cloud)

Read / Write

- Writes
  - ✓ Written to commit log sequentially and deemed successful
  - ✓ Data is indexed and put into in-memory Memtable (one or more per Column Family)
  - ✓ Memtable is flushed to disk SSTable file
  - ✓ SSTable is immutable and append only
  - ✓ Partitioning and replication happens automatically
- Reads
  - ✓ Client connects to any node to read data
  - ✓ Consistency level decides when a read is returned, i.e. how many replicas should contain the same copy
  - ✓ Read repair: replication via a Gossip protocol is triggered as a client issues a read and Cassandra has to meet the required consistency level

Consistency semantics (1)

- No primary replica - high partition tolerance and availability and levels of consistency
- Support for light transactions with “linearizable consistency”
- A Read or Write operation can pick a consistency level
  - ONE, TWO, THREE, ALL - 1,2,3 or all replicas respectively have to ack
  - ANY - Write to any node even if replicas are down (ref Hinted Handoff)
  - QUORUM - majority have to ack
  - LOCAL_QUORUM - majority within same datacenter have to ack
  - …

https://cassandra.apache.org/doc/latest/architecture/dynamo.html
https://cassandra.apache.org/doc/latest/architecture/guarantees.html#

Tunable Consistency

Cassandra guarantees strong consistency if

(nodes_**W**ritten + nodes_**R**ead) > replication_factor **N**

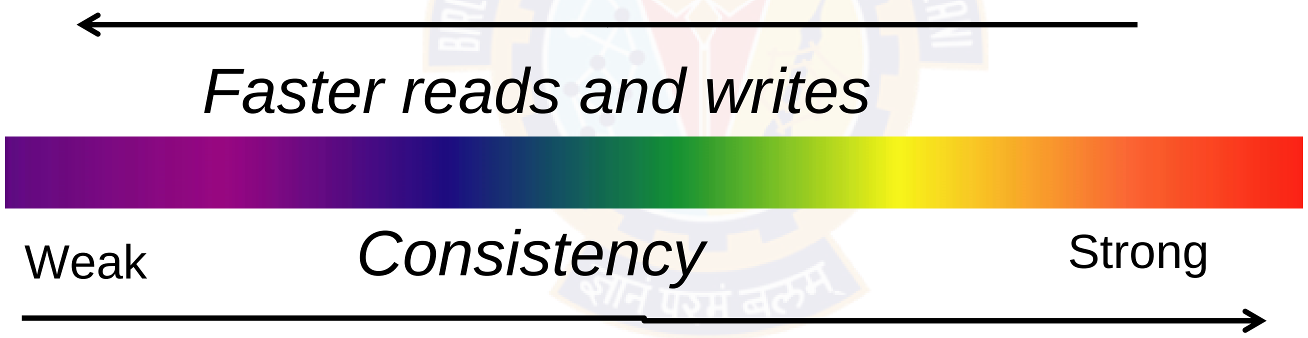
$$\boxed{\mathbf{R + W > N}}$$

Tuning done by controlling the number of nodes

- Selected for Write
- Selected for reads

Consistency Spectrum

- Cassandra has C A P
- But Consistency is tunable
- Give up a little A and P to get more C

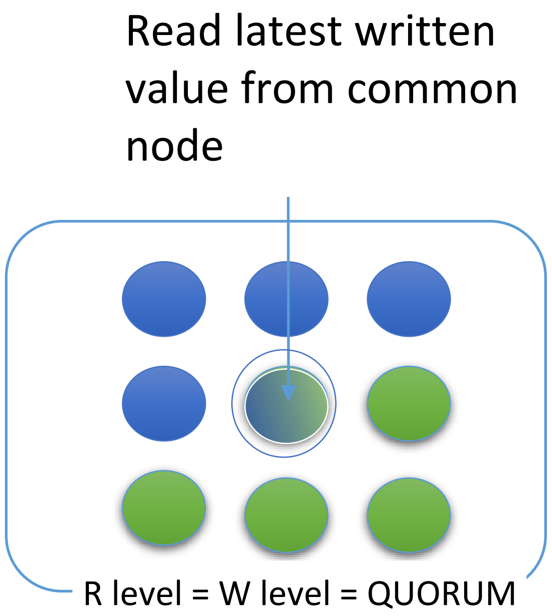


The higher the consistency, the less chance you get stale data during read

- Pay for this with latency
- Depends on your situational needs

Consistency semantics (2)

- For "causal consistency" pick Read consistency level = Write consistency level = QUORUM
- Why ? At least one node will be common between write and read set so a read will get the last write of a data item
- What happens if read and write use LOCAL_QUORUM ?
- If no overlap read and write sets then "Eventual consistency"



https://cassandra.apache.org/doc/latest/architecture/dynamo.html
https://cassandra.apache.org/doc/latest/architecture/guarantees.html#

Lightweight transactions

- INSERT and UPDATE with an IF clause support lightweight tx semantics at data item level
  - ✓ Aka Compare and set
  - ✓ Increases overheads by 4x due to coordination

```
cqlsh> INSERT INTO cycling.cyclist_name (id, lastname, firstname)
  VALUES (4647f6d3-7bd2-4085-8d6c-1229351b5498, 'KNETEMANN', 'Roxxane')
  IF NOT EXISTS;

cqlsh> UPDATE cycling.cyclist_name
  SET firstname = ‘Roxane’
  WHERE id = 4647f6d3-7bd2-4085-8d6c-1229351b5498
  IF firstname = ‘Roxxane’;
```

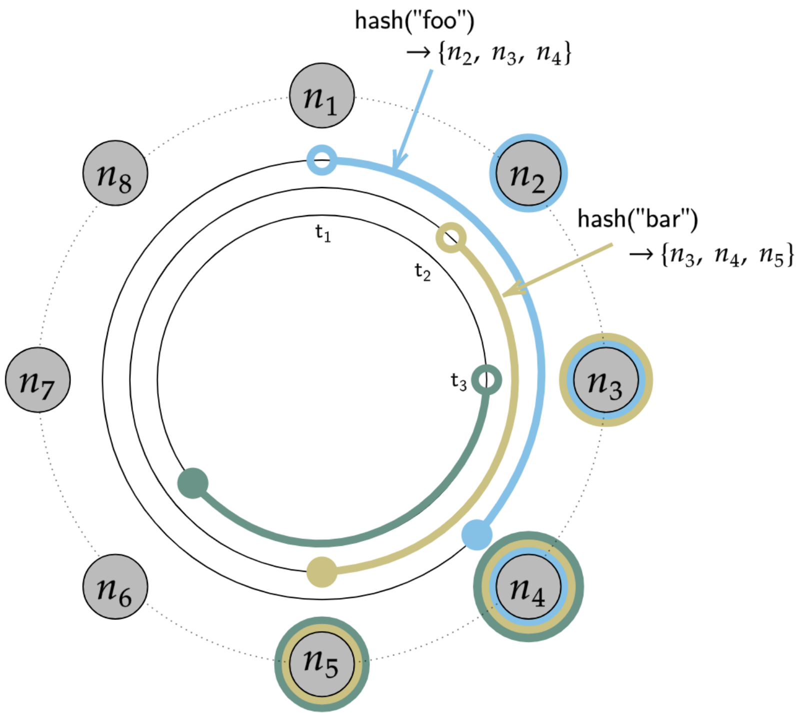
https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useInsertLWT.html

Replication strategy for user data

- Simple
  - ✓ Specify replication factor = N and data is stored in N nodes of cluster
- NetworkTopology
  - ✓ Specify replication factor per DC where we want reliability from DC failures
  - ✓ e.g. CREATE KEYSPACE cluster1 WITH replication = {'class': 'NetworkTopologyStrategy', 'eastDC' : 2, ‘westDC' : 3};

Partitioners

- Partitions data based on hashing to distribute data blocks from a column among nodes*
- Random
  - ✓ Crypto hash (MD5)- more expensive
- Murmur
  - ✓ Non-crypto consistent hash (MU-Multiple / R - Rotate operations but easier to reverse compared to Crypto hash)
  - ✓ 3-5x faster and overall 10% performance improvement
- Byteorder
  - ✓ Lexical order



Sample queries

```
> create keyspace demo with replication={'class':'SimpleStrategy', 'replication_factor':1};
> describe keyspaces;
> use demo;
> create table student_info (rollno int primary key, name text, doj timestamp, lastexampercent double);
> describe table student_info ;
> consistency quorum
> insert into student_info (rollno,name,doj,lastexampercent) values (4,'Roxanne', dateof(now()), 90) using ttl 30;
> select rollno from student_info where name='Roxanne' ALLOW FILTERING;
> update student_info set lastexampercent=98 where rollno=2 IF name='Sam';
```

or columnfamily

Case study - eBay

- Marketplace has 100 million active buyers with 200+ million items
- 2B page views, 80B DB calls, multi-PB storage capacity
- No transactions, joins, referential integrity
- Multi-DC deployment
- 400M+ writes and 200M+ reads
- 3 Use cases
  - ✓ Social signal on product pages (read latency is not important but write performance is key)
  - ✓ Connecting users and items via buy, sell, bid, watch events
  - ✓ Many time series analysis cases, e.g. fraud detection

https://www.slideshare.net/jaykumarpatel/cassandra-at-ebay-13920376/2-eBay_Marketplaces_97_million_active

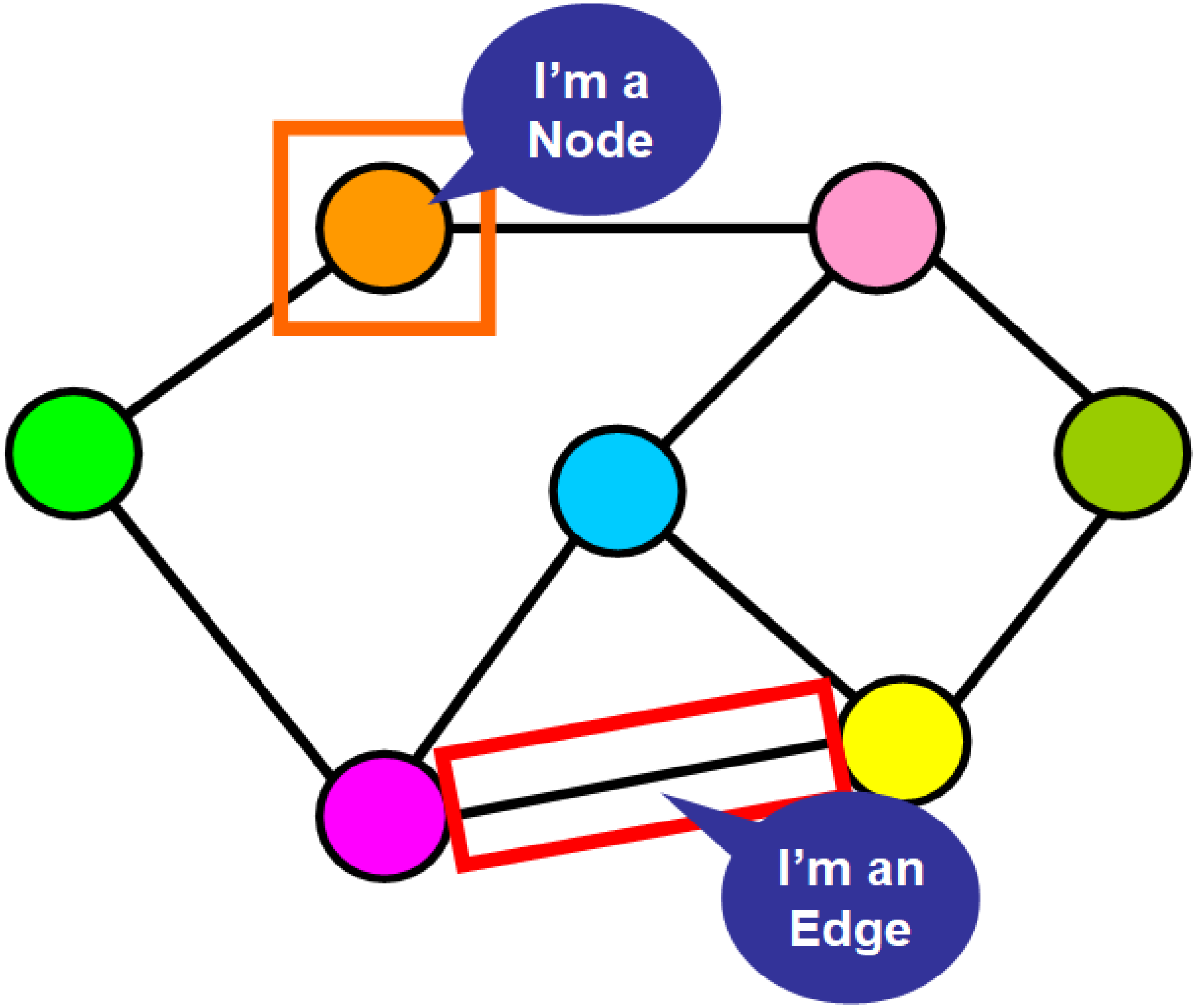
Case study - AdStage (from AWS use cases)

- Sector AdTech
- Online advertising platform to manage multi-channel ad campaigns on Google, FB, Twitter, Bing, LinkedIn
- 3 clusters with 80+ nodes on AWS
- Vast amount of real-time data from 5 channels
- Constantly monitor trends and optimise campaigns for advertisers
- High performance and availability - consistency is not critical as it is read mainly
- Cassandra cluster can scale as more clients are added with no SPOF

Topics for today

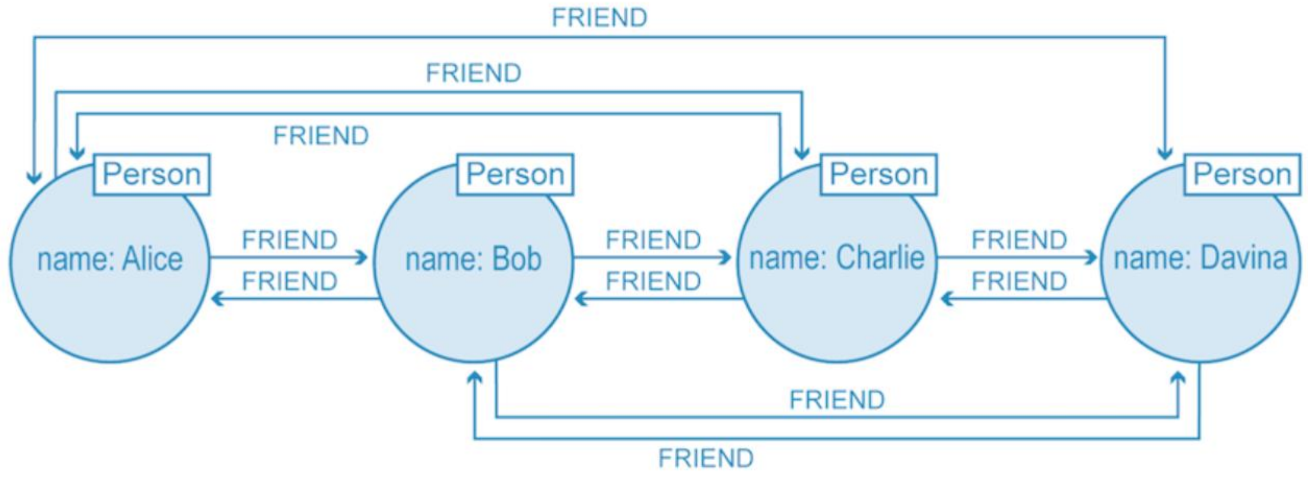
- NoSQL Introduction
- Pros-Cons
- Classification
- Examples
  - MongoDB
  - Cassandra
  - **GraphDBs: Neo4J and Tinkerpop**

Graphs

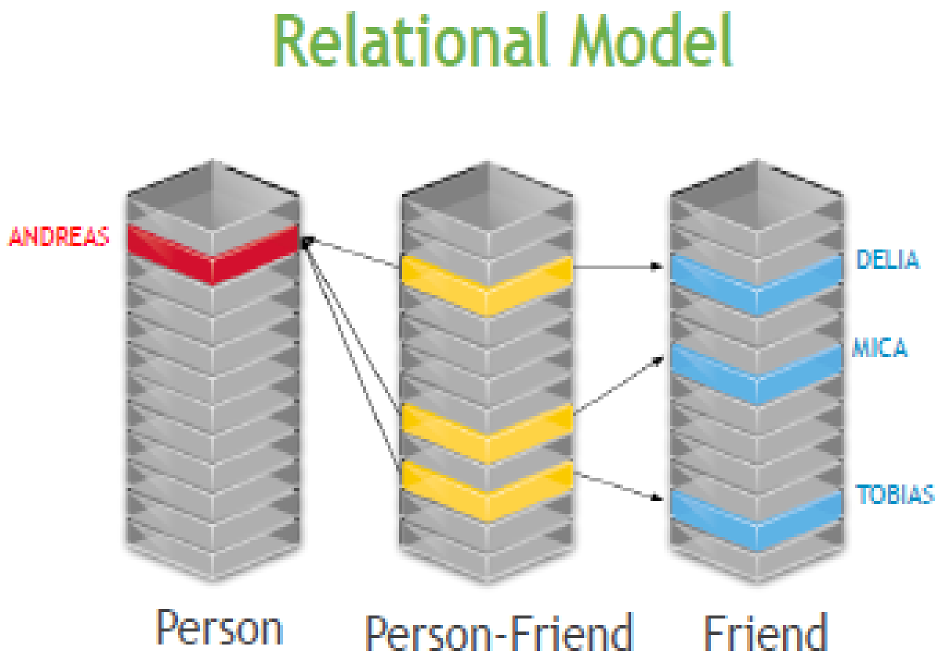


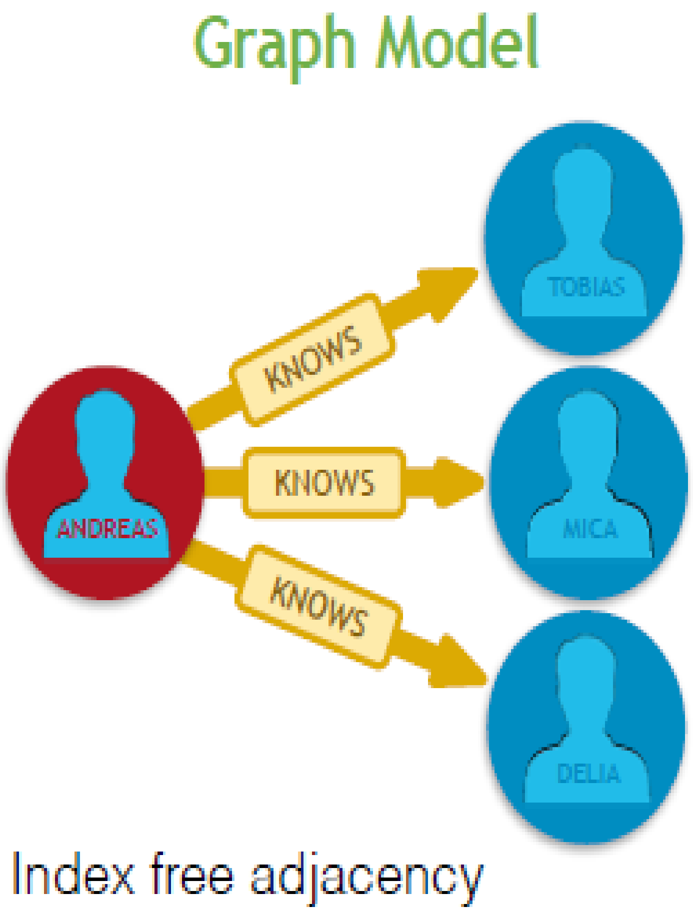
Graph computing

- Property graphs
  - Data is represented as vertices and edges with properties
  - Properties are key value pairs
  - Edges are relationships between vertices
- When to use a graph DB ?
  - A relationship-heavy data set with large set of data items
  - Queries are like graph traversals but need to keep query performance almost constant as database grows
  - A variety of queries may be asked from the data and static indices on data will not work



Relational Vs Graph Models

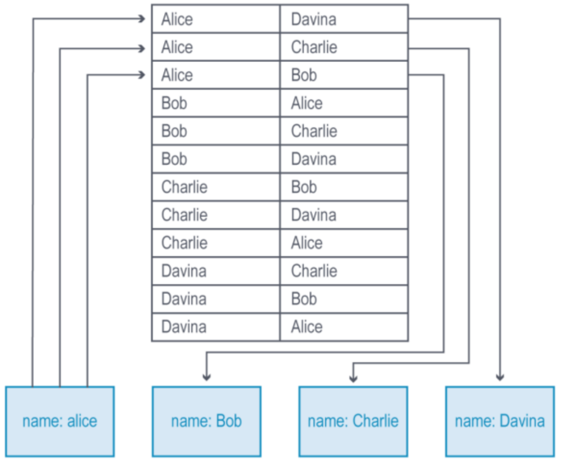




Native vs Non-Native Graph storage

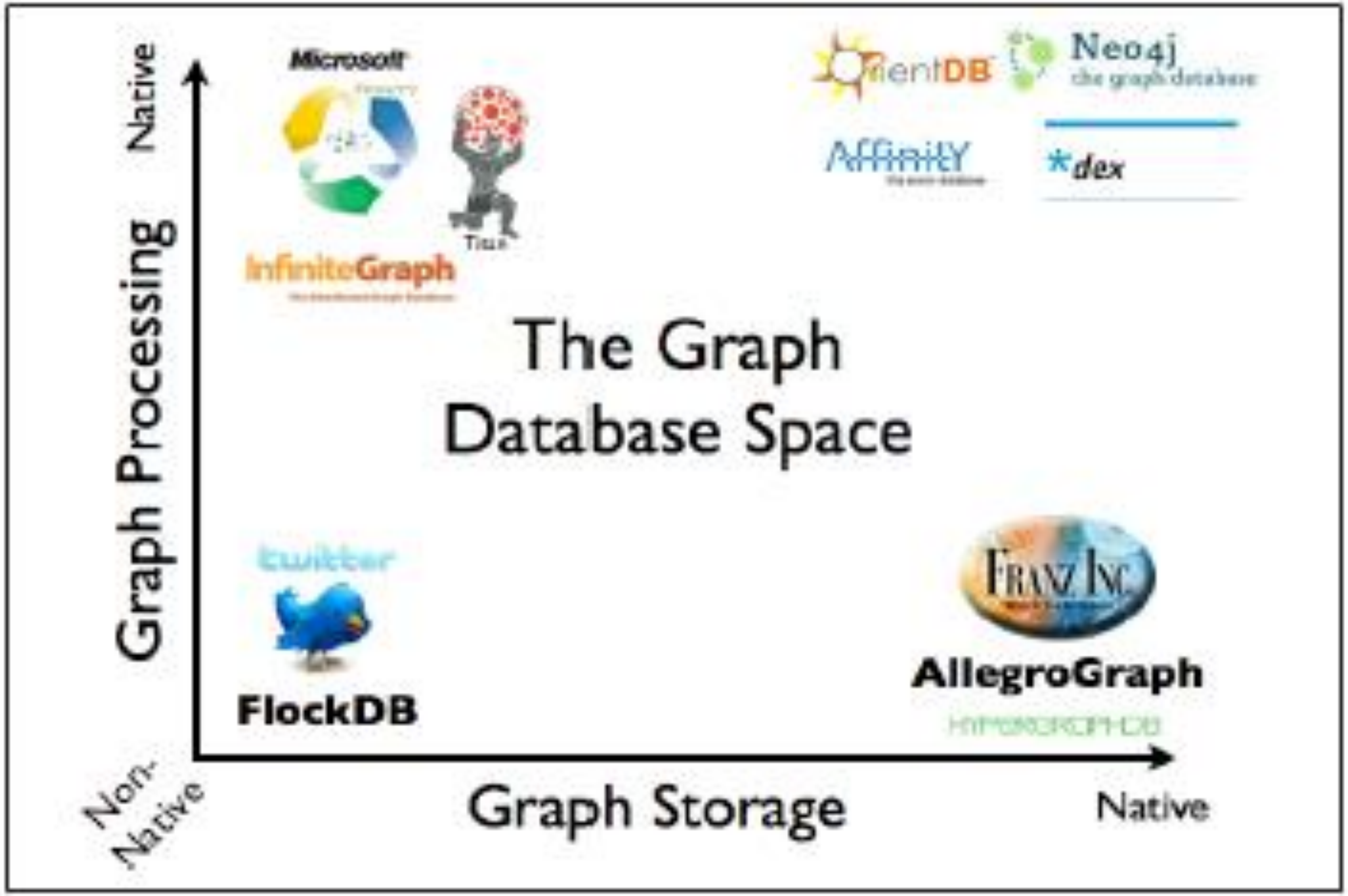
- Non-native graph computing platforms can use external DBs for data storage
  - e.g. TinkerPop is an in-memory DB + computing framework that can store in ElasticSearch, Cassandra etc.
- Native platform support built-in storage
  - e.g. Neo4j
- Native approach is much faster because adjacent nodes and edges are stored closer for faster traversal
  - In a non-native approach, extensive indexing has to be used
- Native approach scales as nodes get added

One-hop index



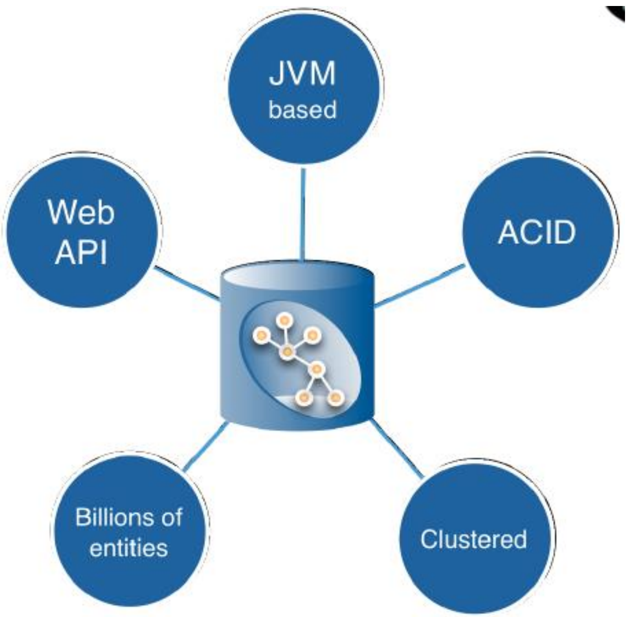
https://neo4j.com/blog/native-vs-non-native-graph-technology/

Graphs for Storage and Processing



What is Neo4J

- It’s is a Graph Database supporting full ACID Transactions
- Embeddable in applications and server deployable
- Java based, Open sourced
- Schema free, bottom-up data model design
- Neo4j is stable
- In 24/7 operation since 2003
- Neo4j is under active development
- High performance graph operations
- Supports the **Cypher** query language
- Traverses 1,000,000+ relationships/sec on commodity hardware
- No. of nodes and relationships decide Volume of data

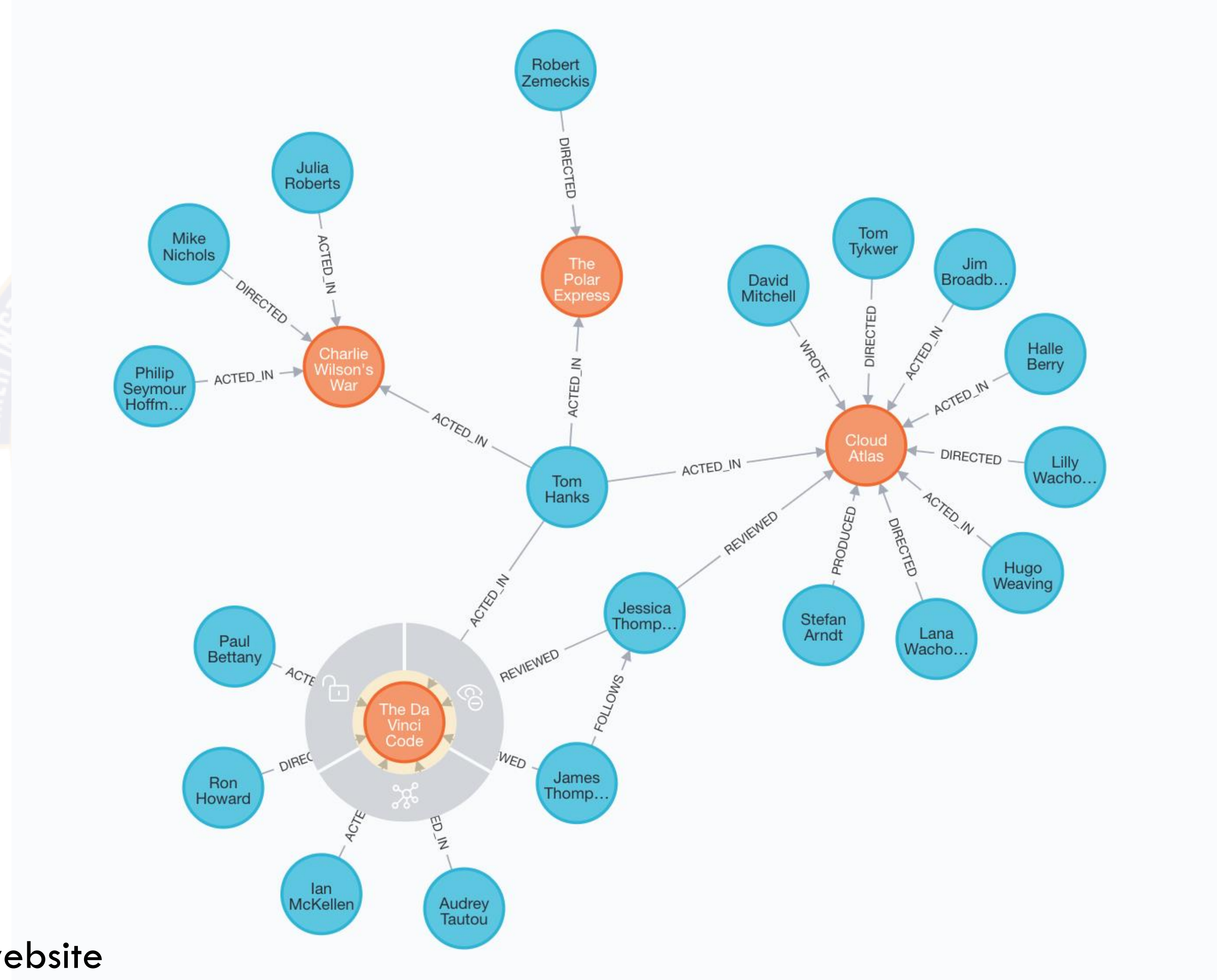


One-hop index

https://neo4j.com/blog/native-vs-non-native-graph-technology/

Neo4j / Cypher

- Cypher is a Declarative language for graph query
- Example: match (:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(m:Movie) where m.released > 2000 RETURN m limit 5



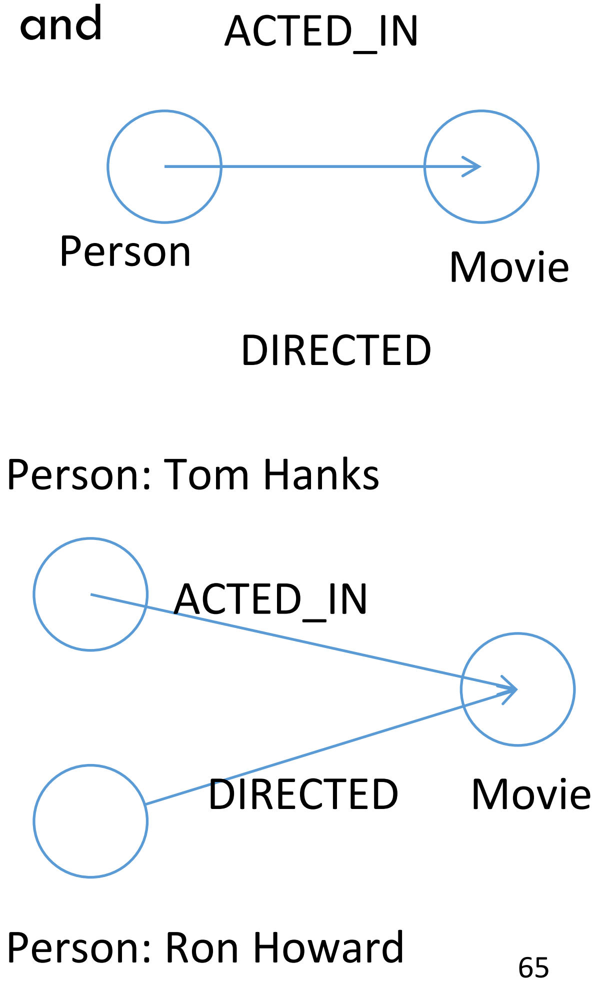
Launch a free sandbox with dataset on neo4j website

Neo4j / Cypher: More queries

- Find movies that Tom Hanks acted in and directed by Ron Howard released after 2000
  - Match (:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(m:Movie),(:Person {name: 'Ron Howard'})-[:DIRECTED]->(m) where m.released > 2000 RETURN m limit 5
- Who were the other actors in the movie where Tom Hanks acted in and directed by Ron Howard released after 2000
  - Match (:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(m:Movie),(:Person {name: 'Ron Howard’})-[:DIRECTED]->(m), (p:Person)-[:ACTED_IN]->(m) where m.released > 2000 RETURN p limit 5

Apache Tinkerpop / Gremlin

- TinkerPop is a computing platform that connects to GraphDBs that actually store the nodes and edges. Built-in TinkerGraph stores in-memory data only.
- Gremlin is the query language (with traversal machine) that supports Declarative and Imperative flavours
- Sample queries
  - movies where Tom Hanks has acted
    - g.V().hasLabel('person').has('name','Tom Hanks').outE('ACTED_IN').hasLabel('movies').values('name')
  - movies where Tom Hanks has acted and directed by Ron Howard
    - g.V().hasLabel('person').has('name','Tom Hanks').outE('ACTED_IN').inE('DIRECTED').has('name','Ron Howard').outE('DIRECTED').values('name')



Neo4j on Cloud

- Neo4j Enterprise Edition on AWS
- Neo4j - The Easiest Way to Graph on MS Azure
- NEO4J AURA on Google Cloud

Summary

- NoSQL databases are useful when
  - ✓ you have to deal with large data sets
  - ✓ may need geographical distribution
  - ✓ No need for ACID transactions and need flexible consistency
- Choices between key-value, column based, document based, graph based data stores
- Graph DBs and computing models are very suitable when data sets are relationship heavy - can be modelled as large number of nodes and edges and queries are similar to graph traversal
  - ✓ Complex relation centric queries are possible
  - ✓ Graph traversal costs can be kept stable with data growth

Next Session:

Cloud computing and storage