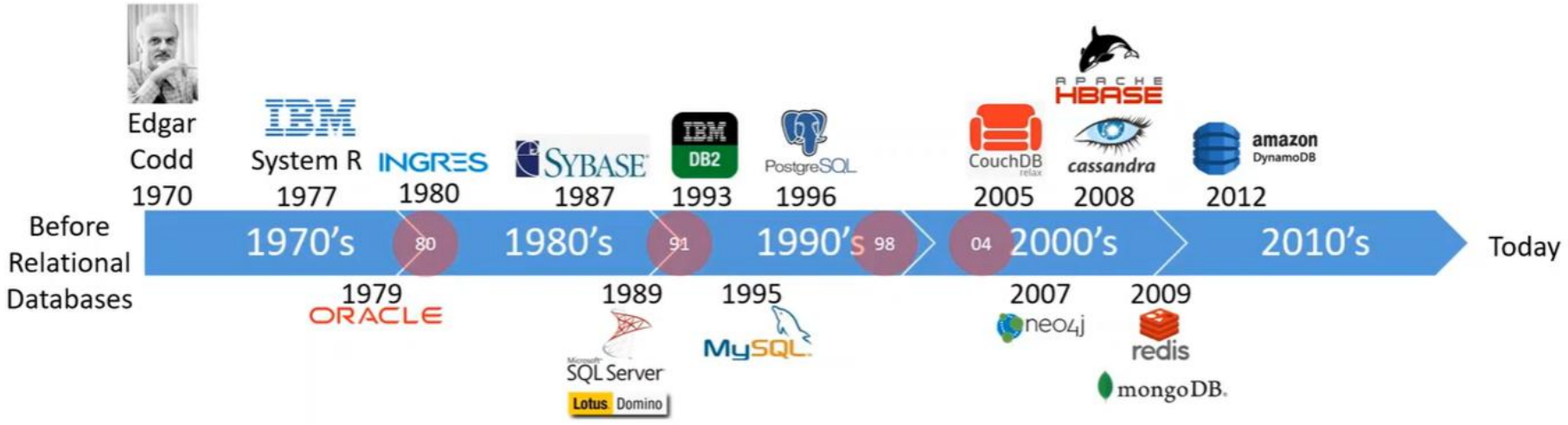
Lesson 3

DATABASES

Today's requirements on databases

- Big data (large datasets)
- Agility
- Unstructured/ semi structured data

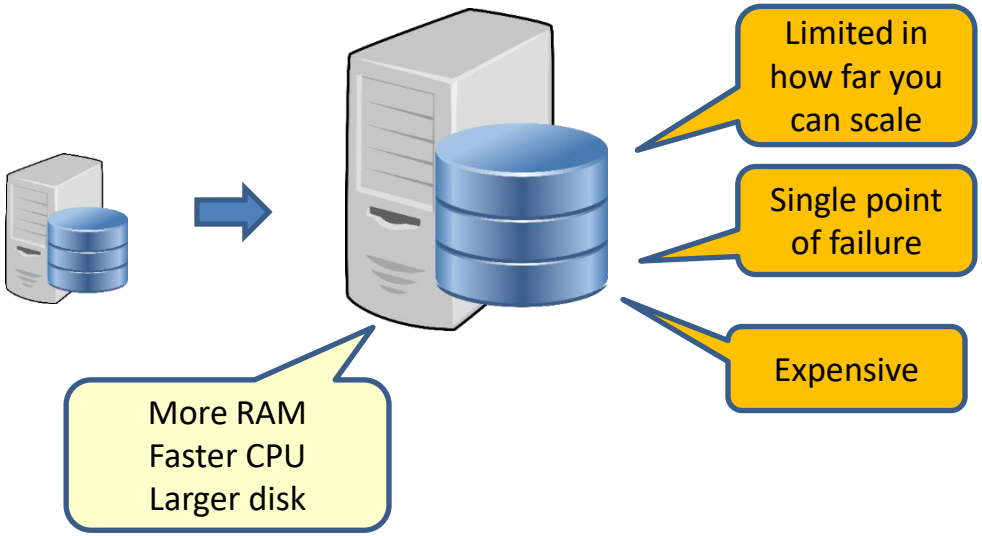


Large datasets

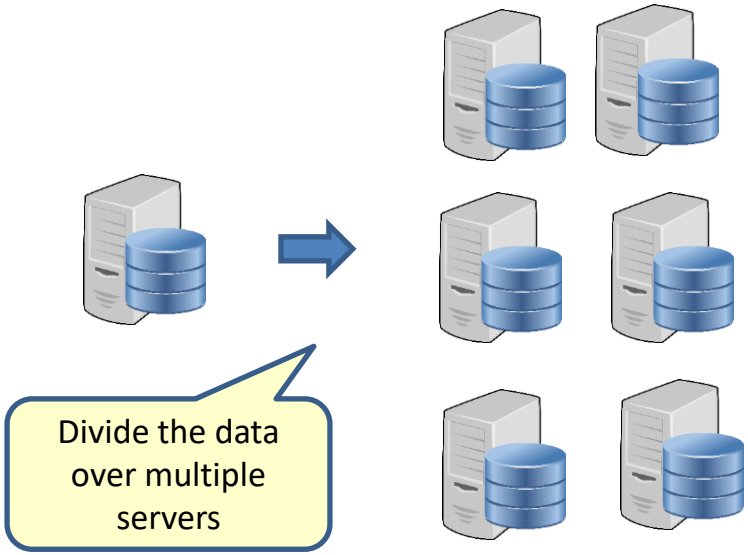
- Too much data
  - The data does not fit anymore on one node

Database Scaling

- Vertical scaling



- Horizontal scaling

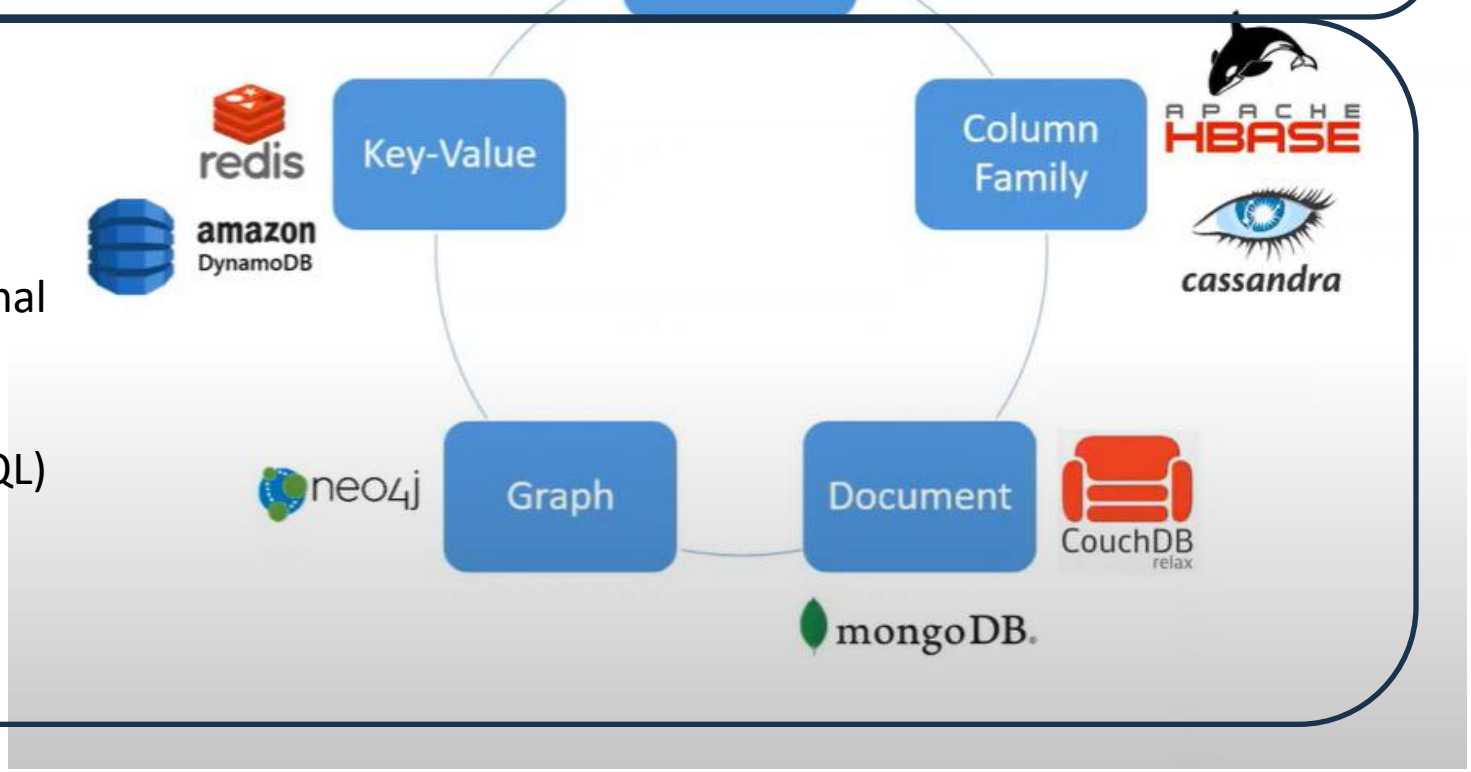


Five different types of databases

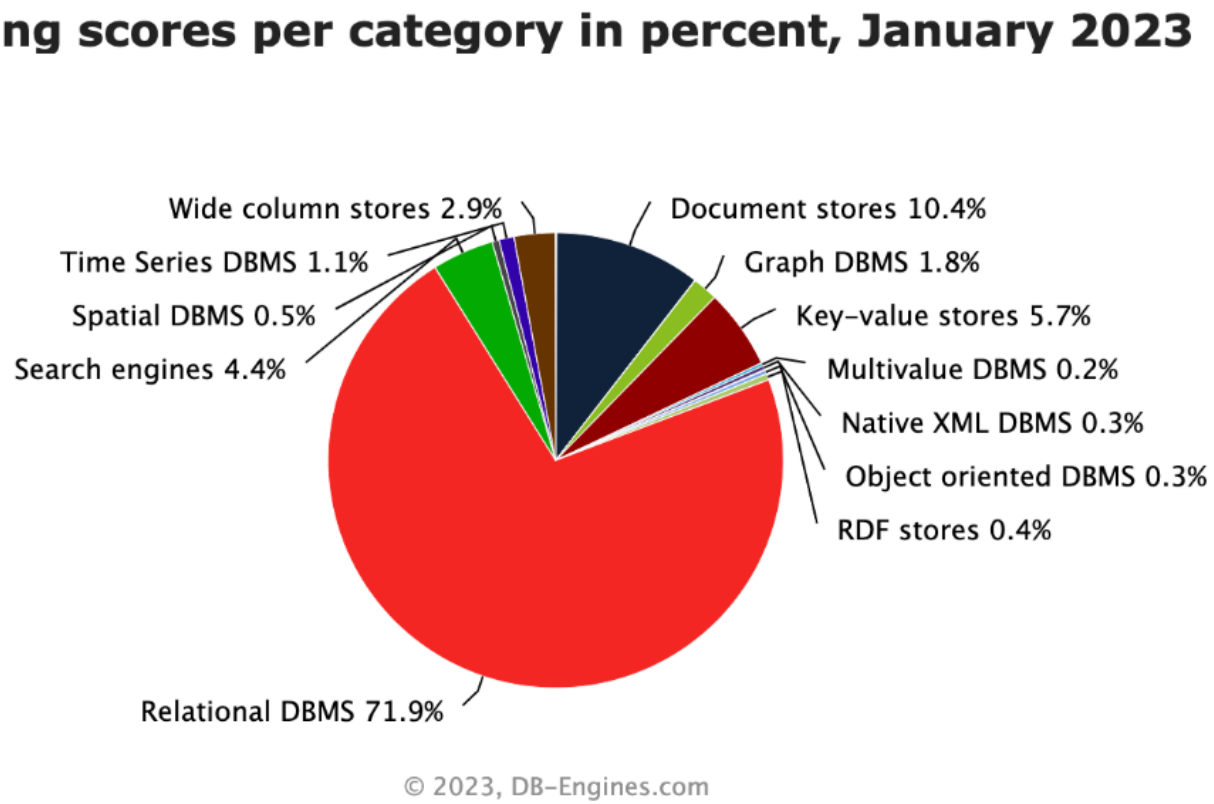
Relational

Non-relational

NoSQL
(Not only SQL)



Relational vs non-relational popularity



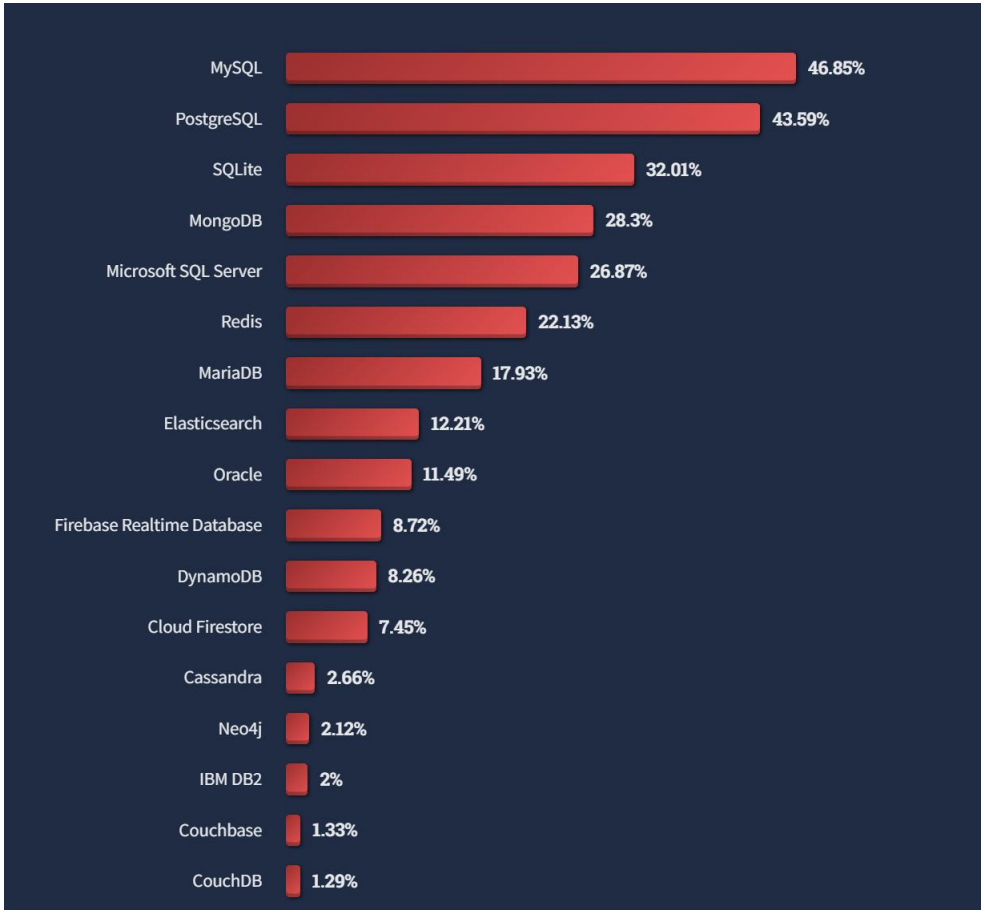
Most popular databases

https://db-engines.com/en/ranking

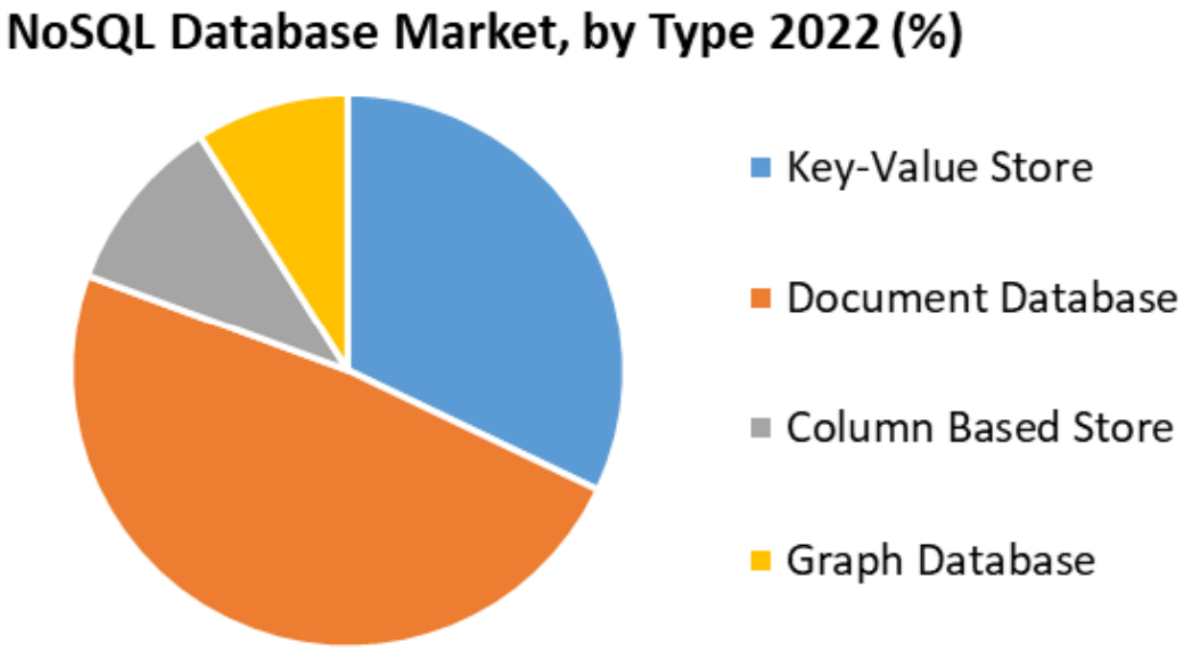
Rank May 2024	Rank Apr 2024	Rank May 2023	DBMS	Database Model	Score May 2024	Score Apr 2024	Score May 2023
1.	1.	1.	Oracle	Relational, Multi-model	1236.29	+2.02	+3.66
2.	2.	2.	MySQL	Relational, Multi-model	1083.74	-3.99	-88.72
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	824.29	-5.50	-95.80
4.	4.	4.	PostgreSQL	Relational, Multi-model	645.54	+0.49	+27.64
5.	5.	5.	MongoDB	Document, Multi-model	421.65	-2.31	-14.96
6.	6.	6.	Redis	Key-value, Multi-model	157.80	+1.36	-10.33
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	135.35	+0.57	-6.28
8.	8.	↓ 7.	IBM Db2	Relational, Multi-model	128.46	+0.97	-14.56
9.	9.	↑ 11.	Snowflake	Relational	121.33	-1.87	+9.61
10.	10.	↓ 9.	SQLite	Relational	114.32	-1.69	-19.54
11.	11.	↓ 10.	Microsoft Access	Relational	104.92	-0.49	-26.26
12.	12.	12.	Cassandra	Wide column, Multi-model	101.89	-1.97	-9.25
13.	13.	13.	MariaDB	Relational, Multi-model	93.21	-0.60	-3.66
14.	14.	14.	Splunk	Search engine	86.45	-2.26	-0.19
15.	↑ 17.	↑ 18.	Databricks	Multi-model	78.61	+2.28	+14.66
16.	↓ 15.	16.	Microsoft Azure SQL Database	Relational, Multi-model	77.99	-0.41	-1.21
17.	↓ 16.	↓ 15.	Amazon DynamoDB	Multi-model	74.07	-3.50	-7.04
18.	18.	↓ 17.	Hive	Relational	61.17	-1.41	-12.44
19.	19.	↑ 20.	Google BigQuery	Relational	60.38	-1.52	+5.51
20.	20.	↑ 21.	FileMaker	Relational	48.20	-1.53	-3.80
21.	21.	↓ 19.	Teradata	Relational, Multi-model	45.33	-2.52	-17.39
22.	22.	↑ 23.	SAP HANA	Relational, Multi-model	44.69	-1.14	-5.67
23.	23.	↓ 22.	Neo4j	Graph	44.46	-0.01	-6.65

Most popular databases

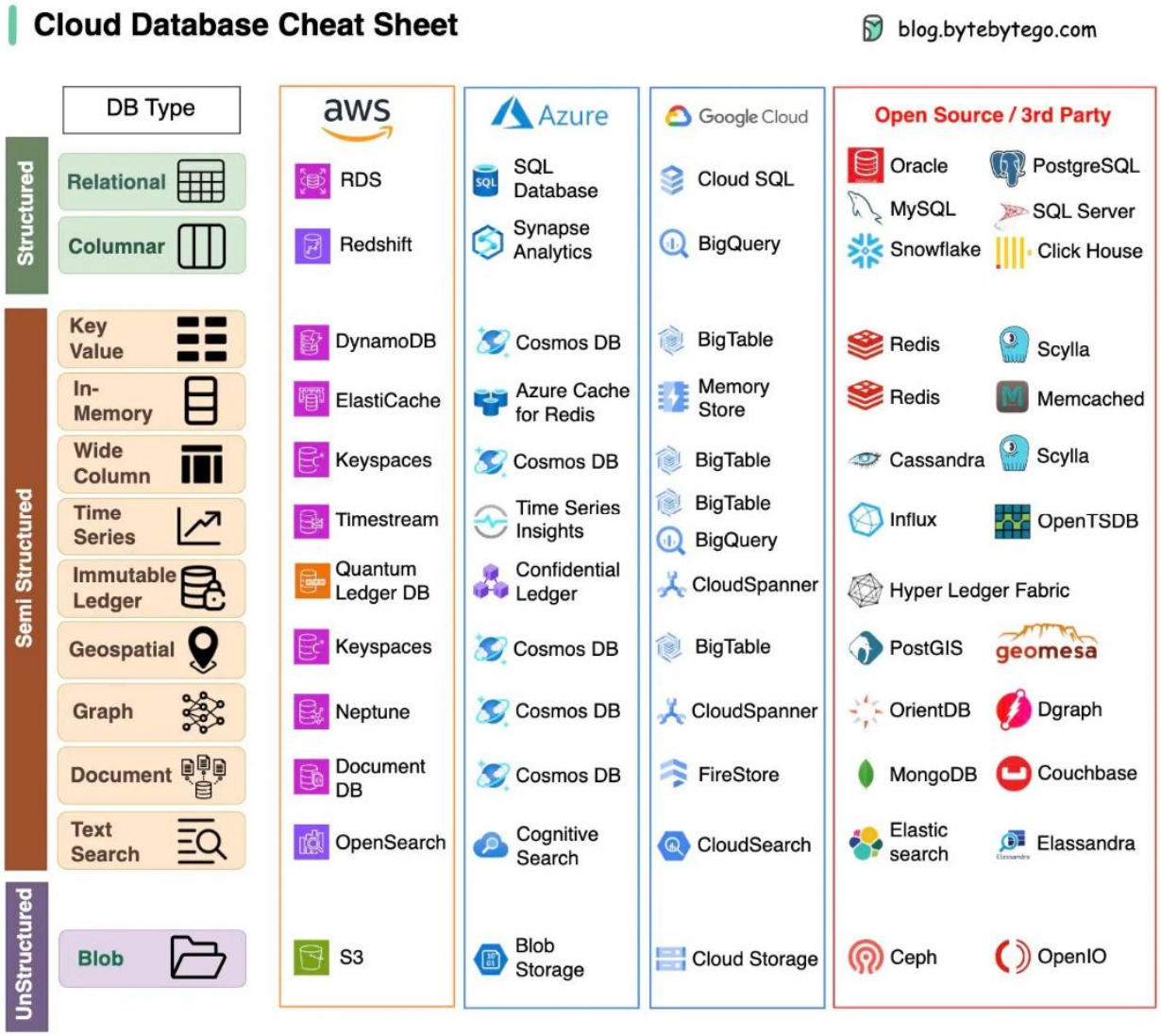
https://survey.stackoverflow.co/2022#technology

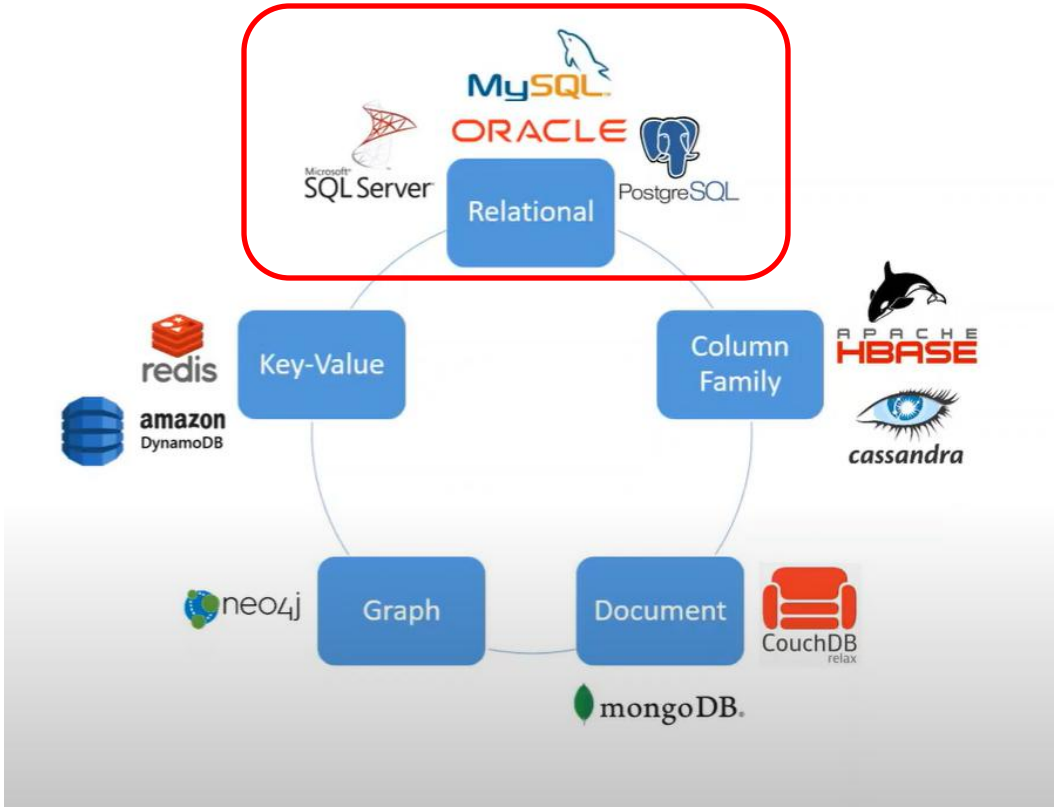


Most popular NoSQL databases



Cloud database





RELATIONAL DATABASE

Relational databases

Data is normalized: No duplication of data

Key

Relational - Tables

Relationships are expressed with key's, foreign keys and link tables

Customer ID	First Name	Last Name	City
0	John	Doe	New York
1	Mark	Smith	San Francisco
2	Jay	Black	Newark
3	Meagan	White	London
4	Edward	Daniels	Boston

Account Number	Branch ID	Account Type	Customer ID
10	100	Checking	0
11	101	Savings	0
12	101	IRA	0
13	200	Checking	1
14	200	Savings	1
15	201	IRA	2

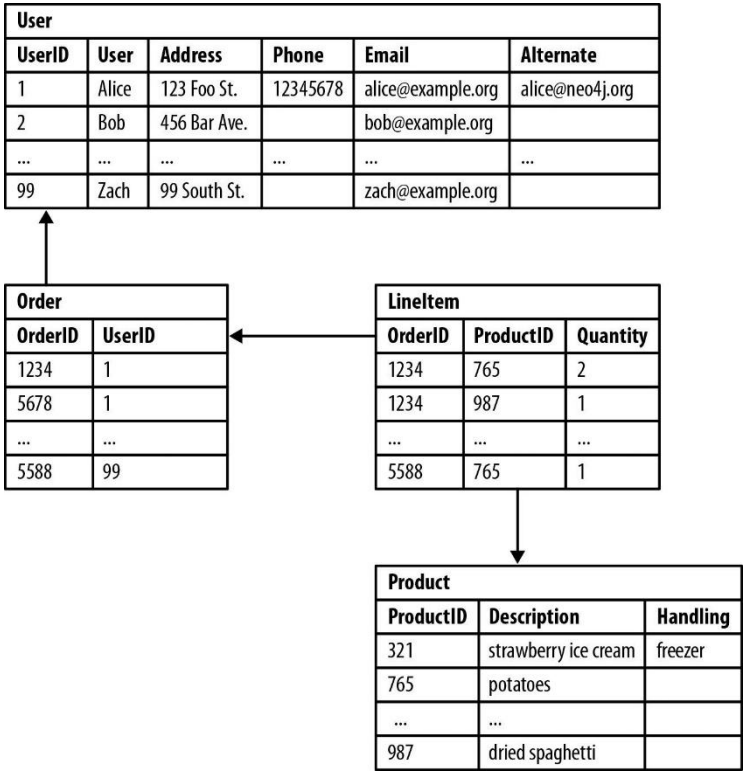
Foreign key

To find related data you need to JOIN tables together

SQL as language to manage data in the database

Relational database characteristics

- Structured data
- Normalized
  - No duplication
- ACID transactions
- Fixed schema
- SQL as query language
- Scale vertically
  - Difficult to scale horizontally



User

UserID	User	Address	Phone	Email	Alternate
1	Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
2	Bob	456 Bar Ave.		bob@example.org	
...	...	...	...	...	...
99	Zach	99 South St.		zach@example.org	

Order

OrderID	UserID
1234	1
5678	1
...	...
5588	99

LineItem

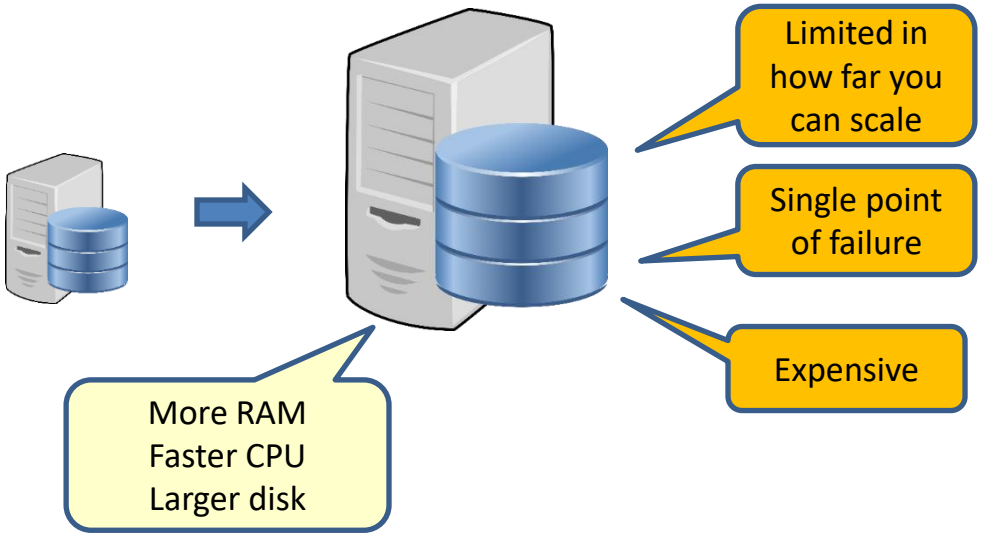
OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...	...	...
5588	765	1

Product

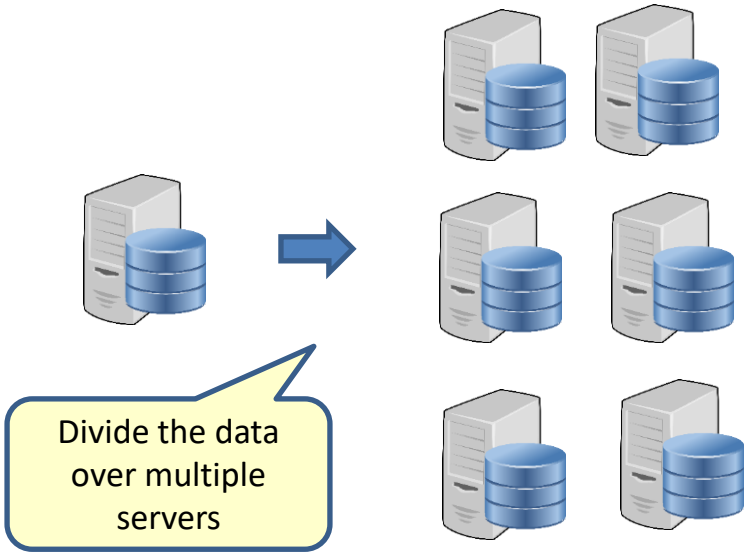
ProductID	Description	Handling
321	strawberry ice cream	freezer
765	potatoes	
...	...	
987	dried spaghetti	

Database Scaling

- Vertical scaling



- Horizontal scaling



Horizontal scaling

- 2 techniques
  1. Sharding/partitioning: Divide the data over the database instances
  2. Replication: make a copy of the data on different database instances

Sharding

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

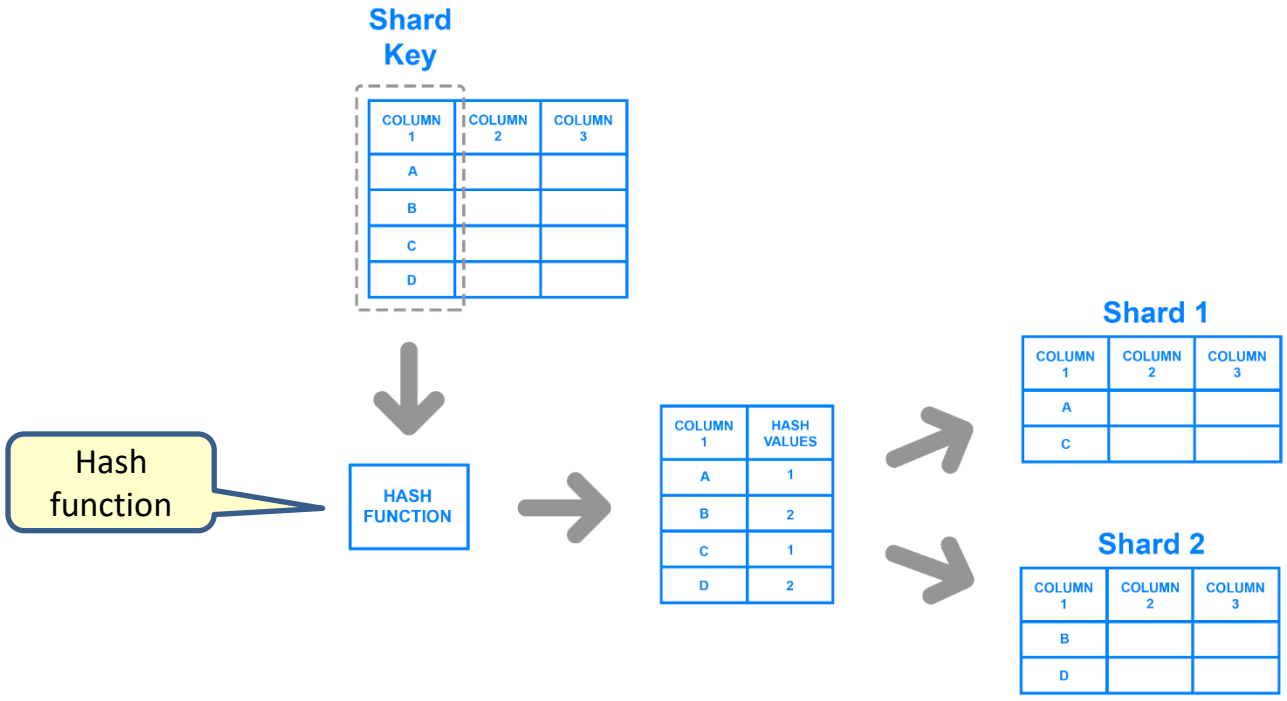
HP1

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN

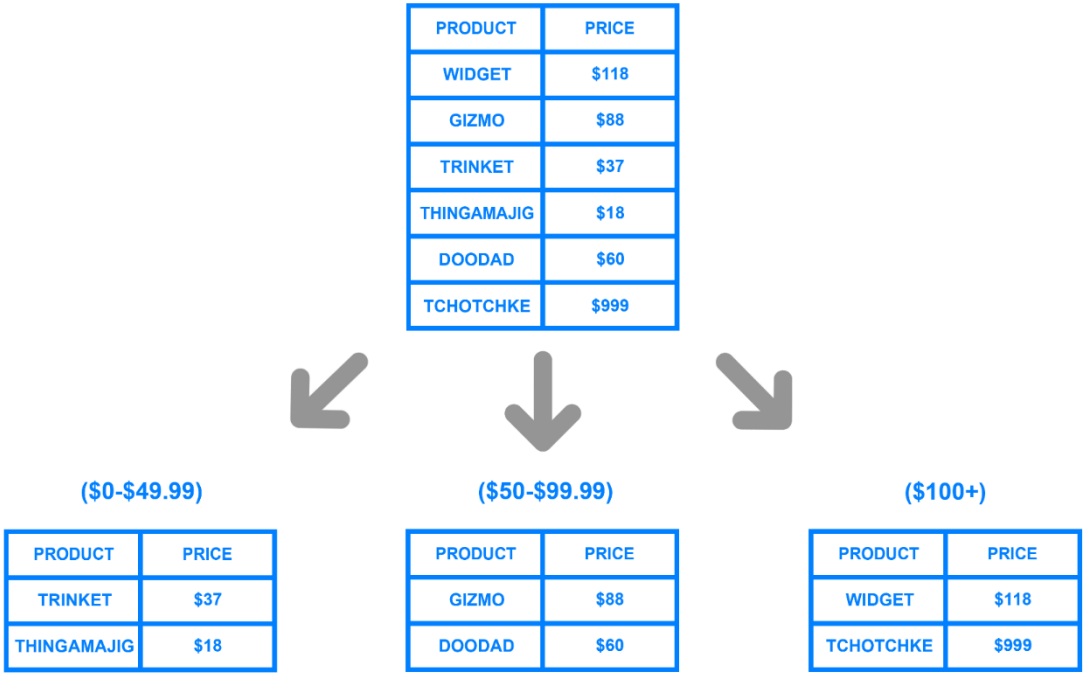
HP2

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

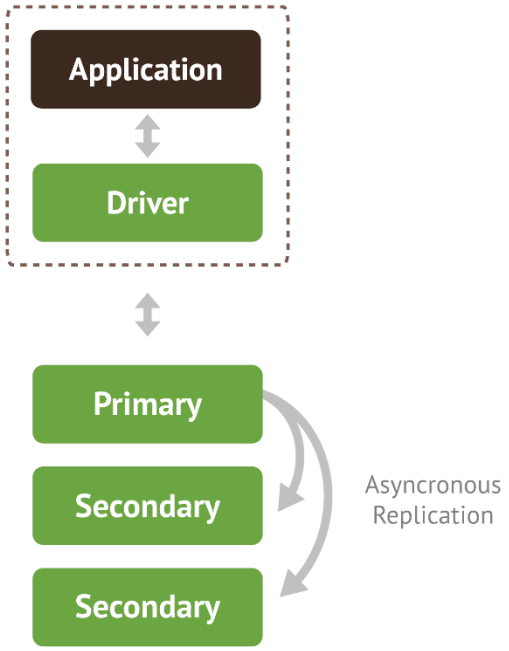
Key based sharding



Range based sharding



Replica Sets



- Replica Set – two or more copies
- Failover
  - “Self-healing” shard
- Addresses many concerns:
  - High Availability
  - Disaster Recovery
  - Maintenance

Relational database advantages

- Data consistency and integrity
  - ACID transactions
- Data accuracy
  - No duplication
- Standard query language (SQL)
- Years of experience
- Structured database design
  - Normalization

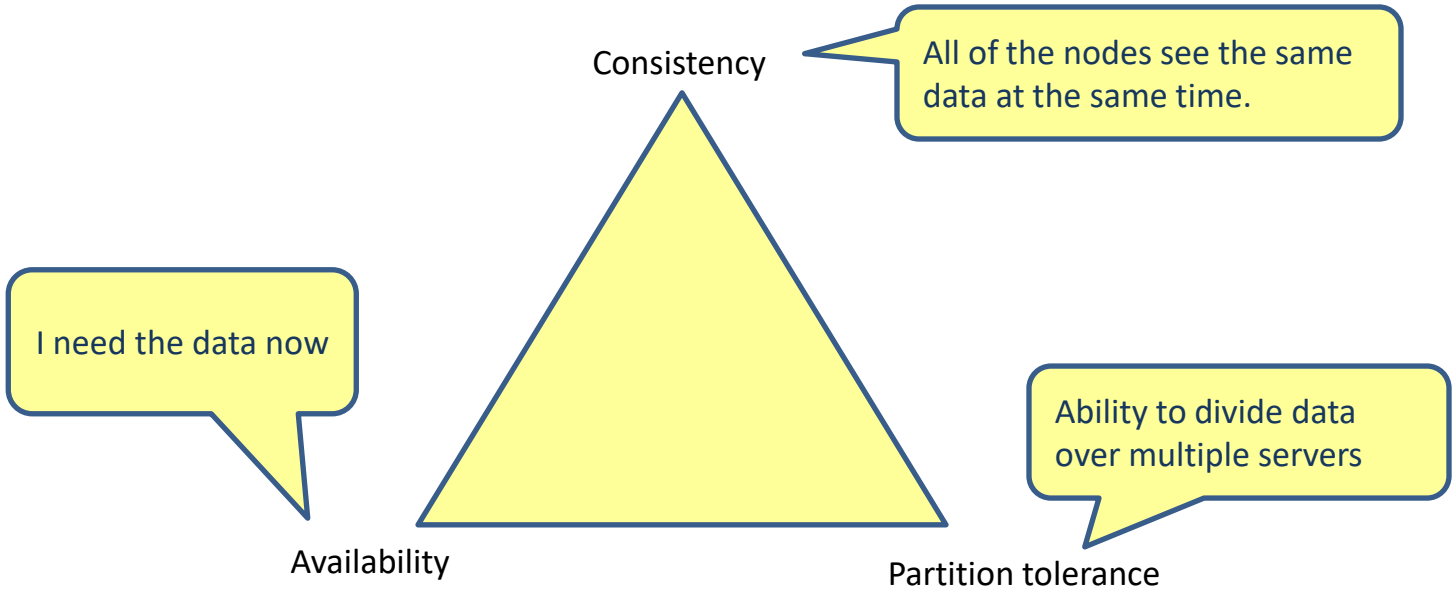
Problems with relational databases

- Scaling writes are is very difficult and limited
  - Vertical scaling is limited and is expensive
  - Horizontal scaling is limited and is complex
    - Queries work only within shards
    - Strict consistency and partition tolerance leads to availability problems

A relational database is hard to scale

Brewer‘s CAP Theorem

- A distributed system can support only two of the following characteristics



Consistency

- Strict consistency
  - The data that I read is always correct
  - You never loose data
- Eventual consistency
  - The data might not be correct
    - But will eventually become correct

Problems with relational databases

- The schema in a database is fixed
- Schema evolution
  - Adding attributes to an object => have to add columns to table
  - You need to do a migration project
  - Application downtime ...

A relational database is hard to change

Problems with relational databases

- Relational schema doesn‘t easily handle unstructured and semi-structured data
  - Emails
  - Tweets
  - Pictures
  - Audio
  - Movies
  - Text

Unstructured data

The university has 5600 students.
John's ID is number 1, he is 18 years old and already holds a B.Sc. degree.
David's ID is number 2, he is 31 years old and holds a Ph.D. degree. Robert's ID is number 3, he is 51 years old and also holds the same degree as David, a Ph.D. degree.

Semi-structured data

```
<University>
  <Student ID="1">
    <Name>John</Name>
    <Age>18</Age>
    <Degree>B.Sc.</Degree>
  </Student>
  <Student ID="2">
    <Name>David</Name>
    <Age>31</Age>
    <Degree>Ph.D. </Degree>
  </Student>
  ....
</University>
```

Structured data

ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.

A relational database does not handle unstructured and semi structured data very well

Relational database disadvantages

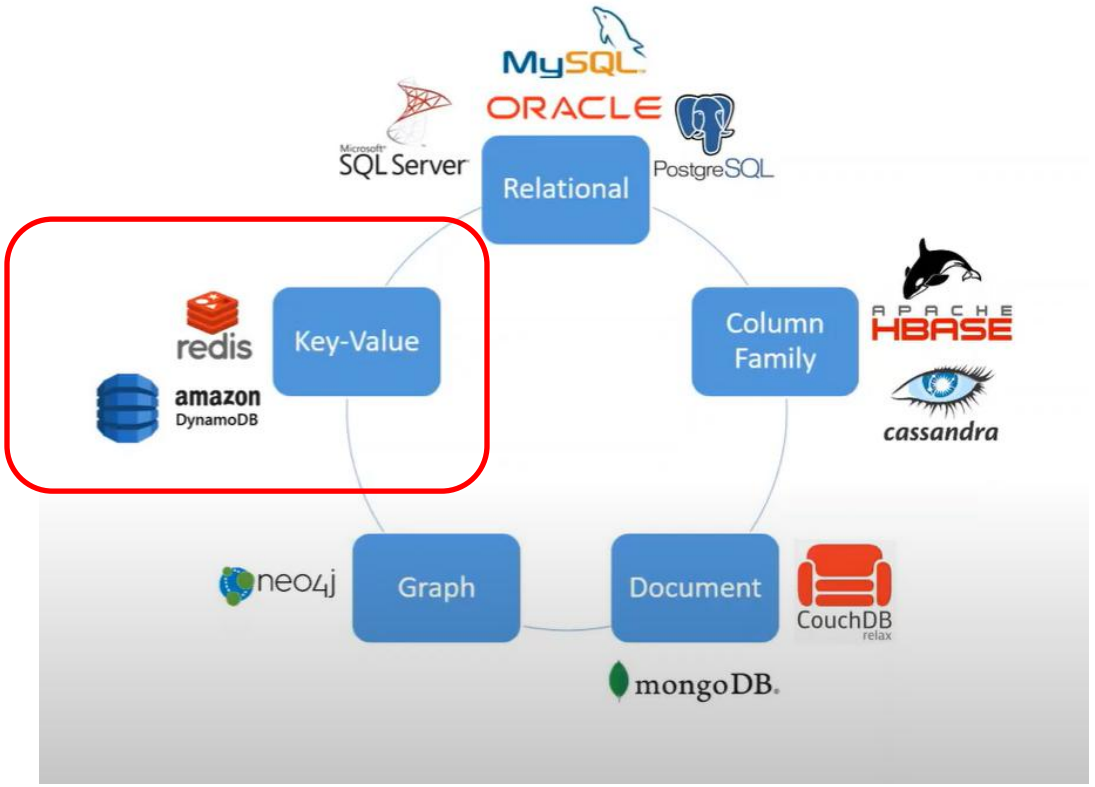
- Schema evolution is difficult
- Horizontal scaling is difficult
- Does not handle unstructured data very well
  - Not good in full text search
- Queries can be slow due to joins

Non relational datases

- Schema free
- Distributed
  - Horizontal scalable
  - Replicated
- Some support unstructured or semi-structured data
- BASE

BASE

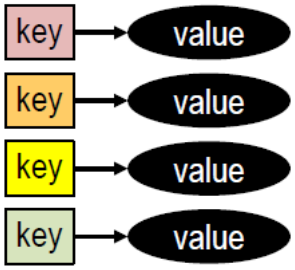
- Basically available
  - All users can concurrently access the data without having to wait
- Soft state
  - Data can have transient or temporary states that change over time
- Eventual consistency
  - The data will become consistent eventually



KEY VALUE STORE

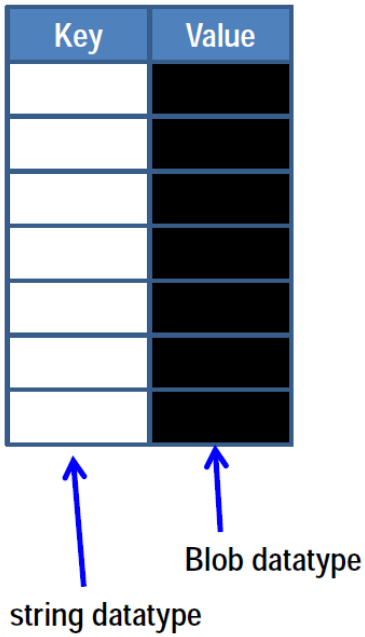
Key-value store

- One key -> one value



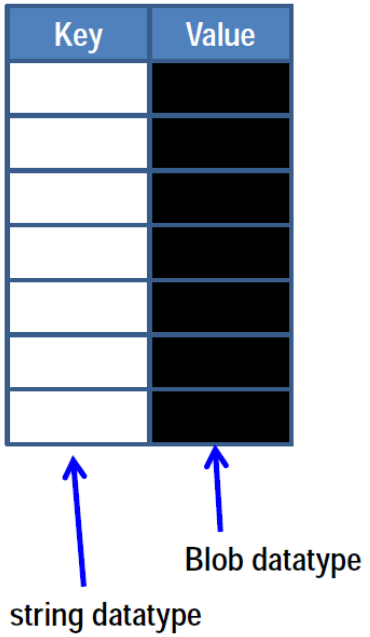
Key-value store

- One key -> one value
  - Simple hash table
  - Very fast
- Value is a binary object (BLOB)
  - DB does not understand the content
- Use cases
  - Storing session data
  - User profiles and preferences
  - Shopping cart data



Key-value store

- **Pros:**
  - Very fast
  - Scalable
  - Simple API
    - Put(key, value)
    - Get(key)
    - Delete(key)
- **Cons:**
  - No way to query based on the content of the value

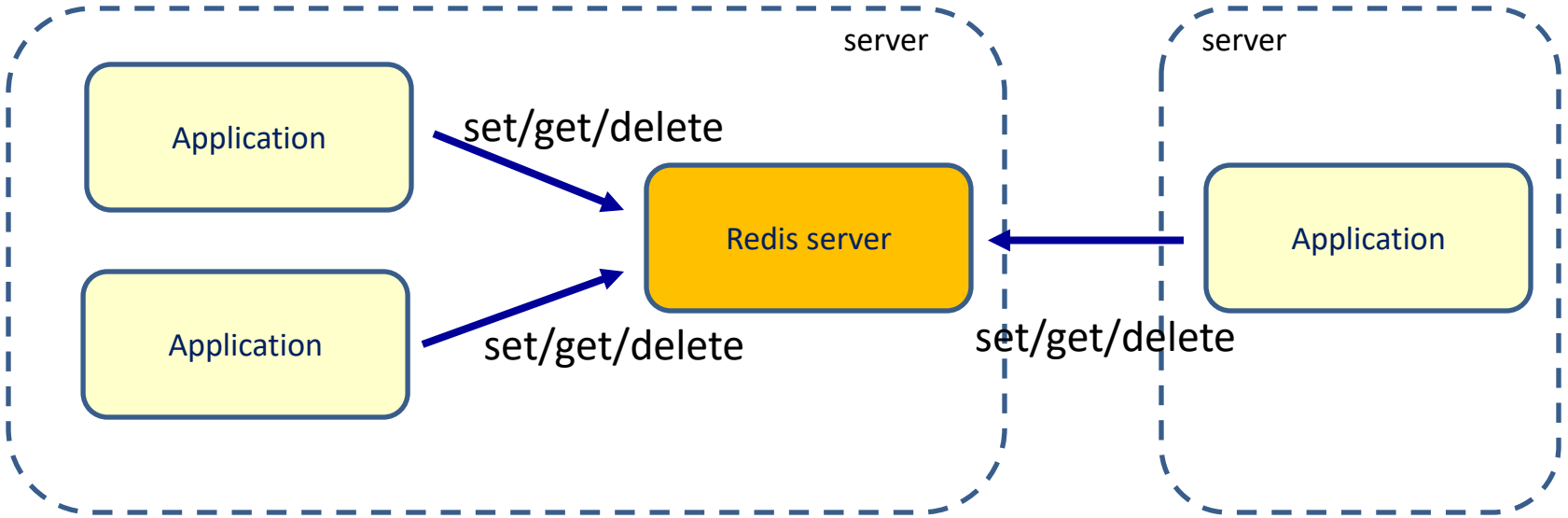


REDIS

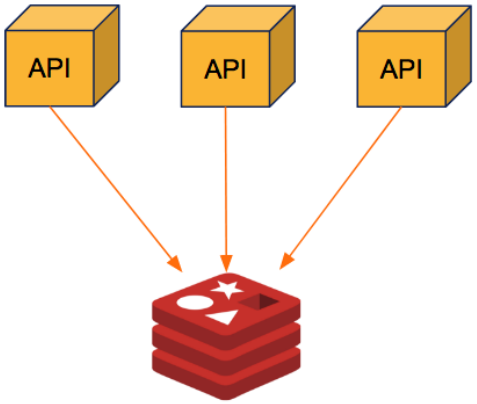
Redis

- **RE**mote **DI**ctionary **S**ervice
- Key-value store
- Very, very fast
  - Microseconds (not milliseconds)
- “in-memory” database
  - Can periodically write to disk
  - Size of data is limited by amount of memory in the sever

Using redis



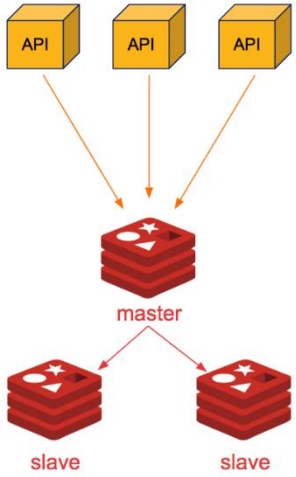
One redis instance



No failover
No replication
No sharding

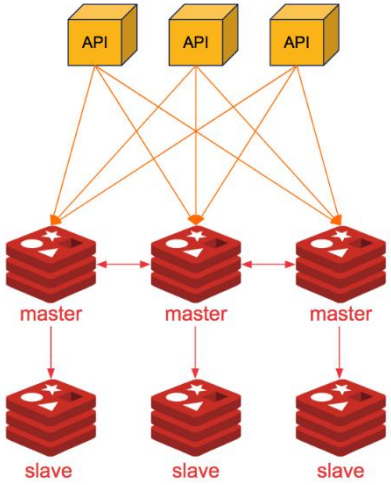
Master slave replication and sentinel

Master slave replication



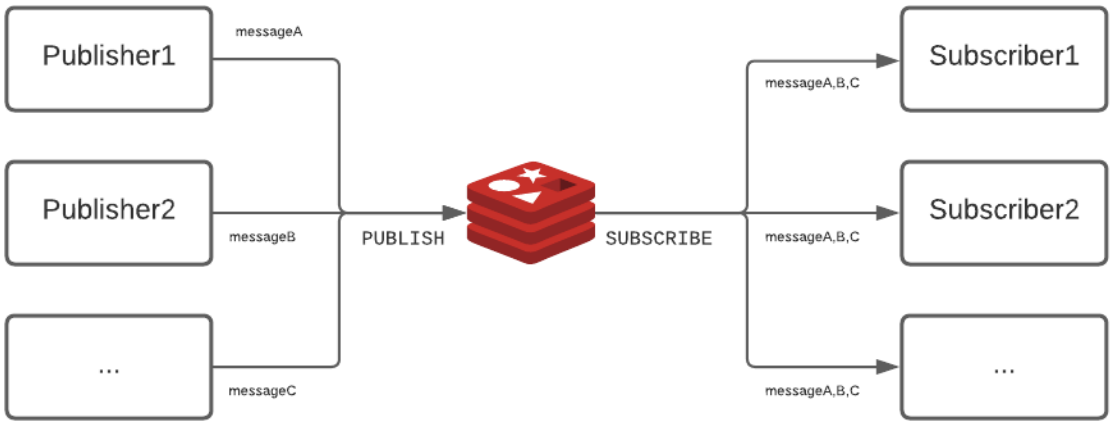
- Failover because of replication
- Slaves are read only
- No automatic failover

Redis cluster



- Automatic failover
- Replication
- Key based sharding

Redis streams



Publish-subscribe functionality

Redis advantages & disadvantages

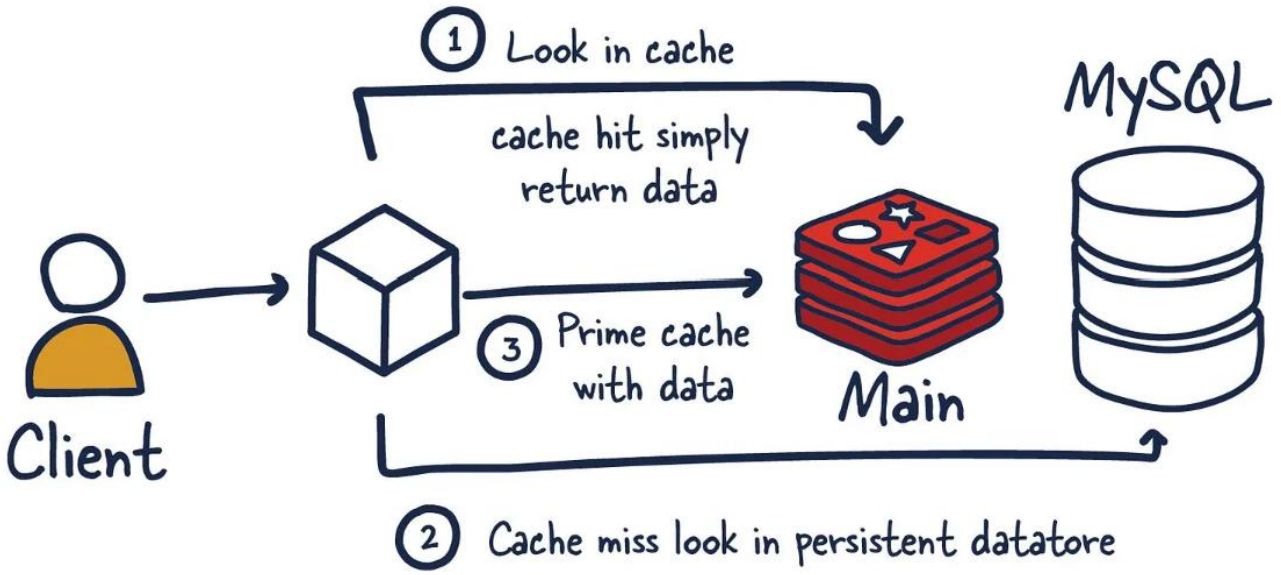
- Advantages
  - Very fast in memory persistence
  - Scalable
  - High available
  - Message functionality
    - Pub-sub
- Disadvantages
  - Not for long term data persistence
  - You cannot query the value
    - No query language
  - Storage is limited to available memory

Redis use cases

- In memory cache
  - Data that changes infrequently and is requested often
  - Data that is less mission-critical and is frequently evolving
- Session management
- Queue systems
- Often used in combination with another database for permanent storage

Redis use cases

- Often used in combination with another database for permanent storage

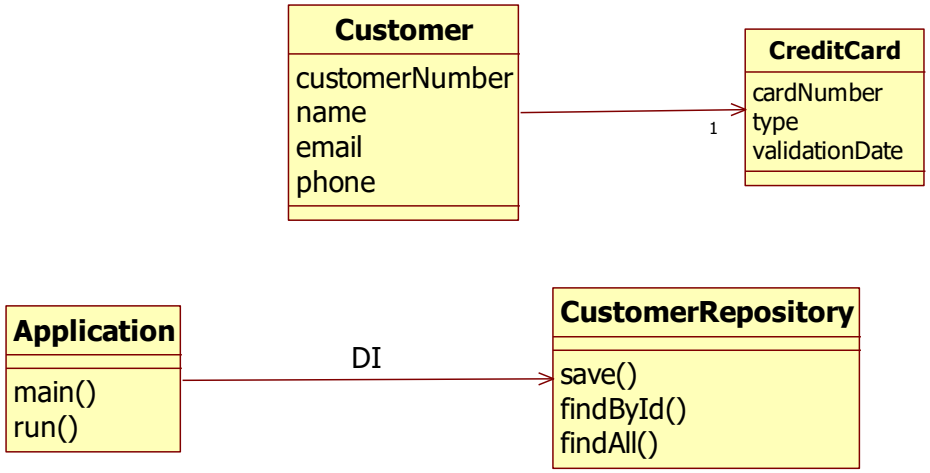


SPRING BOOT REDIS EXAMPLE

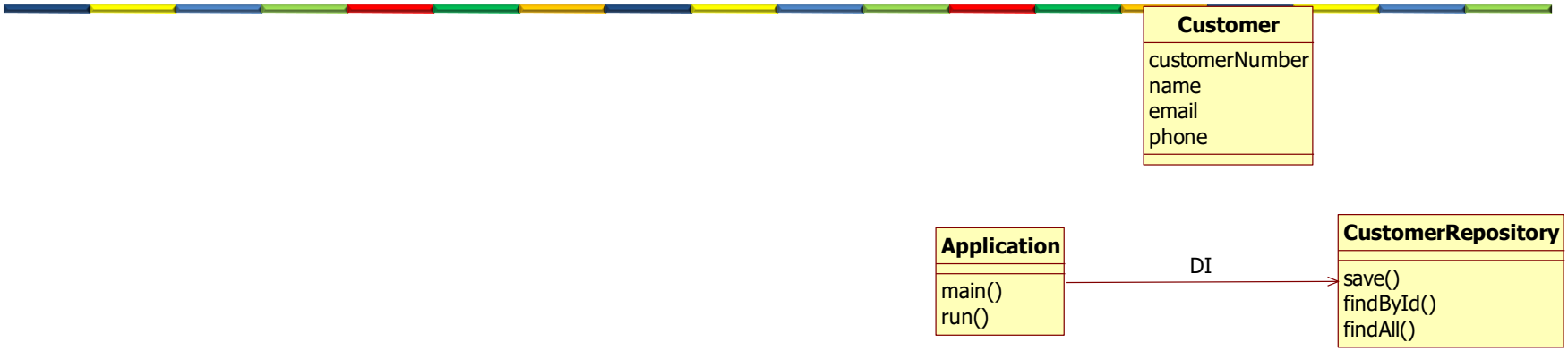
Domain classes

```java
@RedisHash("Customer")
public class Customer {
  @Id
  private int customerNumber;
  private String name;
  private String email;
  private String phone;
  private CreditCard creditCard;

...
```

```java
public class CreditCard {
  private String cardNumber;
  private String type;
  private String validationDate;

...
```



Repository



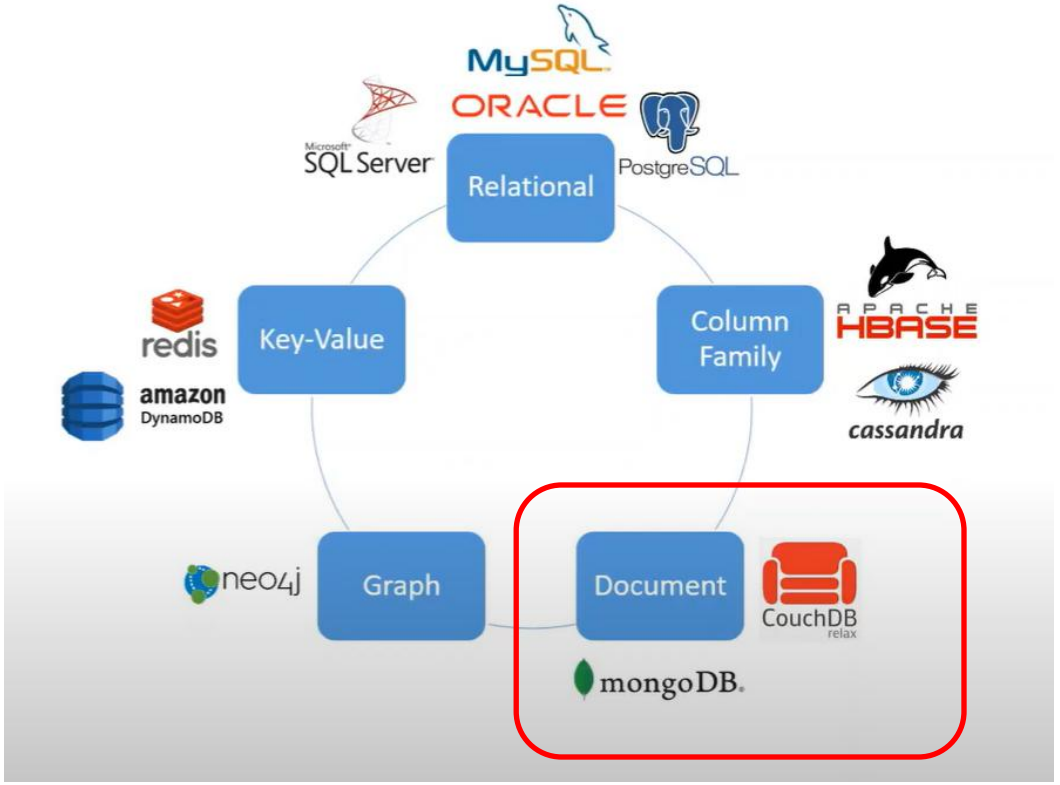

```
@Repository
public interface CustomerRepository extends CrudRepository<Customer, Integer> {}
```

application.properties

```
# Redis configuration.
spring.redis.host=localhost
spring.redis.port=6379


logging.level.root=ERROR
logging.level.org.springframework=ERROR
```

Application

```
public void run(String... args) throws Exception {
    // create customer
  Customer customer = new Customer(101,"John doe", "johnd@acme.com", "0622341678");
  CreditCard creditCard = new CreditCard("12324564321", "Visa", "11/23");
  customer.setCreditCard(creditCard);
  customerRepository.save(customer);
  customer = new Customer(66,"James Johnson", "jj123@acme.com", "068633452");
  creditCard = new CreditCard("99876549876", "MasterCard", "01/24");
  customer.setCreditCard(creditCard);
  customerRepository.save(customer);
  //get customers
  System.out.println(customerRepository.findById(66).get());
  System.out.println(customerRepository.findById(101).get());
  System.out.println("-----------All customers ----------------");
  System.out.println(customerRepository.findAll());
  //update customer
  customer = customerRepository.findById(101).get();
  customer.setEmail("jd@gmail.com");
  customerRepository.save(customer);
  //delete customer
  customerRepository.deleteById(66);
  System.out.println("-----------All customers ----------------");
  System.out.println(customerRepository.findAll());
}
```



DOCUMENT DATABASE

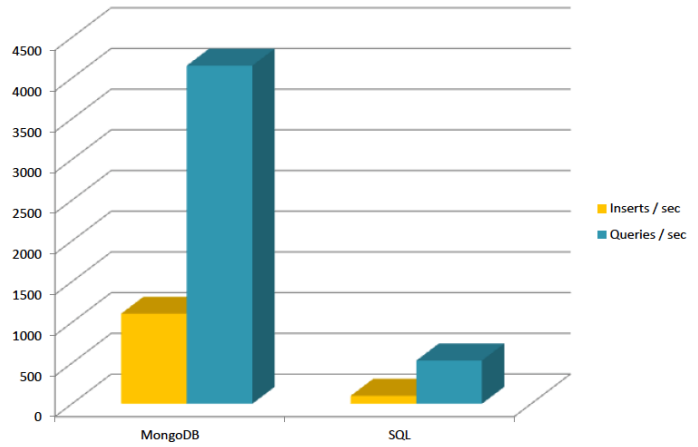
Document database

- Store and retrieve documents
  - Like files in directories
- No schema
- Store documents in JSON

MONGODB

MongoDB

- Document database
- Fast
- Can handle large datasets



Mongo terminology

RDBMS	MongoDB
Database	Database
Table	Collection
Index	Index
Row	Document
Join	Embedding & Linking

Document data model (JSON)

Relational - Tables

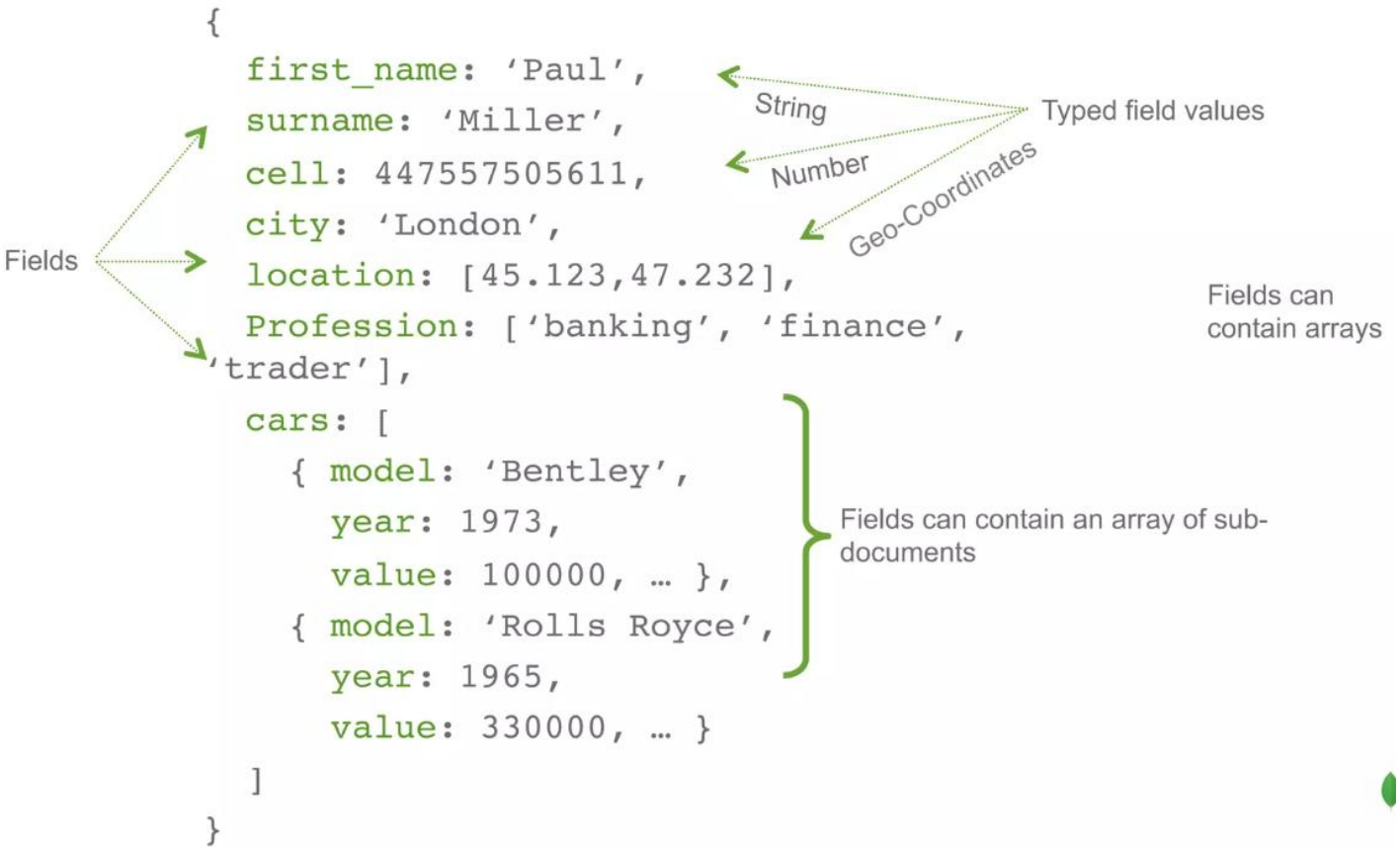
Customer ID	First Name	Last Name	City
0	John	Doe	New York
1	Mark	Smith	San Francisco
2	Jay	Black	Newark
3	Meagan	White	London
4	Edward	Daniels	Boston

Account Number	Branch ID	Account Type	Customer ID
10	100	Checking	0
11	101	Savings	0
12	101	IRA	0
13	200	Checking	1
14	200	Savings	1
15	201	IRA	2

Document - Collections

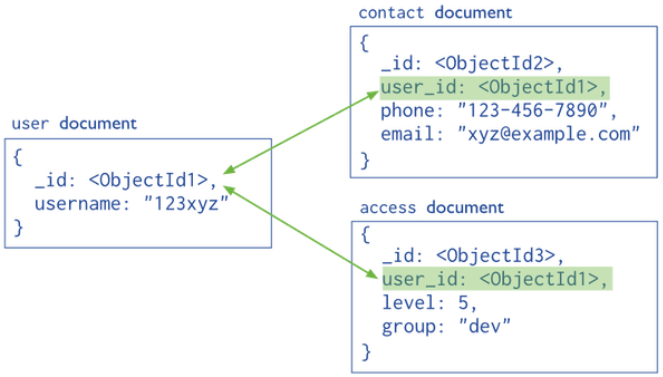
```
{   customer_id : 1,
    first_name : "Mark",
    last_name : "Smith",
    city : "San Francisco",
    accounts : [   {
        account_number : 13,
        branch_ID : 200,
        account_type : "Checking"
    },
    {   account_number : 14,
        branch_ID : 200,
        account_type : ”IRA”,
        beneficiaries: […]
    } ]
}
```

Documents are rich data structures



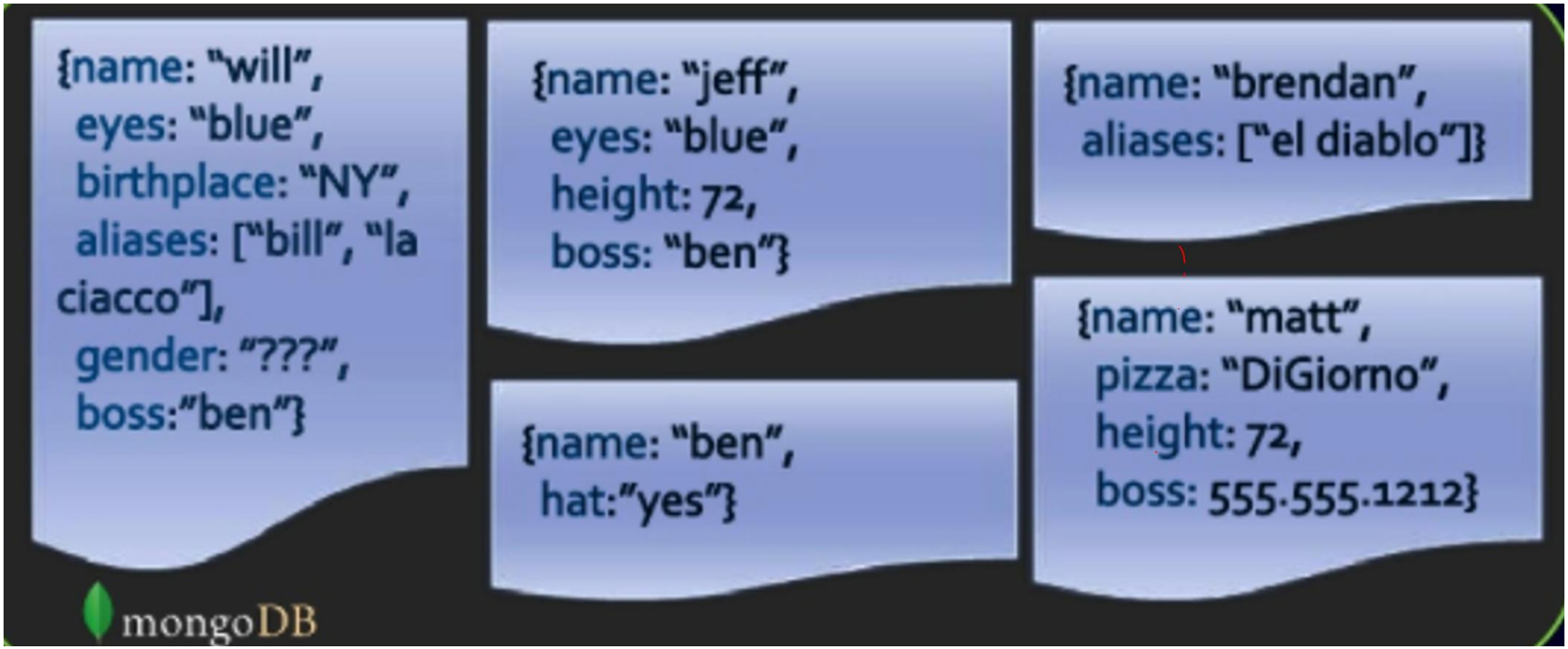
References vs. embedded data

Reference



Embedded data

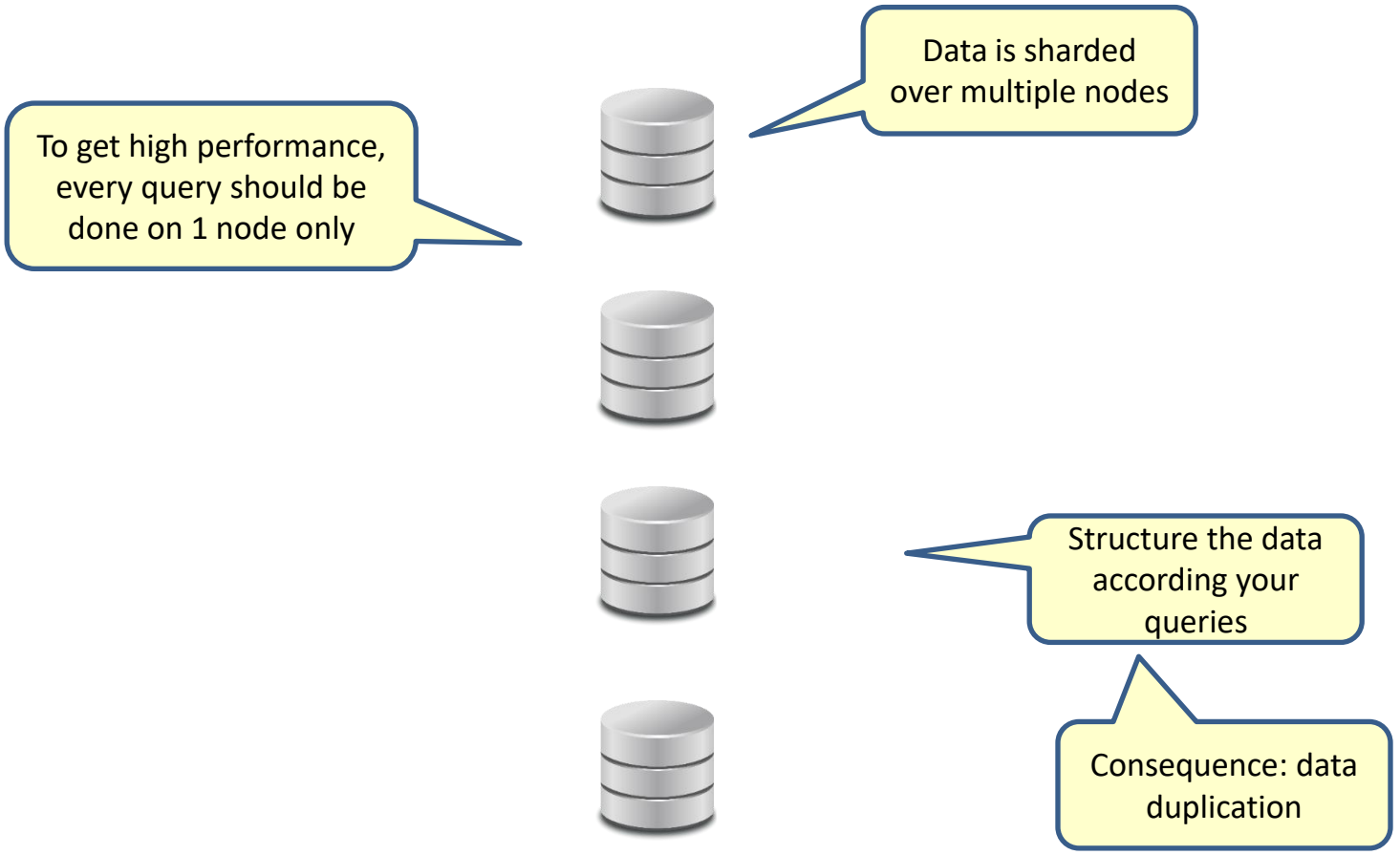
Schema free



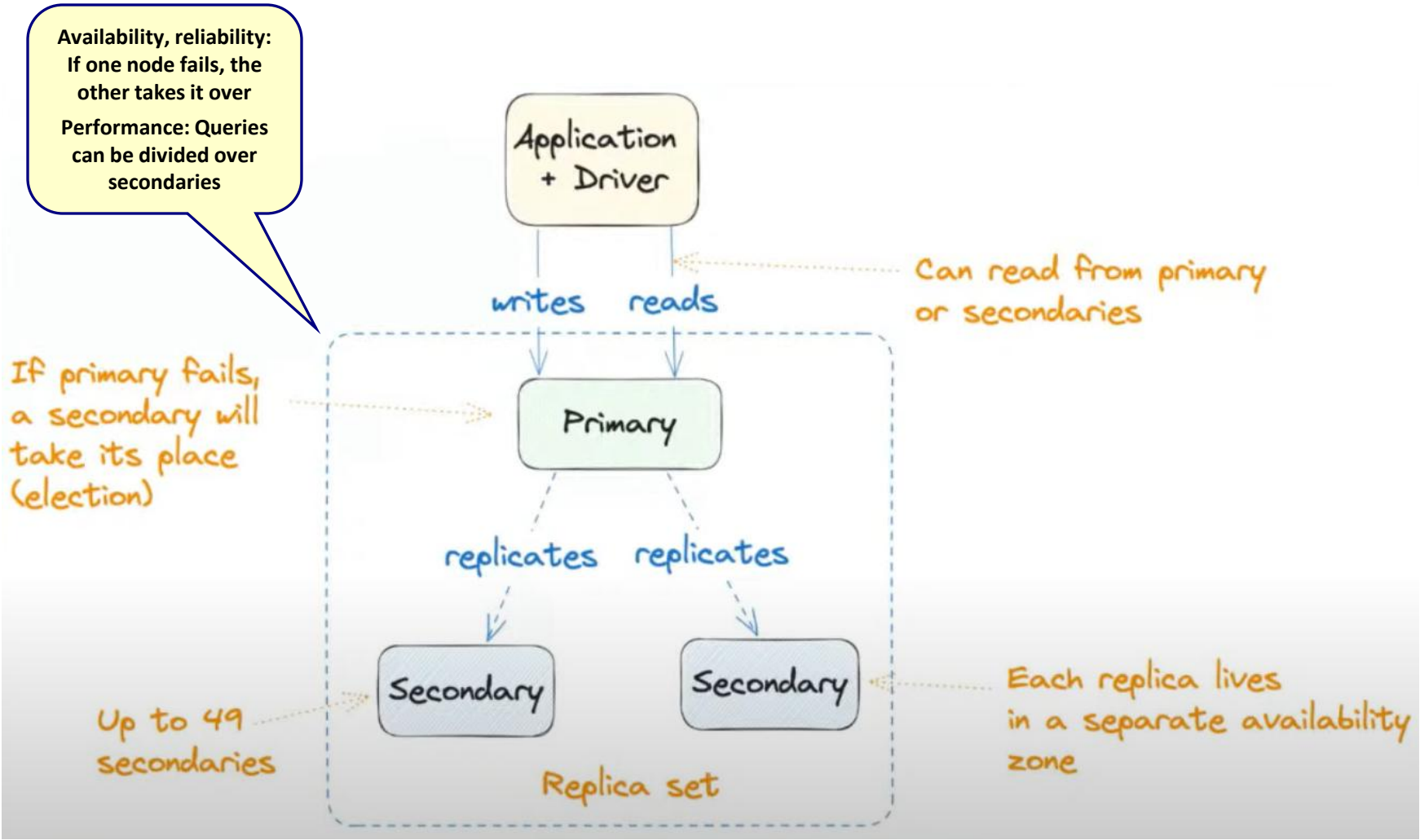
Find() method

SQL SELECT Statements	MongoDB find() Statements
`SELECT * FROM users`	`db.users.find()`
`SELECT id, user_id, status FROM users`	`db.users.find( { }, { user_id: 1, status: 1 } )`
`SELECT user_id, status FROM users`	`db.users.find( { }, { user_id: 1, status: 1, _id: 0 } )`
`SELECT * FROM users WHERE status = "A"`	`db.users.find( { status: "A" } )`
`SELECT user_id, status FROM users WHERE status = "A"`	`db.users.find( { status: "A" }, { user_id: 1, status: 1, _id: 0 } )`
`SELECT * FROM users WHERE status != "A"`	`db.users.find( { status: { $ne: "A" } } )`
`SELECT * FROM users WHERE status = "A" AND age = 50`	`db.users.find( { status: "A", age: 50 } )`
`SELECT * FROM users WHERE status = "A" OR age = 50`	`db.users.find( { $or: [ { status: "A" } , { age: 50 } ] } )`
`SELECT * FROM users WHERE age > 25`	`db.users.find( { age: { $gt: 25 } } )`

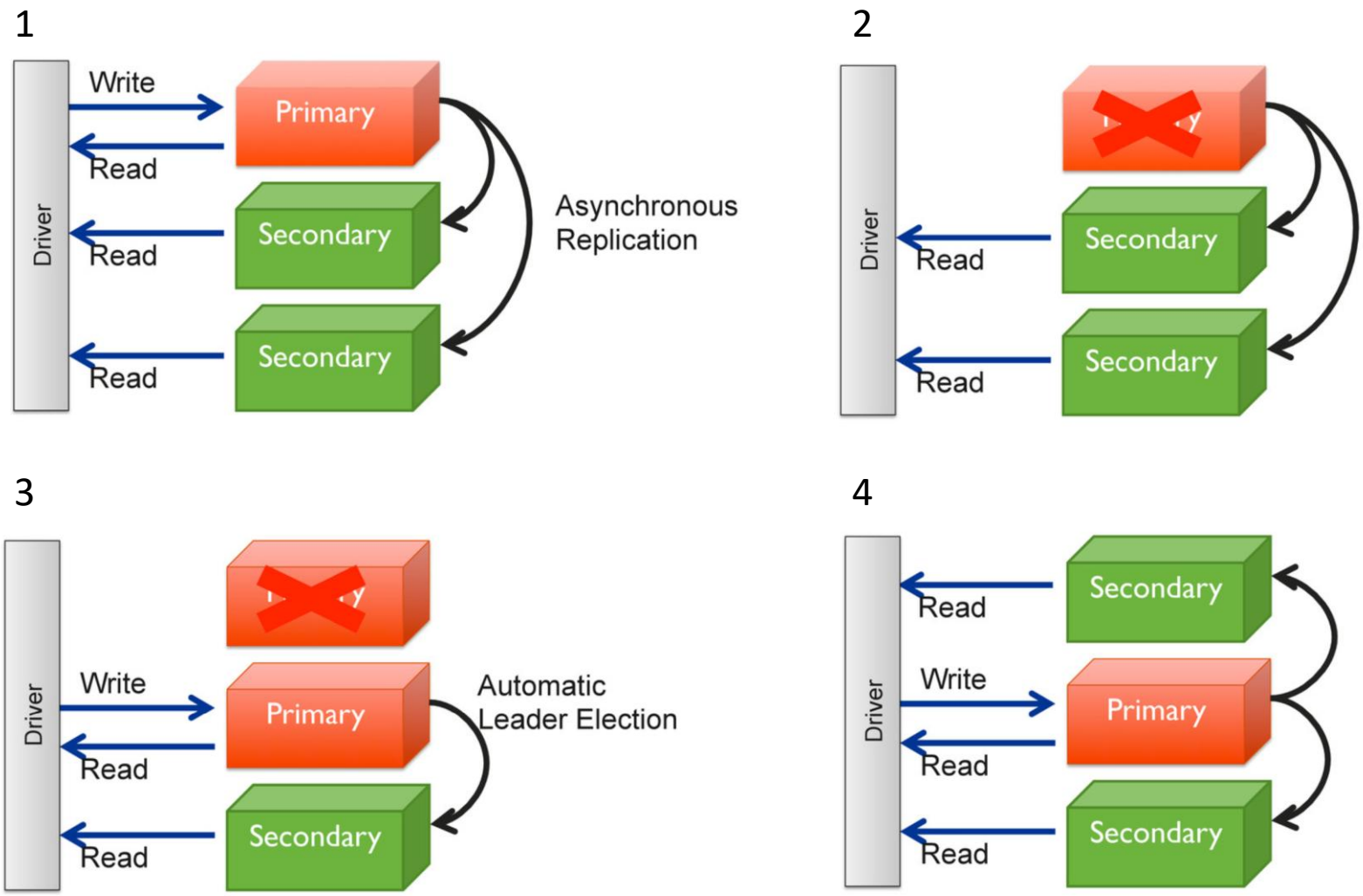
How to structure data in mongo?



MongoDB replication

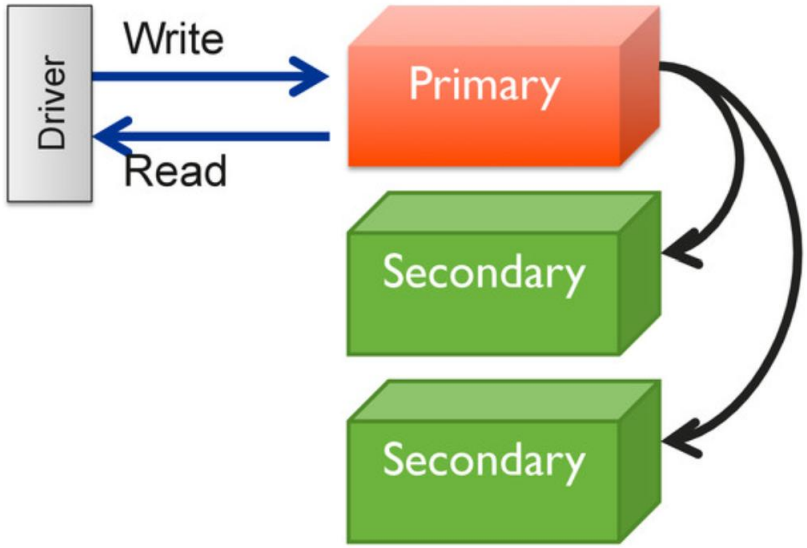


Replica sets

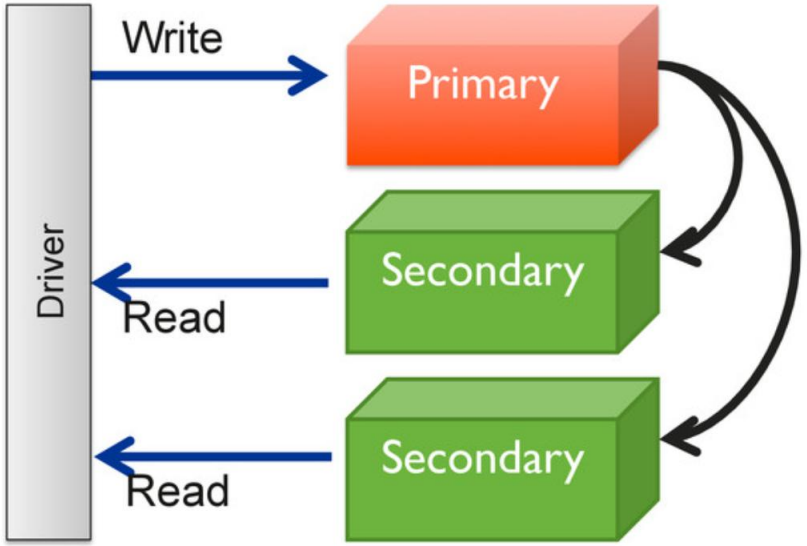


Consistency

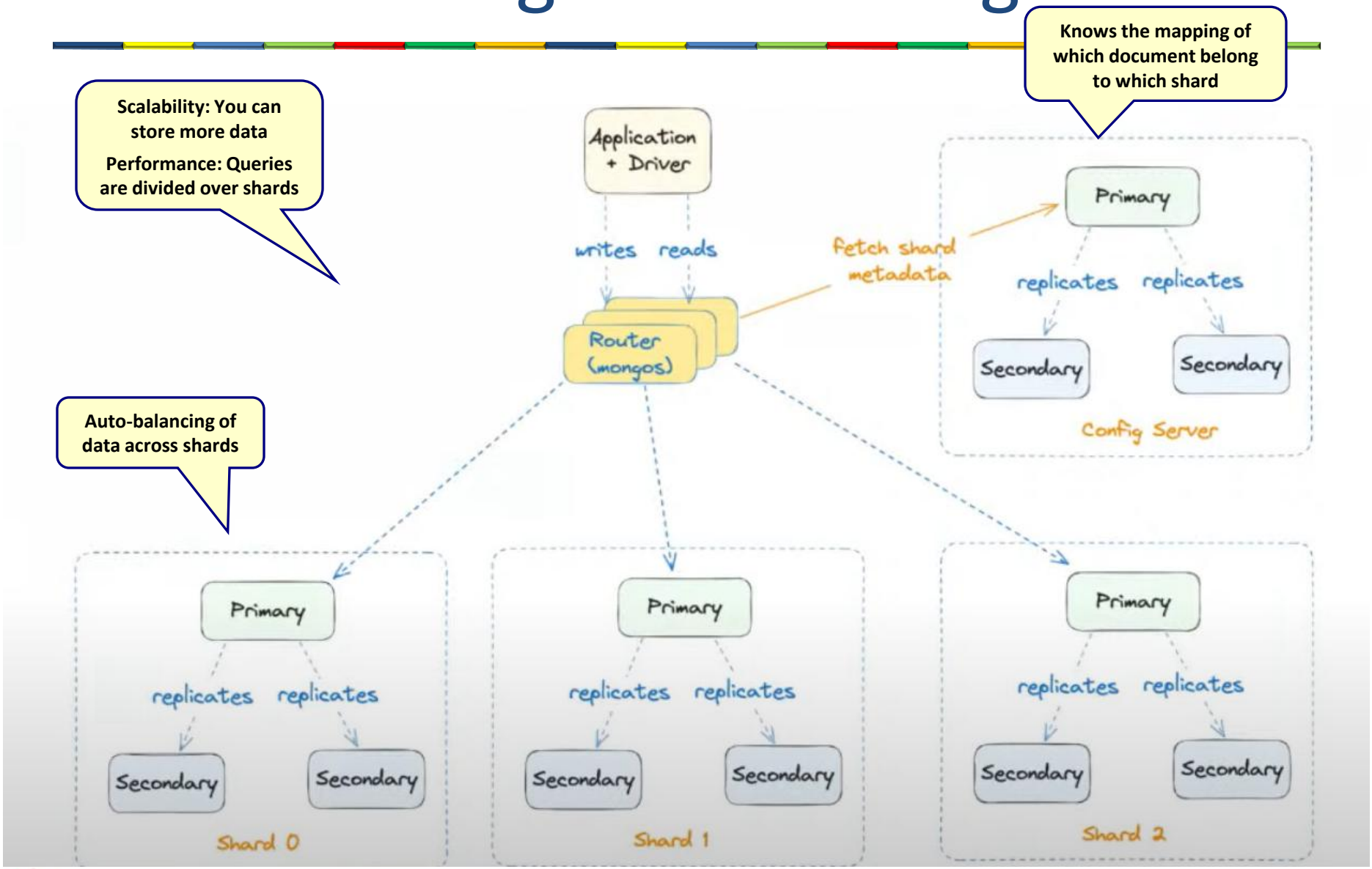
Strong consistency



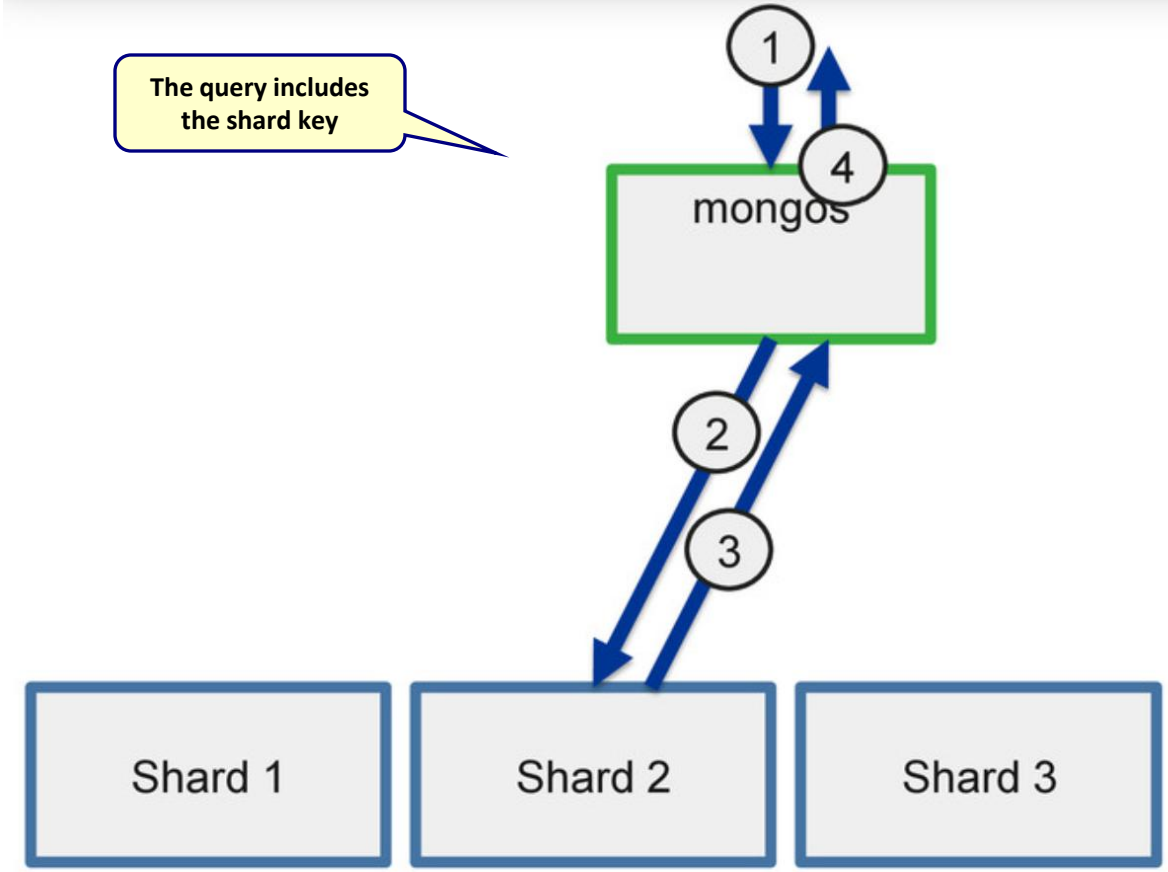
Eventual consistency



MongoDB sharding

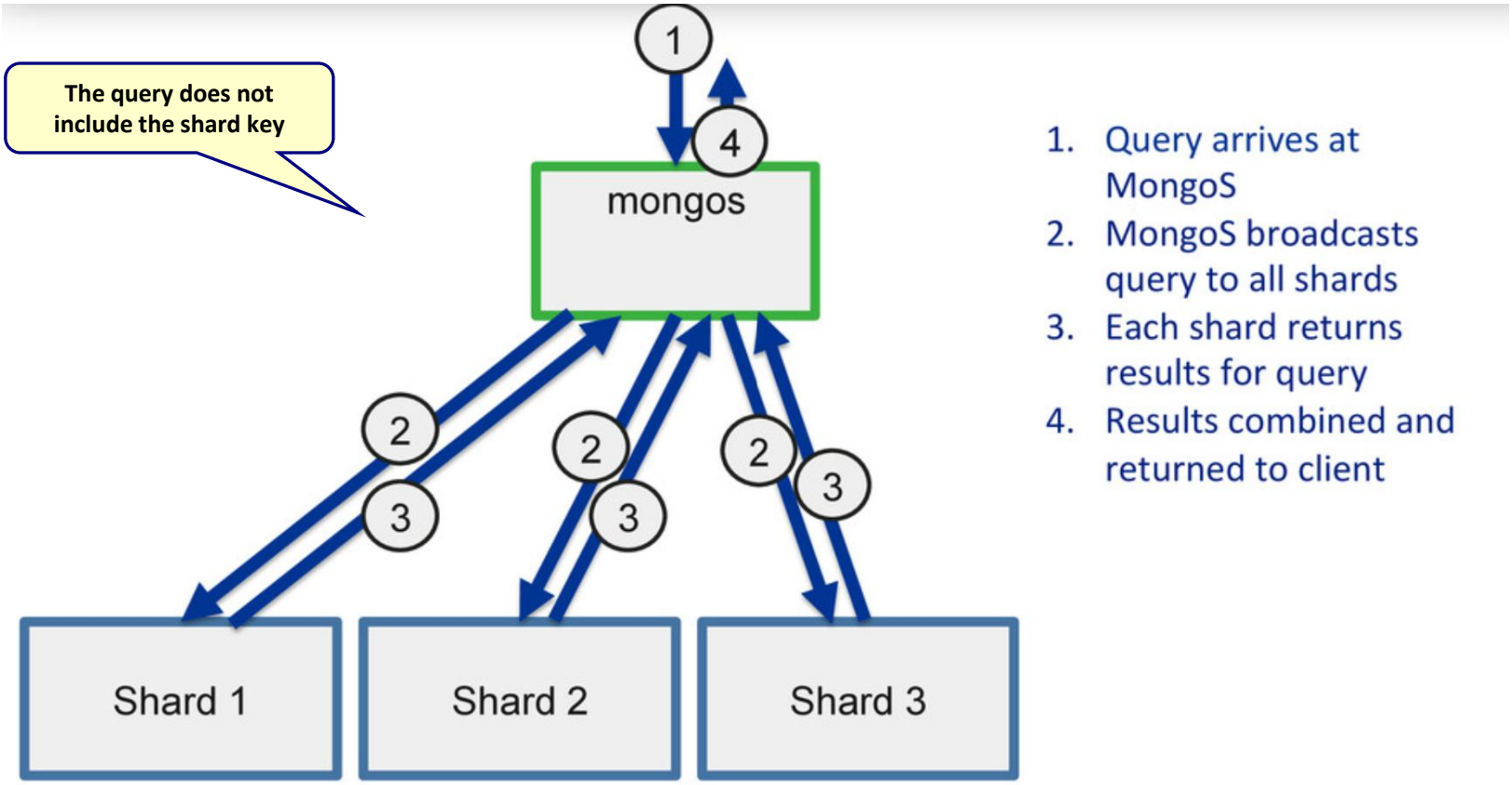


Routed query



1. Query arrives at MongoS
2. MongoS routes query to a single shard
3. Shard returns results of query
4. Results returned to client

Scatter gather



1. Query arrives at MongoS
2. MongoS broadcasts query to all shards
3. Each shard returns results for query
4. Results combined and returned to client

Mongo advantages & disadvantages

- Advantages
  - Very fast
  - Scalable
  - High available
  - Handles unstructured data
  - Rich query language
  - Indexing
  - No schema
  - ACID transaction support
- Disadvantages
  - Limited joins
  - Data redundancy
  - Document size limit (16 MB)
  - Limited nested documents levels (100)

Mongo use cases

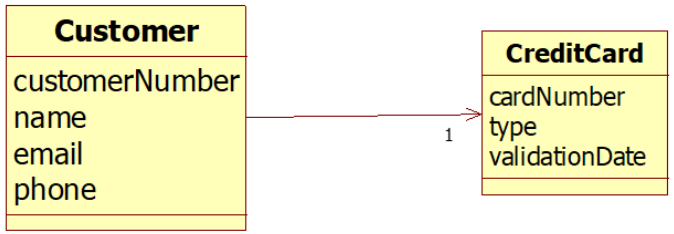
- IoT
  - Large growing data sets, flexible schema
- Real-time analytics
  - Large growing data sets, flexible schema
- Content Management
  - Large growing data sets, flexible schema
- Product catalog
  - Large growing data sets, flexible schema

SPRING BOOT MONGO EXAMPLE

Customer

```
@Document
public class Customer {
  @Id
  private int customerNumber;
  private String name;
  private String email;
  private String phone;
  private CreditCard creditCard;

...
```

```
public class CreditCard {
   private String cardNumber;
   private String type;
   private String validationDate;

...
```



Repository

```
@Repository
public interface CustomerRepository extends MongoRepository<Customer,
Integer> {
   Customer findByPhone(String phone);
   Customer findByName(String name);


   @Query("{'email' : ?0}")
   Customer findCustomerWithPhone(String email);

   @Query("{'creditCard.type' : ?0}")
   List<Customer>  findCustomerWithCreditCardType(String cctype);
}
```

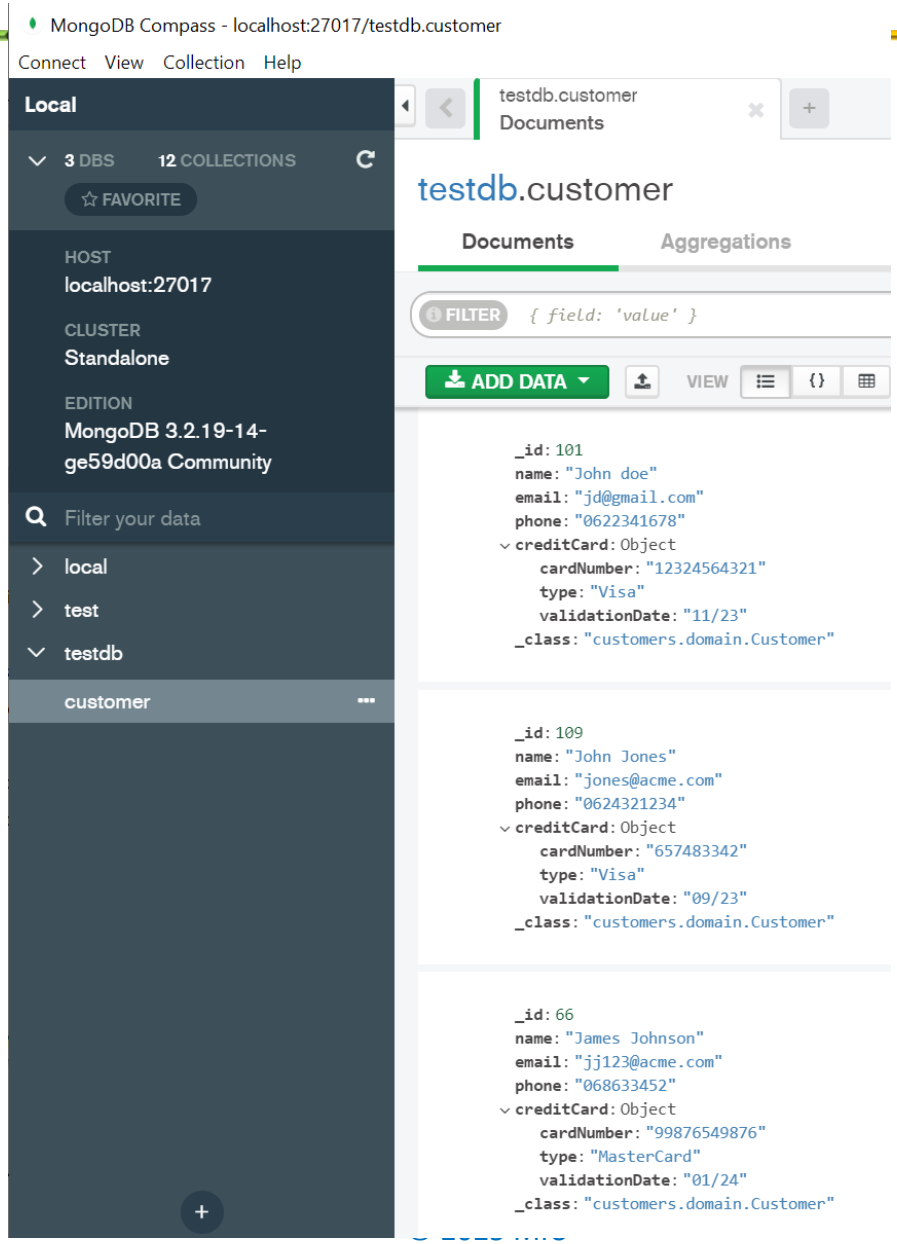
Application (1/2)

```
public void run(String... args) throws Exception {
 // create customer
Customer customer = new Customer(101,"John doe", "johnd@acme.com", "0622341678");
CreditCard creditCard = new CreditCard("12324564321", "Visa", "11/23");
customer.setCreditCard(creditCard);
customerRepository.save(customer);
customer = new Customer(109,"John Jones", "jones@acme.com", "0624321234");
creditCard = new CreditCard("657483342", "Visa", "09/23");
customer.setCreditCard(creditCard);
customerRepository.save(customer);
customer = new Customer(66,"James Johnson", "jj123@acme.com", "068633452");
creditCard = new CreditCard("99876549876", "MasterCard", "01/24");
customer.setCreditCard(creditCard);
customerRepository.save(customer);
//get customers
System.out.println(customerRepository.findById(66).get());
System.out.println(customerRepository.findById(101).get());
```

Application (2/2)

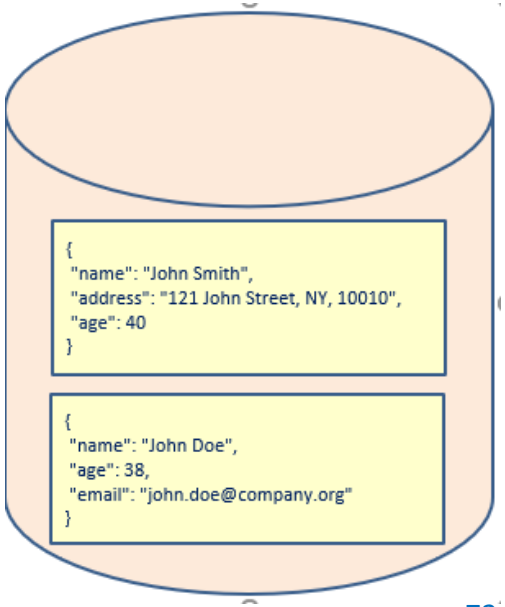
```
System.out.println("-----------All customers ----------------");
 System.out.println(customerRepository.findAll());
 //update customer
 customer = customerRepository.findById(101).get();
 customer.setEmail("jd@gmail.com");
 customerRepository.save(customer);
 System.out.println("-----------find by phone ----------------");
 System.out.println(customerRepository.findByPhone("0622341678"));
 System.out.println("-----------find by email ----------------");
 System.out.println(customerRepository.findCustomerWithPhone("jj123@acme.com"));
 System.out.println("-----------find customers with a certain type of creditcard ----------------");
 List<Customer> customers = customerRepository.findCustomerWithCreditCardType("Visa");
 for (Customer cust : customers){
   System.out.println(cust);
 }

}
```

Mongo collections

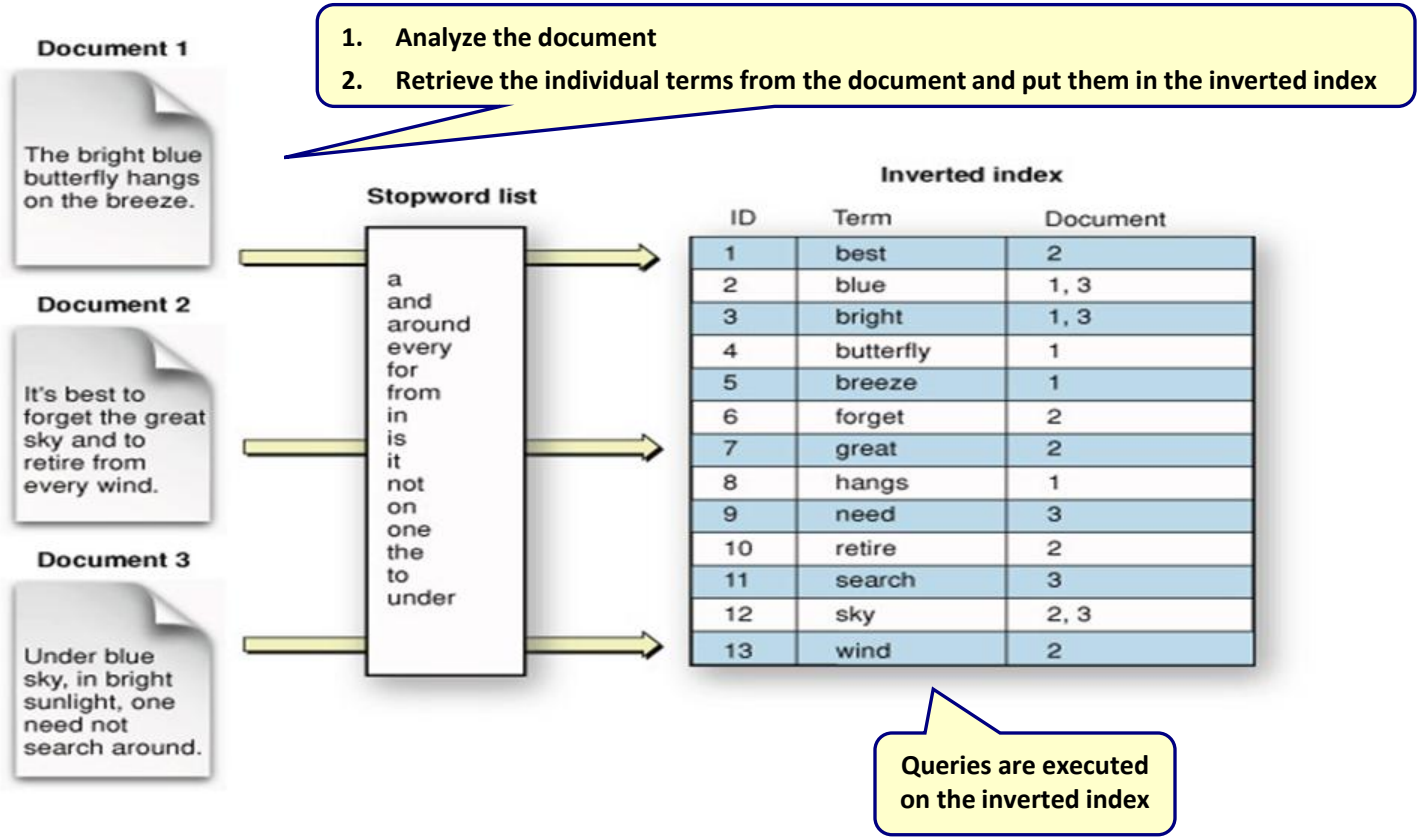


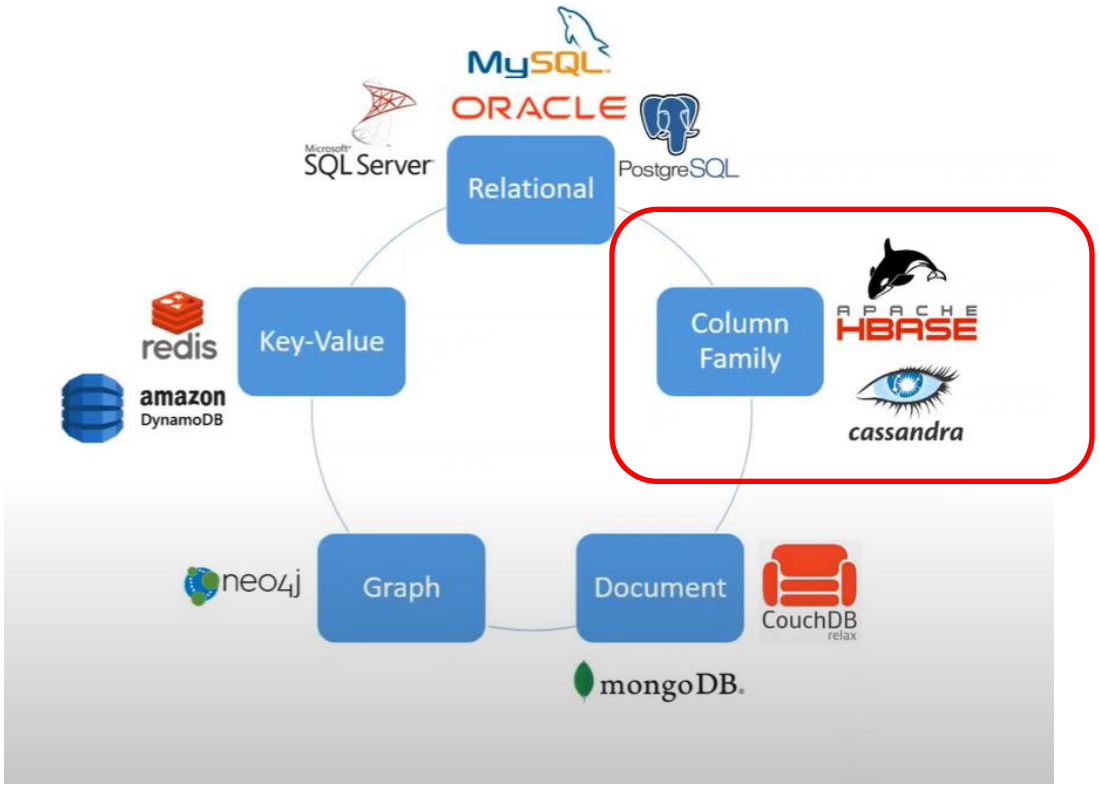
Elasticsearch

- Database
  - Data is stored as documents
  - Data is structured in JSON format
- Full text search engine



Inverted index

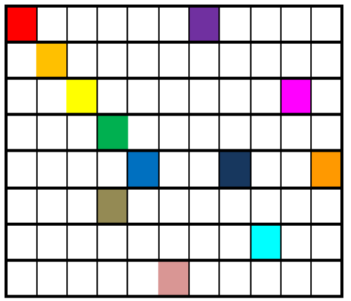




COLUMN FAMILY DATABASE

Column family

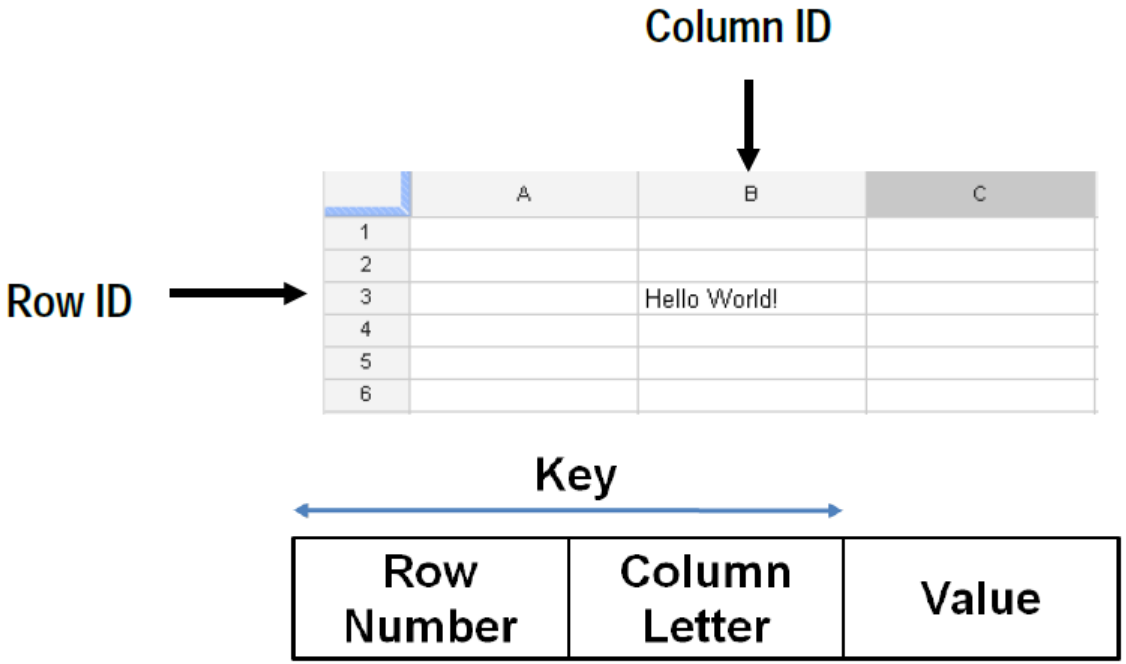
- Place data in a certain column
- Ideal for high-variability data sets
- Colum families allow to query all columns that have a specific property or properties
- Allow new columns to be inserted without doing an "alter table"



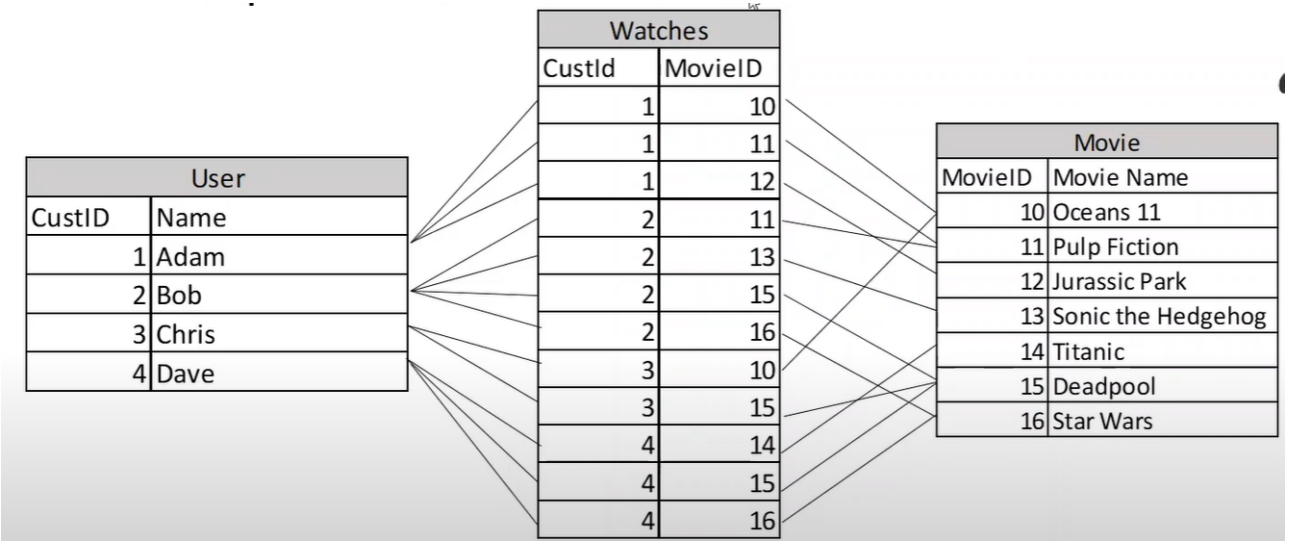
- Players
  - Cassandra
  - HBase
  - Google BigTable

Excel

- The key is the row number and the column letter



Relational



User

CustID	Name
1	Adam
2	Bob
3	Chris
4	Dave

Watches

CustId	MovieID
1	10
1	11
1	12
2	11
2	13
2	15
2	16
3	10
3	15
4	14
4	15
4	16

Movie

MovieID	Movie Name
10	Oceans 11
11	Pulp Fiction
12	Jurassic Park
13	Sonic the Hedgehog
14	Titanic
15	Deadpool
16	Star Wars

What movies has Adam watched? ⇐ Need to join 3 tables (slow)

Who watched Deadpool? ⇐ Need to join 3 tables (slow)

Column family

User	Movies			
	Oceans 11	Pulp Fiction	Jurassic Park	
Adam	TRUE	TRUE	TRUE	
	Pulp Fiction	Sonic	Deadpool	Star Wars
Bob	TRUE	TRUE	TRUE	TRUE
	Oceans 11	Deadpool		
Chris	TRUE	TRUE		
	Titanic	Deadpool	Star Wars	
Dave	TRUE	TRUE	TRUE	

Structure the data according your queries

What movies has Adam watched? ⇐ Very easy and fast

Who watched Deadpool? ⇐ Difficult

CASSANDRA

Cassandra

- Column family NoSQL database
  - Uses tables, primary keys, queries, …
- Unlimited elastically scalable
- Always available (no downtime)
- High performance
- Fault tolerance

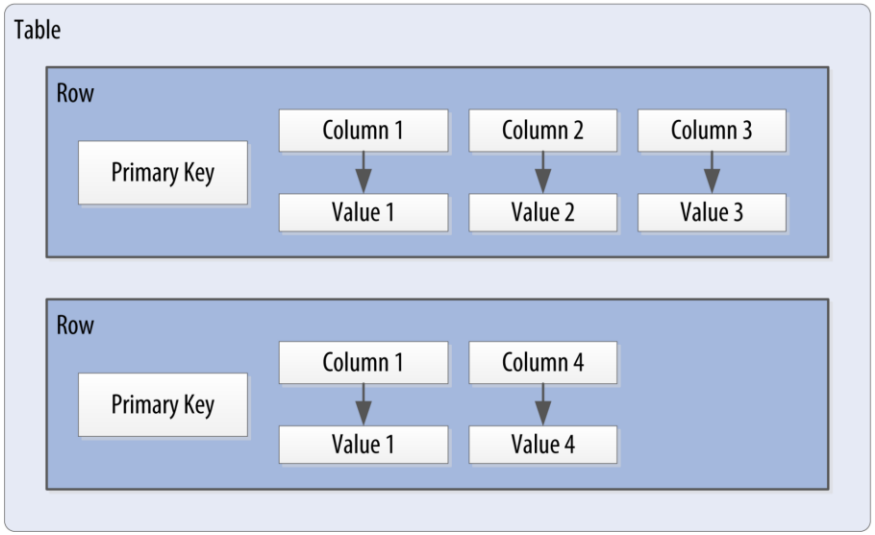
High performance

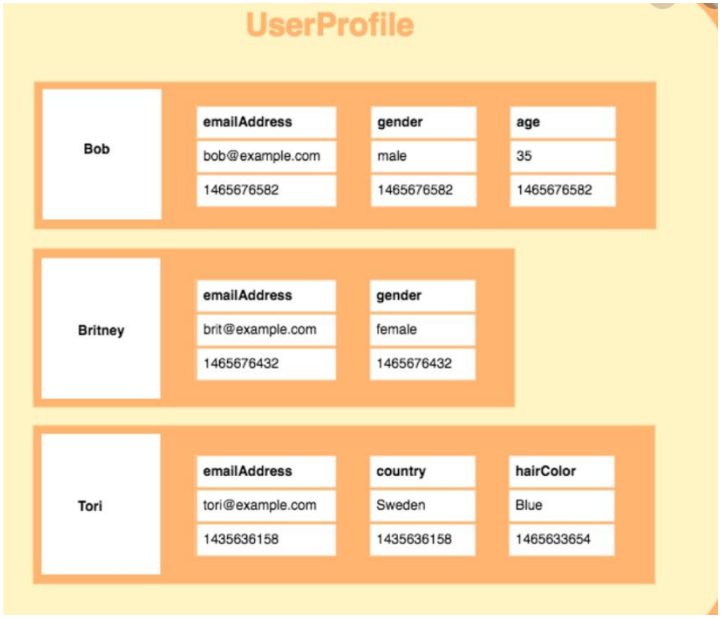
MySQL Comparision:

Statistics based on 50 GB Data

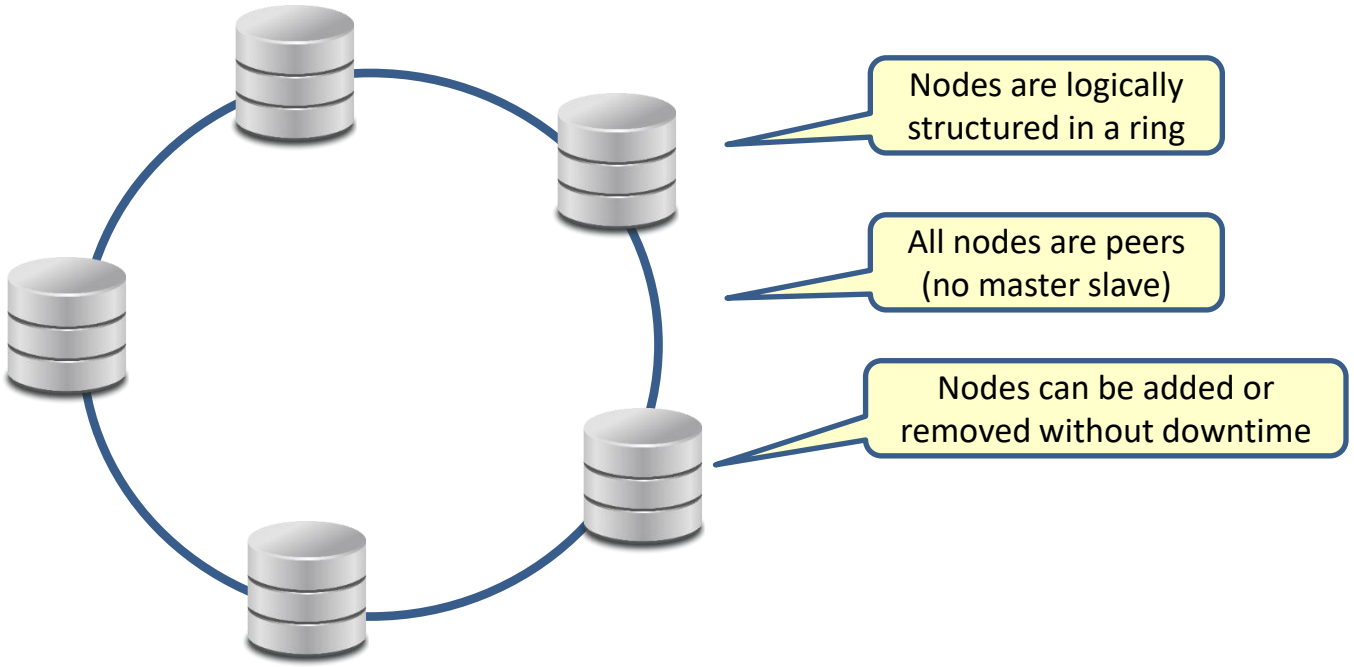
	Cassandra	MySQL
Average Write	**0.12 ms**	**~300 ms**
Average Read	**15 ms**	**~350 ms**

Cassandra data model





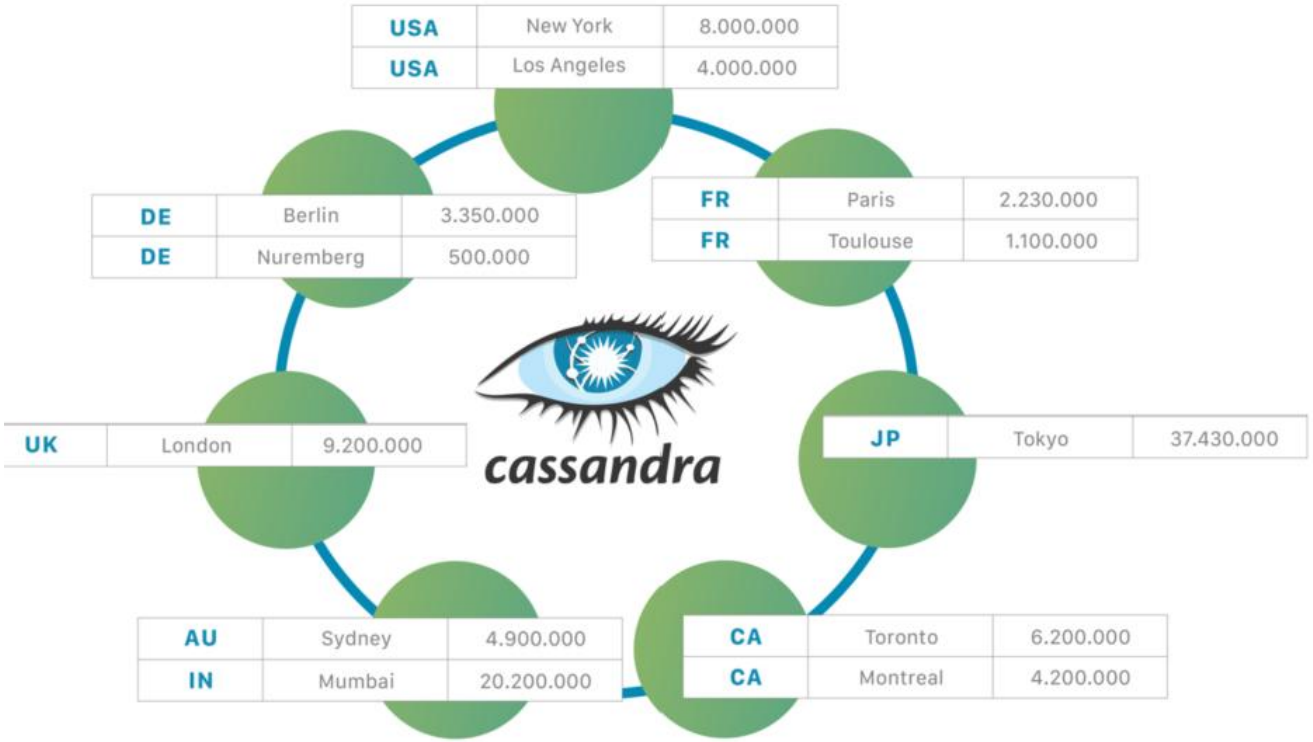
Cassandra cluster of nodes



Cassandra partitioning

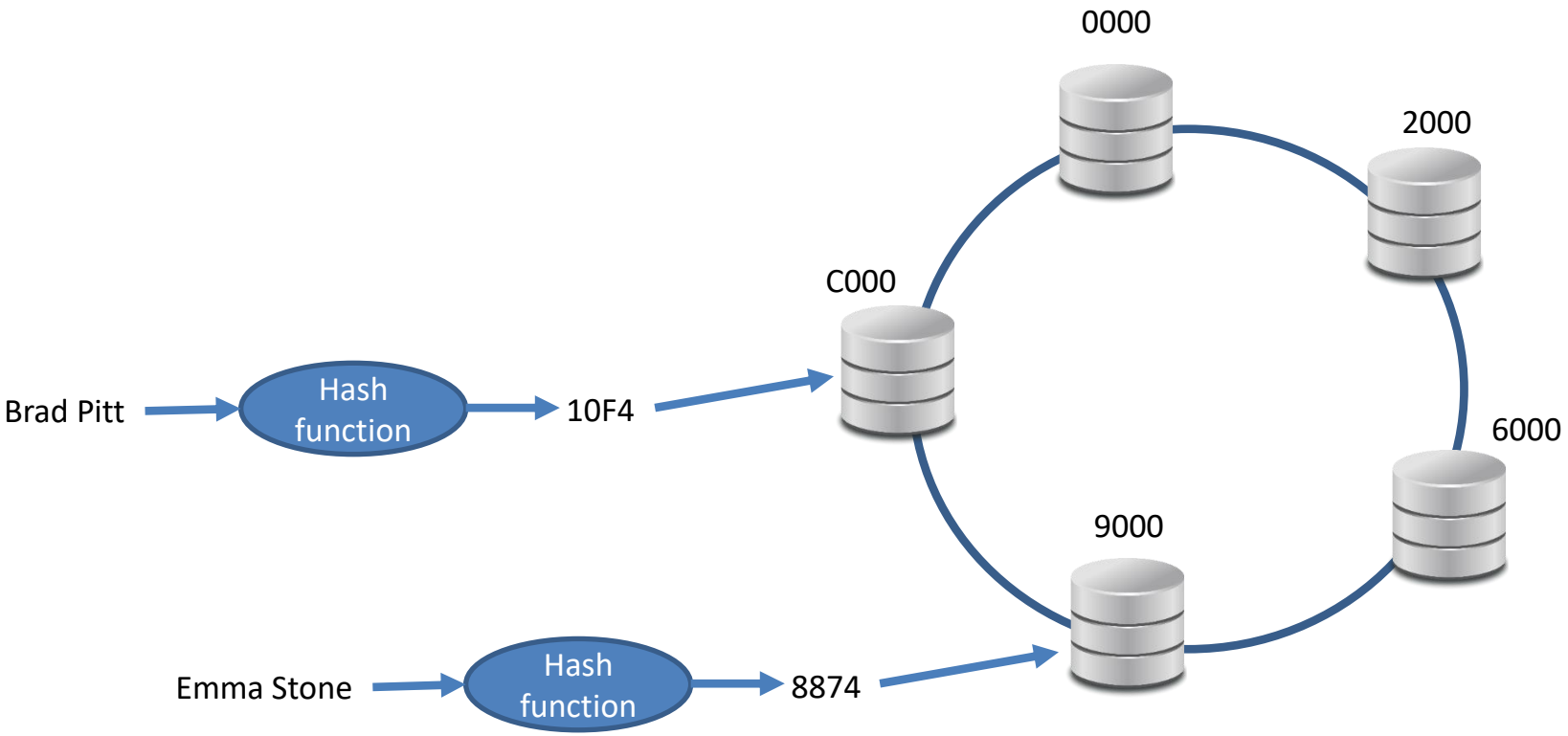
COUNTRY	CITY	POPULATION
USA	New York	8.000.000
USA	Los Angeles	4.000.000
FR	Paris	2.230.000
DE	Berlin	3.350.000
UK	London	9.200.000
AU	Sydney	4.900.000
DE	Nuremberg	500.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
FR	Toulouse	1.100.000
JP	Tokyo	37.430.000
IN	Mumbai	20.200.000

Partition Key

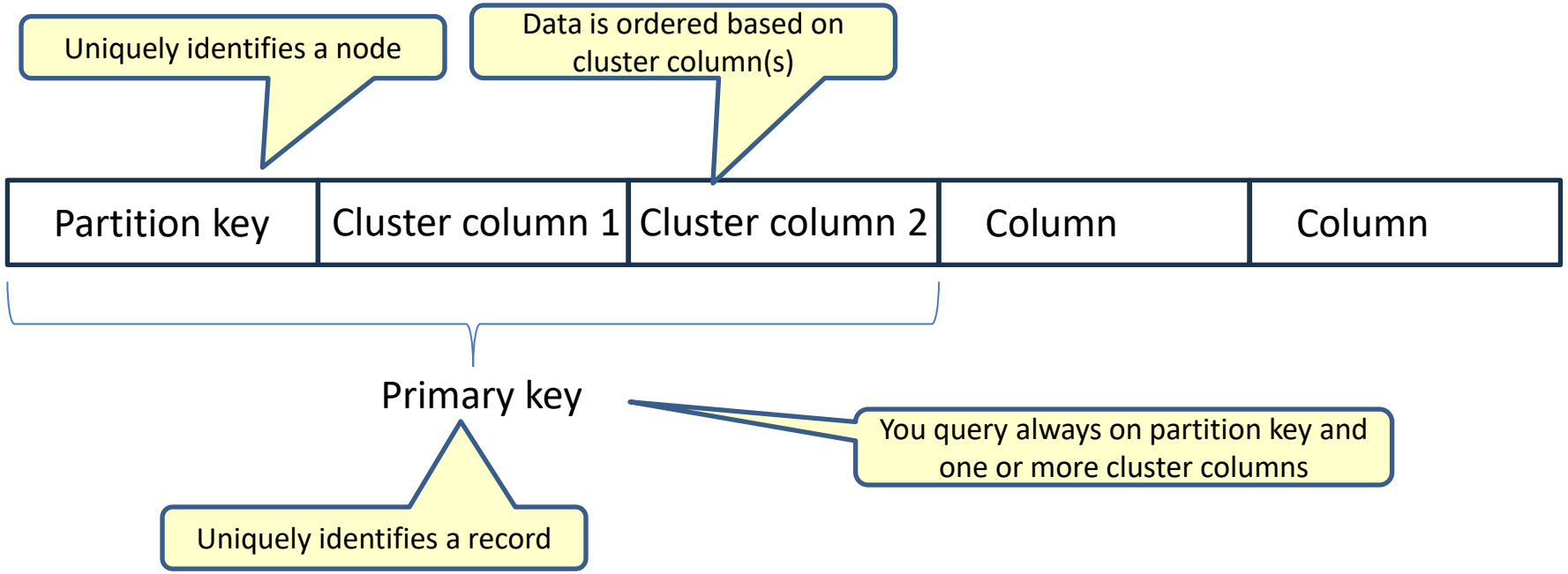


Partition key

- Decides where to place this record



Primary key




```
CREATE TABLE movies_by_actor (
  first_name TEXT,
  last_name TEXT,
  title TEXT,
  year INT,
  PRIMARY KEY ((first_name, last_name), title, year)
); ↩
```

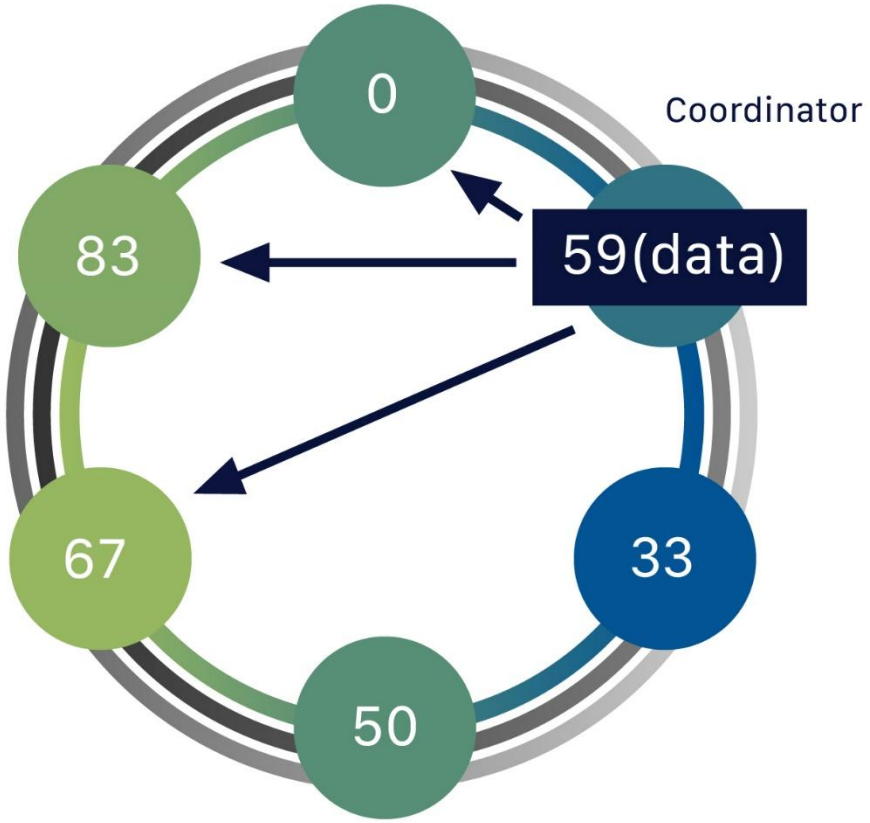
partition key(s)

clustering key(s)

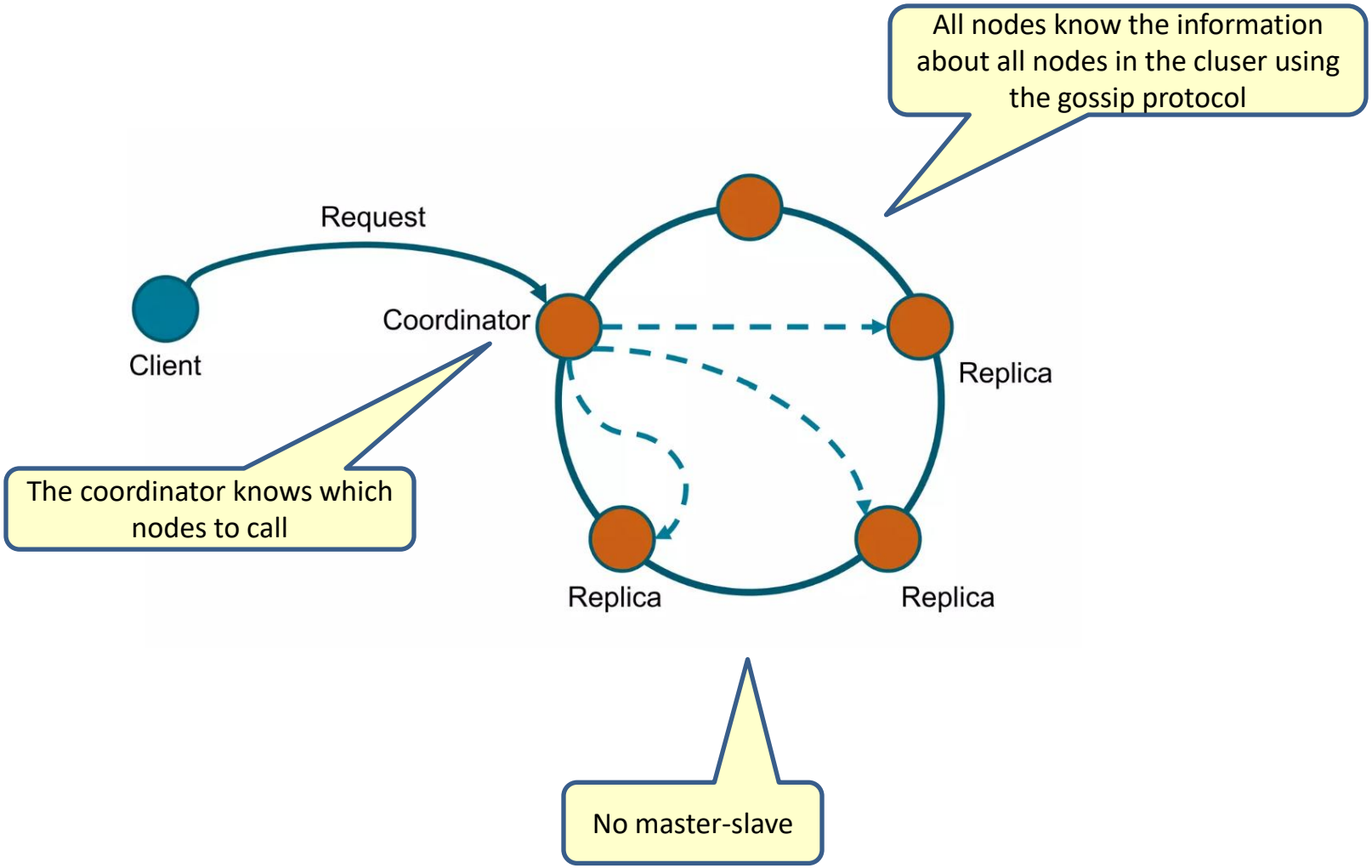
first_name	last_name	title	year
Johnny	Depp	Alice in Wonderland	2010
Johnny	Depp	Edward Scissorhands	1990
Anne	Hathaway	Alice in Wonderland	2010

Cassandra replication

RF = 3

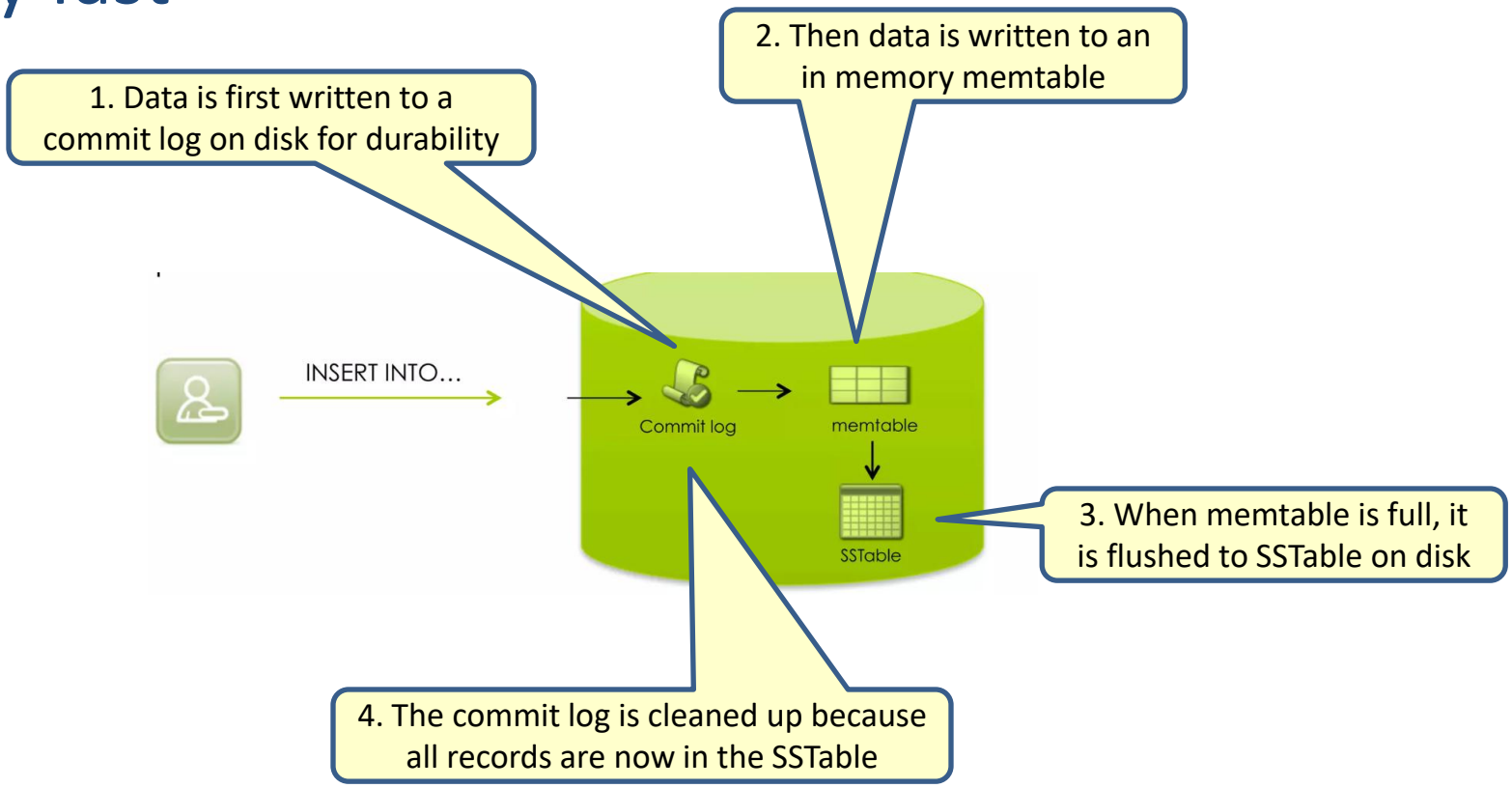


You can read and write to any node



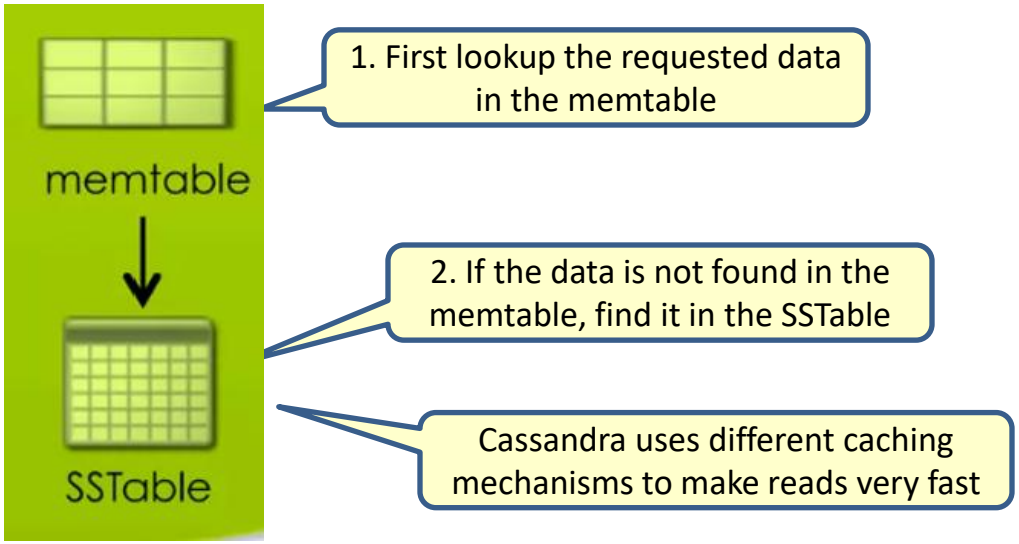
Writes in Cassandra

- Very fast



Reads in Cassandra

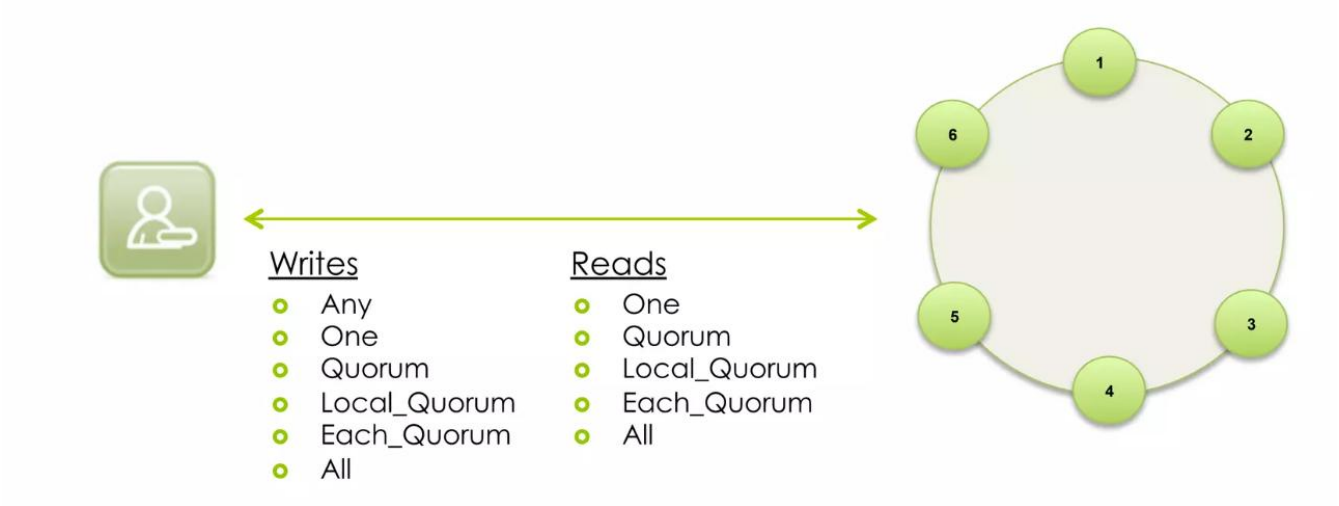
- Very fast



Tunable consistency

- Choose the balance between **data consistency** and **availability** for each read and write operation by setting a consistency level.
- **Write operations:** The client determines how many replicas must acknowledge the write before the operation is reported as successful.
- **Read operations:** The client specifies how many replicas must respond to a read request before data is returned to the client.
- **Key consistency levels**
  - **ONE**: The operation is successful if only one replica responds. This is the fastest but offers the lowest consistency.
  - **QUORUM**: A majority of replicas must respond. For a replication factor of 3, this means 2 out of 3 replicas must acknowledge the operation. This provides a balance between consistency and performance.
  - **ALL**: All replicas must respond. This provides the strongest consistency but is the slowest option.

Cassandra tunable consistency

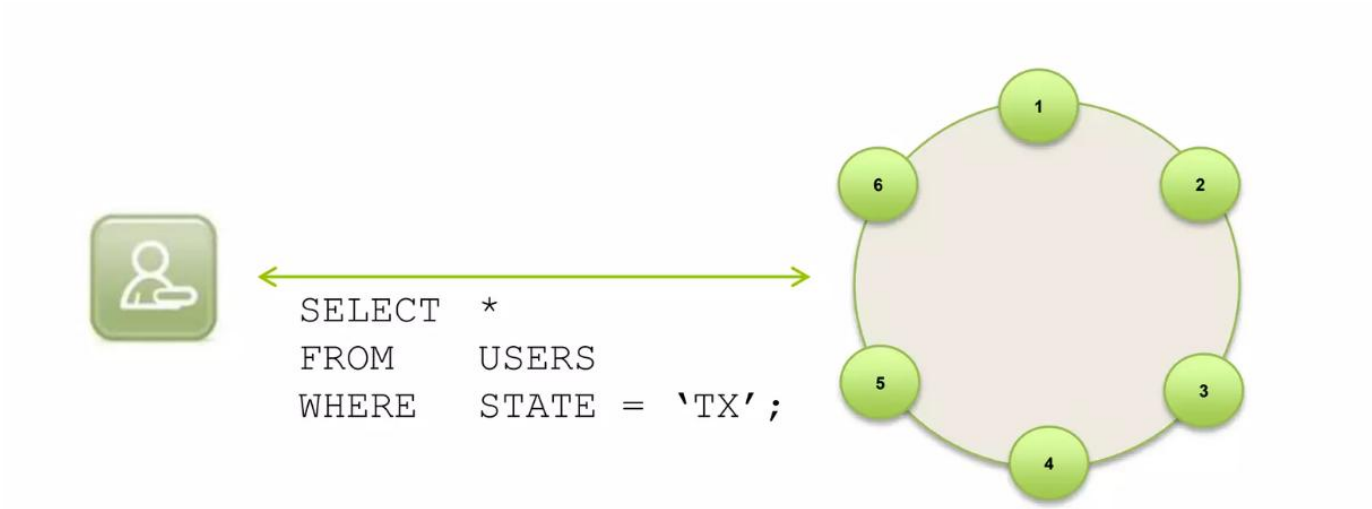


CQL Examples

```
SELECT total_purchases FROM SALES
USING  CONSISTENCY QUORUM
WHERE  customer_id = 5

UPDATE SALES
USING  CONSISTENCY ONE
SET    total_purchases = 500000
WHERE  customer_id = 4
```

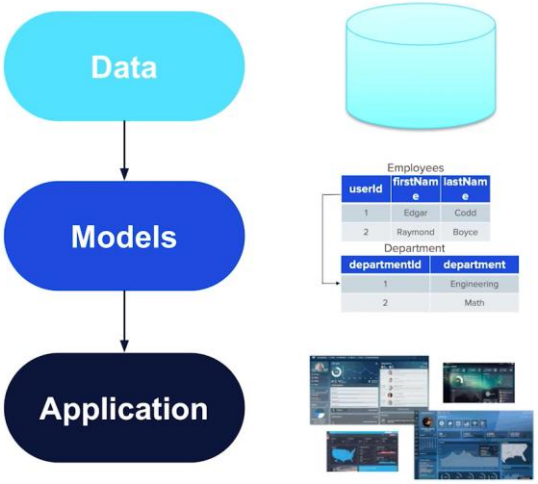
Cassandra Query Language (CQL)

- Very similar to SQL

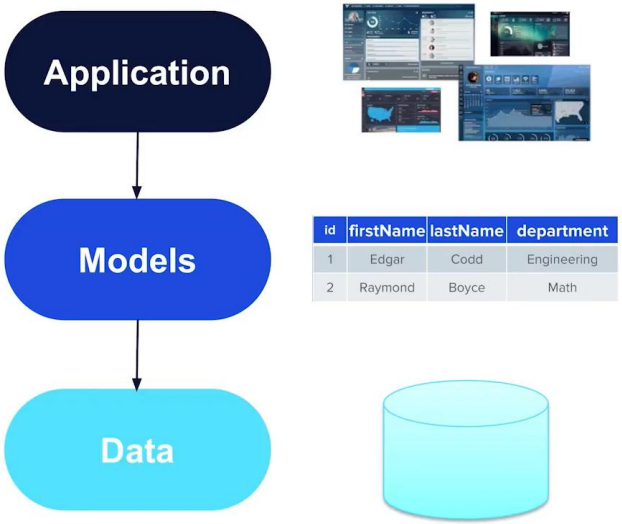


Data modeling

- RDBMS



- Cassandra



Ratings by movie

title	year	email	rating
Alice in Wonderland	2010	jen@datastax.com	10
Alice in Wonderland	2010	joe@datastax.com	9
Alice in Wonderland	1951	jen@datastax.com	8
Edward Scissorhands	1990	joe@datastax.com	10

```
CREATE TABLE ratings_by_movie (
  title TEXT,
  year INT,
  email TEXT,
  rating INT,
  PRIMARY KEY ((title, year), email)
); ↩
```

```
INSERT INTO ratings_by_movie (title, year, email,
rating)
VALUES ('Alice in Wonderland', 2010,
'jen@datastax.com', 10);
INSERT INTO ratings_by_movie (title, year, email,
rating)
VALUES ('Alice in Wonderland', 2010,
'joe@datastax.com', 9);
INSERT INTO ratings_by_movie (title, year, email,
rating)
VALUES ('Alice in Wonderland', 1951,
'jen@datastax.com', 8);
INSERT INTO ratings_by_movie (title, year, email,
rating)
VALUES ('Edward Scissorhands', 1990,
'joe@datastax.com', 10); ↩
```

Actors by movie

title	year	first_name	last_name
Alice in Wonderland	2010	Anne	Hathaway
Alice in Wonderland	2010	Johnny	Depp
Edward Scissorhands	1990	Johnny	Depp

```
CREATE TABLE actors_by_movie (
  title TEXT,
  year INT,
  first_name TEXT,
  last_name TEXT,
  PRIMARY KEY ((title, year), first_name, last_name)
); ↩
```

Movies by actor

first_name	last_name	title	year
Johnny	Depp	Alice in Wonderland	2010
Johnny	Depp	Edward Scissorhands	1990
Anne	Hathaway	Alice in Wonderland	2010

```
CREATE TABLE movies_by_actor (
  first_name TEXT,
  last_name TEXT,
  title TEXT,
  year INT,
  PRIMARY KEY ((first_name, last_name), title, year)
); ↩
```

Movies by user

email	watched_on	title	year
joe@datastax.com	2020-04-28	Edward Scissorhands	1990
joe@datastax.com	2020-03-08	Alice in Wonderland	2010
joe@datastax.com	2020-02-13	Toy Story 3	2010
joe@datastax.com	2020-01-22	Despicable Me	2010
joe@datastax.com	2019-12-30	Alice in Wonderland	1951
jen@datastax.com	2011-10-01	Alice in Wonderland	2010

```
CREATE TABLE movies_by_user (
  email TEXT,
  title TEXT,
  year INT,
  watched_on DATE,
  PRIMARY KEY ((email), watched_on, title, year)
) WITH CLUSTERING ORDER BY (watched_on DESC); ↩
```

Data duplication

email	title	year	rating
joe@datastax.com	Alice in Wonderland	2010	9
joe@datastax.com	Edward Scissorhands	1990	10
jen@datastax.com	Alice in Wonderland	1951	8
jen@datastax.com	Alice in Wonderland	2010	10

title	year	email	rating
Alice in Wonderland	2010	jen@datastax.com	10
Alice in Wonderland	2010	joe@datastax.com	9
Alice in Wonderland	1951	jen@datastax.com	8
Edward Scissorhands	1990	joe@datastax.com	10

title	year	first_name	last_name
Alice in Wonderland	2010	Anne	Hathaway
Alice in Wonderland	2010	Johnny	Depp
Edward Scissorhands	1990	Johnny	Depp

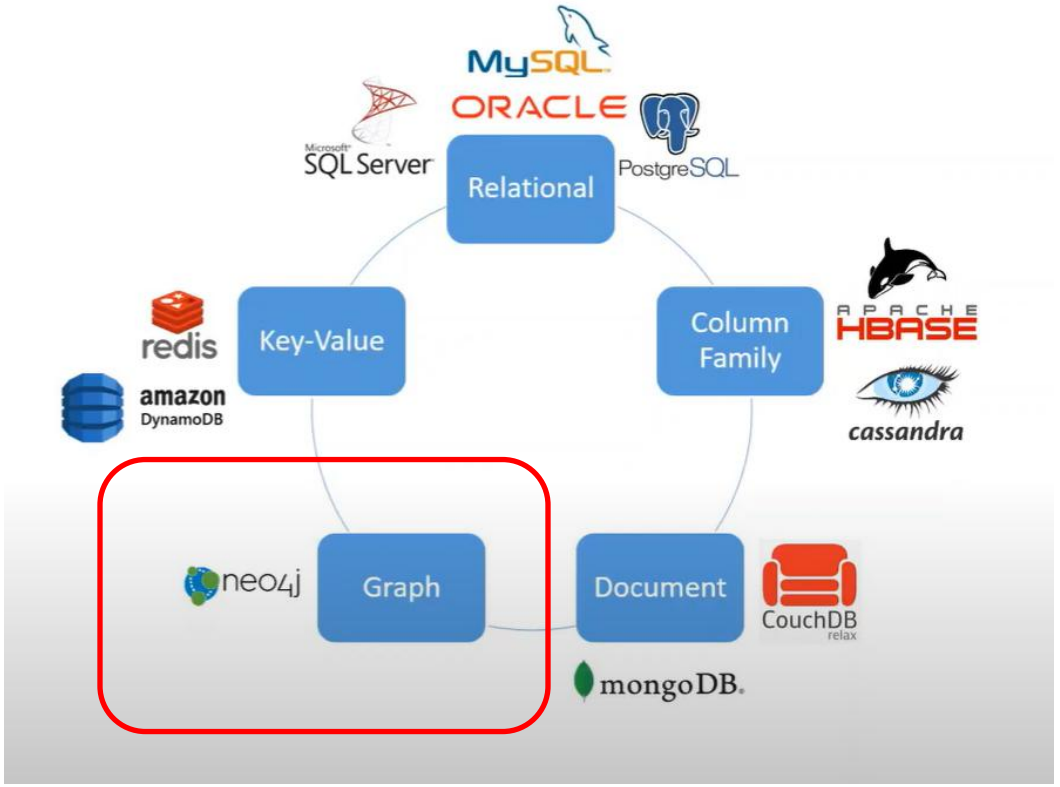
email	watched_on	title	year
joe@datastax.com	2020-04-28	Edward Scissorhands	1990
joe@datastax.com	2020-03-08	Alice in Wonderland	2010
joe@datastax.com	2020-02-13	Toy Story 3	2010
joe@datastax.com	2020-01-22	Despicable Me	2010
joe@datastax.com	2019-12-30	Alice in Wonderland	1951
jen@datastax.com	2011-10-01	Alice in Wonderland	2010

Cassandra advantages & disadvantages

- Advantages
  - Very fast for writes and reads
  - Linear scalability
  - You can write to any node to geographically distributed clusters
  - Tunable consistency
- Disadvantages
  - No joins and aggregates
  - No transactions
    - No strict consistency
  - Data duplication
  - Slow updates and deletes
  - Only structured data

Cassandra use cases

- Real-time analytics pipeline
  - Large data sets, only write and read
- IoT data management
  - Large data sets, only write and read
- Content management systems
  - High write volumes, ensure data availability, and maintain consistency
- Real-time analysis of large datasets
  - Fraud detection and risk management
- Time-series data analysis
  - Large data sets, only write and read



GRAPH DATABASE

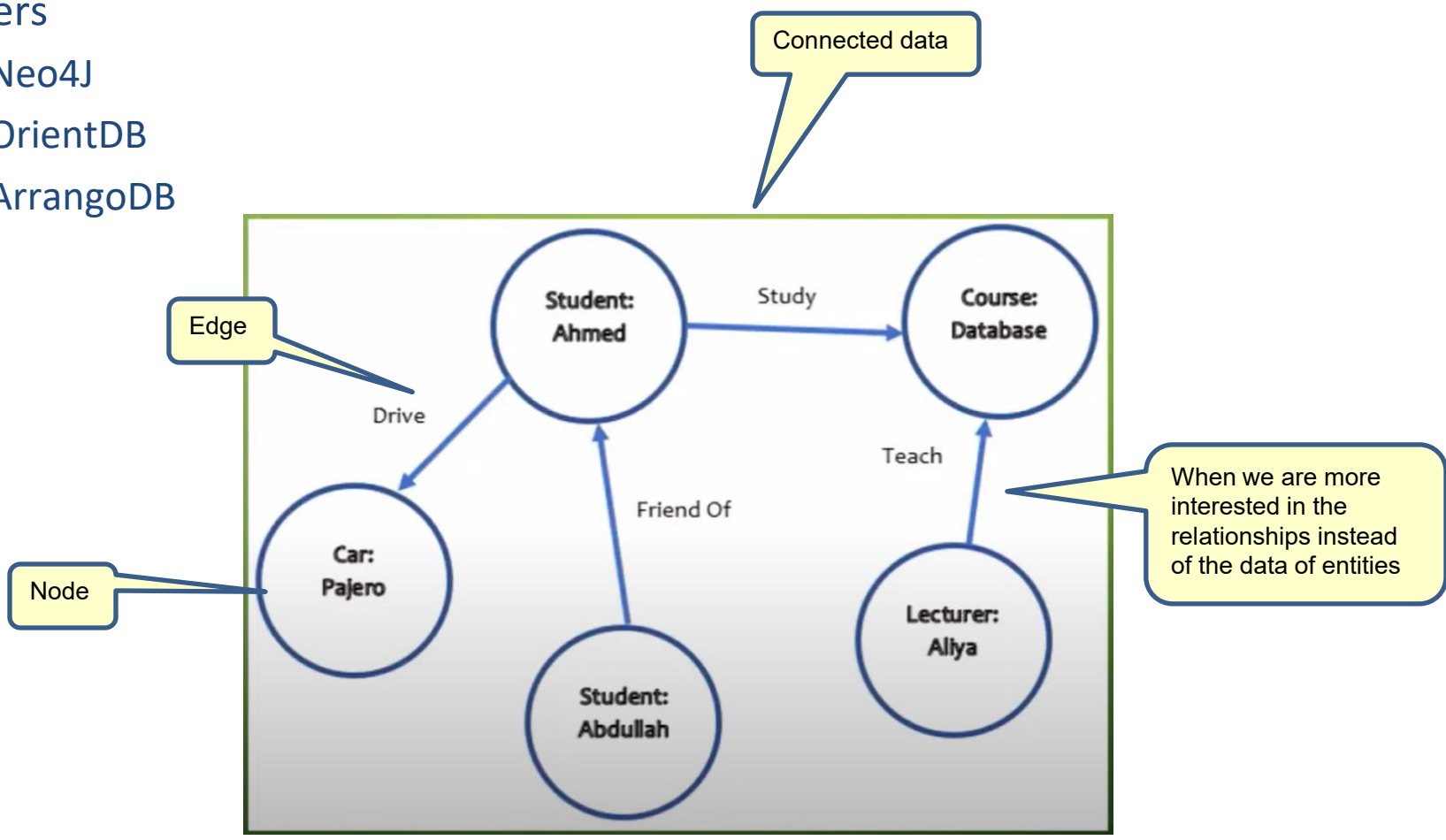
NEO4J

Graph database

- Store data in nodes and relationships
  - No Joins
- Very fast in analyzing data with lots of relationships
- No fixed schema
  - Easy to evolve your dataset

Graph database

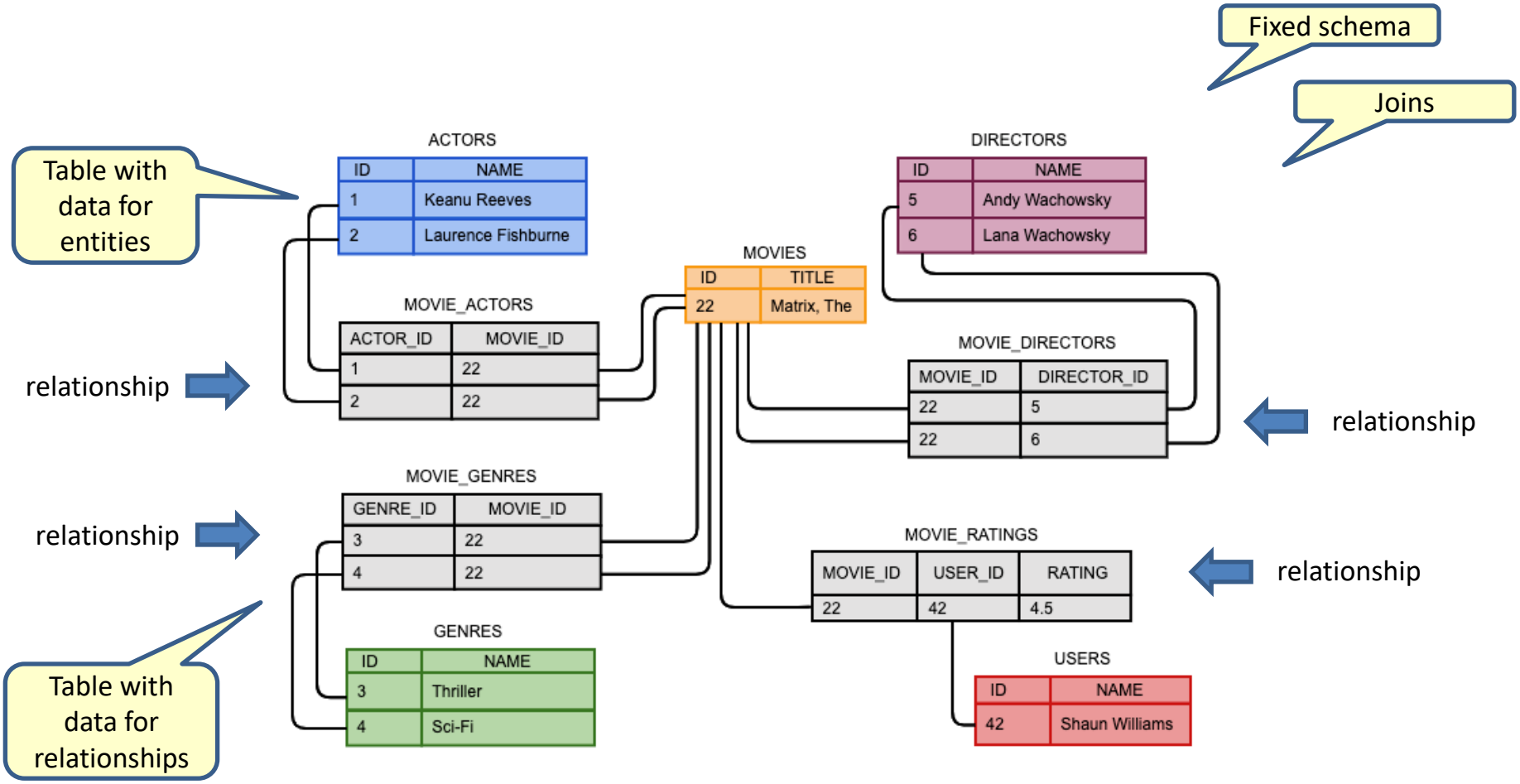
- Players
  - Neo4J
  - OrientDB
  - ArrangoDB



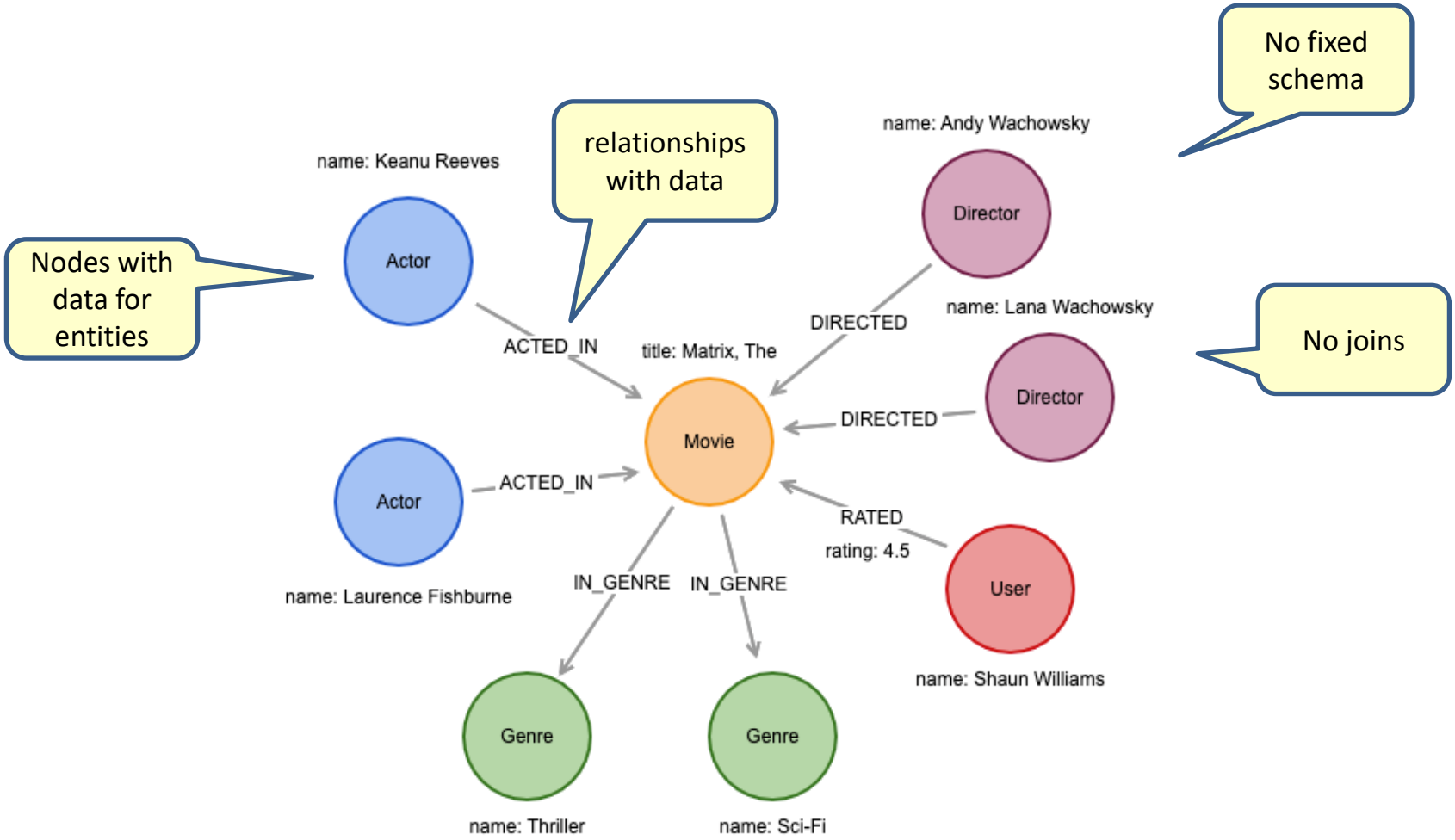
Graph DB use cases

- Social networks
- Knowledge graphs
- Fraud detection
- Contact tracing
- Real time recommendation
- Graph based search
- Network & IT operations
- Identity and access management

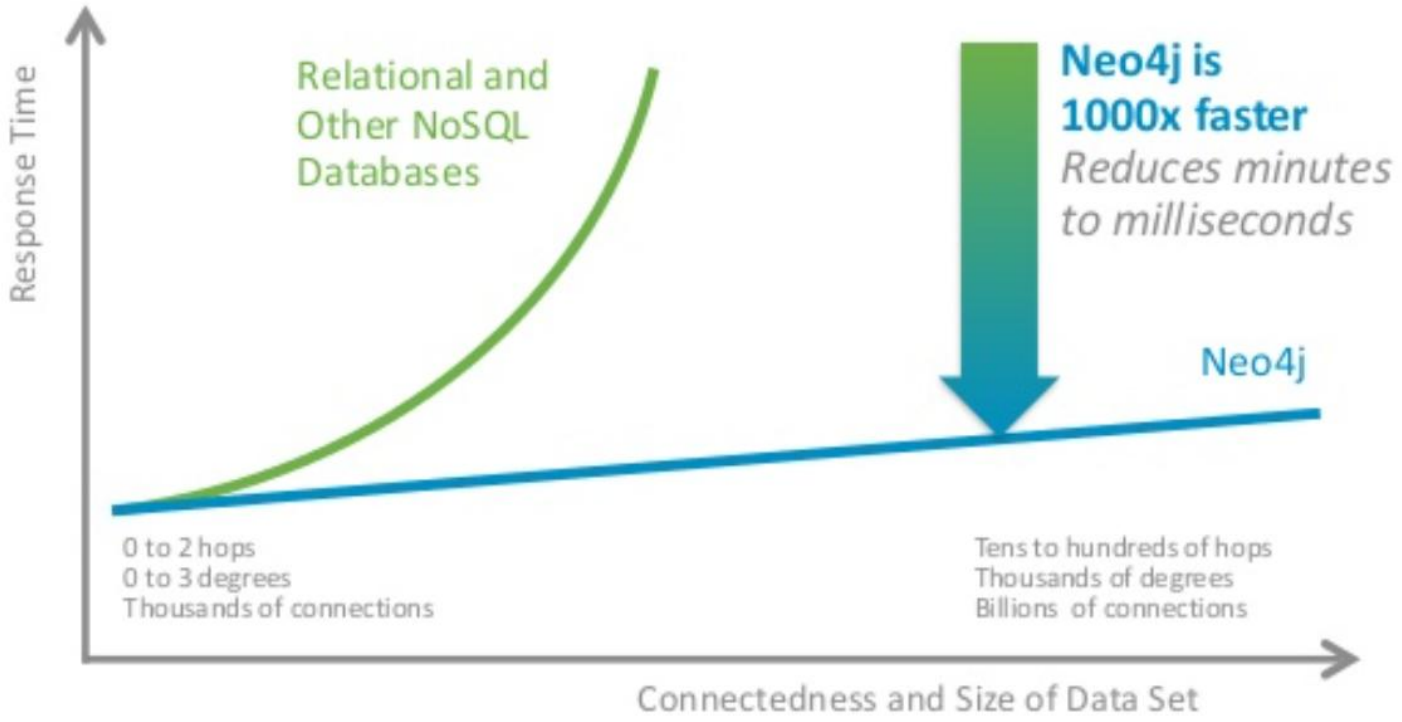
Relational database



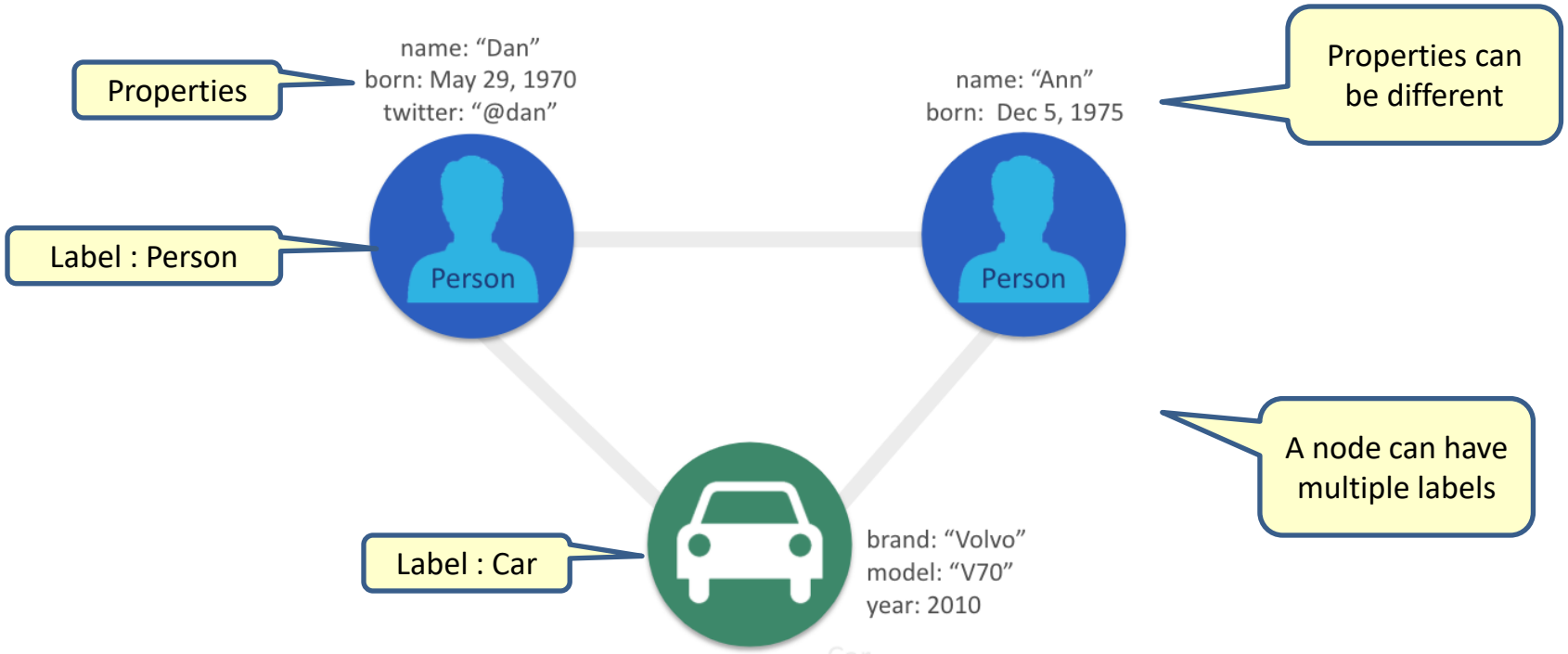
Graph database: NEO4J



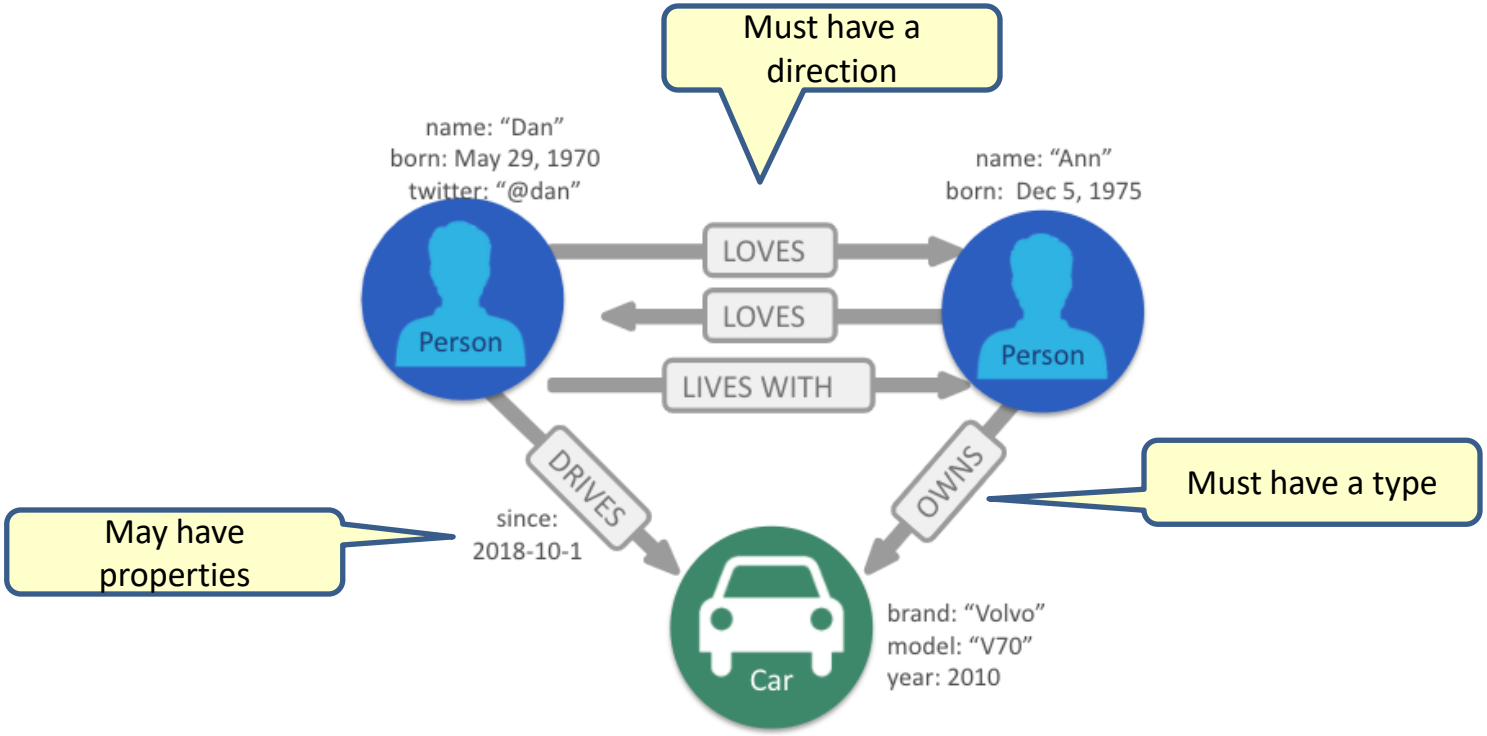
Query performance



Nodes

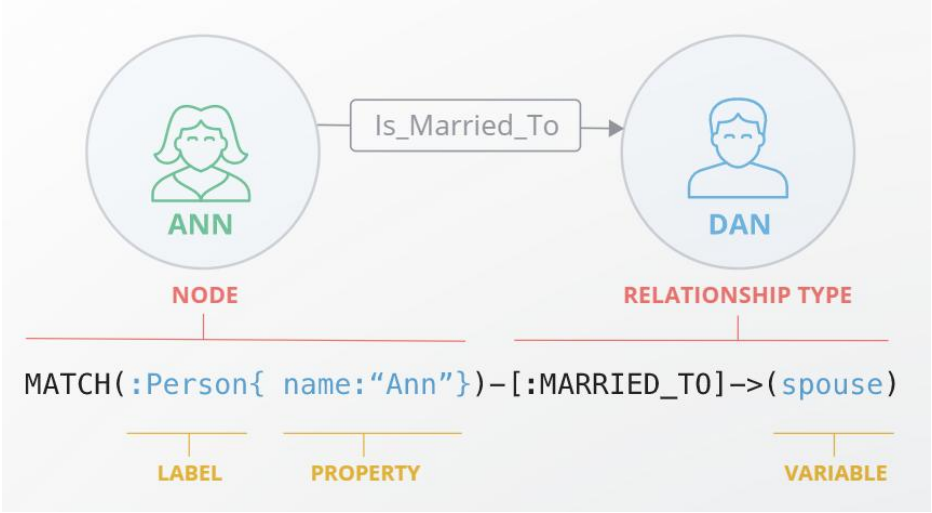


Relationships



Cypher query language

- Declarative
  - Describe what you want
  - Not how to do it



Create a node

Label

Property

```
CREATE (:Movie {title: 'Batman Begins'})
```

```
neo4j$ MATCH (n) RETURN n
```

Graph

Table

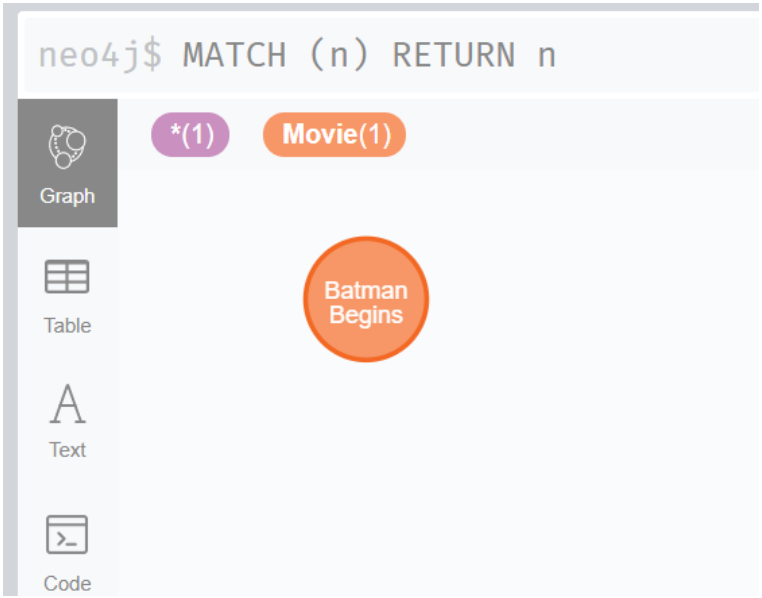
Text

Code

n

1

```
{
  "identity": 0,
  "labels": [
    "Movie"
  ],
  "properties": {
"title": "Batman Begins"
  }
}
```



Create a relationship

```
MATCH (a:Person), (m:Movie)
WHERE a.name = 'Michael Caine' AND m.title = 'Batman Begins'
CREATE (a)-[:ACTED_IN]->(m)
```


```
MATCH (a:Person), (m:Movie)
WHERE a.name = 'Christian Bale' AND m.title = 'Batman Begins'
CREATE (a)-[:ACTED_IN {roles: ['Bruce Wayne', 'Batman']}]->(m)
RETURN a, m
```

Queries on nodes

```
MATCH (n)
RETURN n
```

Retrieve all nodes

```
MATCH (p:Person)
RETURN p
```

Retrieve all Person nodes

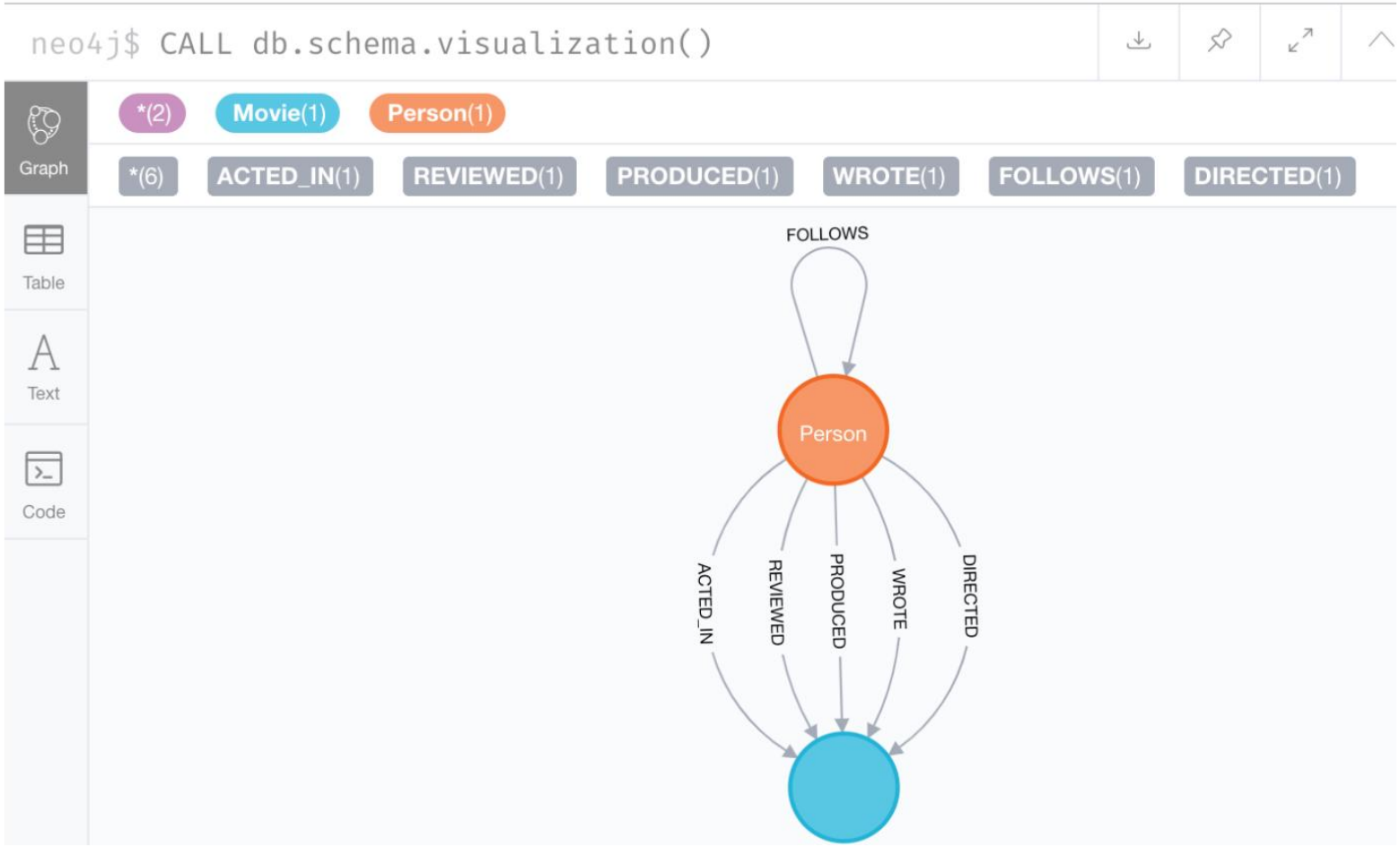
```
MATCH (p:Person {born: 1970})
RETURN p
```

Retrieve all Person nodes born in 1970

```
MATCH (m:Movie {released: 2003, tagline: 'Free your mind'})
RETURN m
```

Retrieve all Movie nodes released in 2003 with the tagline ‘free your mind’

Visualize the schema



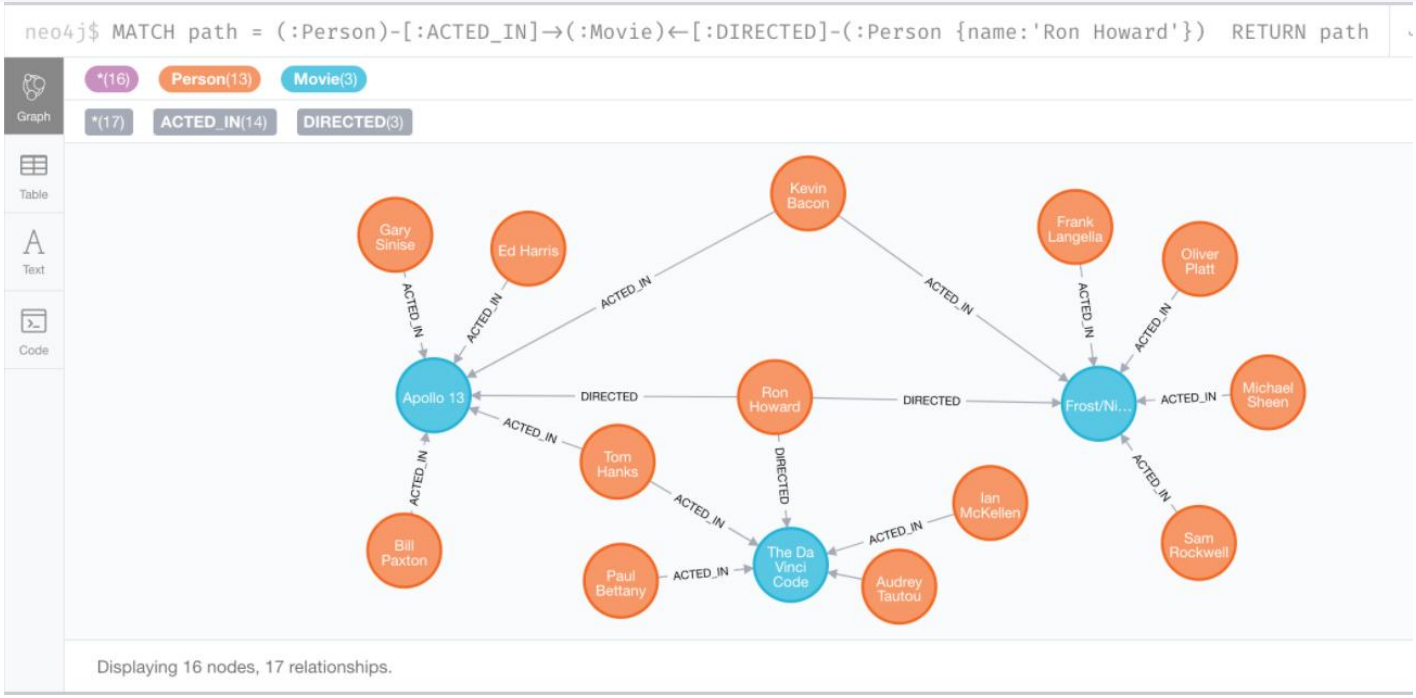
Queries on relationships

```
MATCH (p:Person)-[rel:ACTED_IN]->(m:Movie {title: 'The Matrix'})
RETURN p, rel, m
```

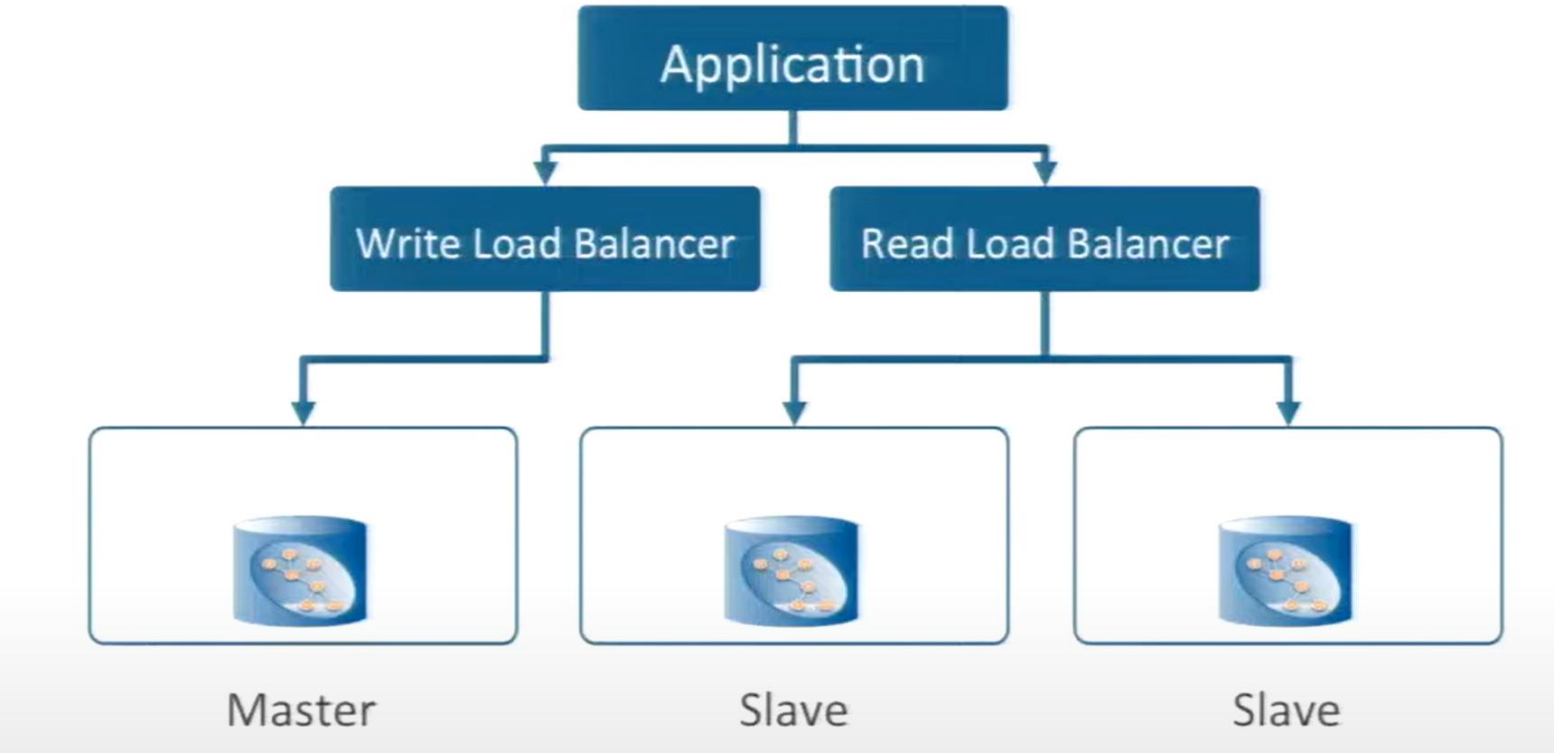
Give all actors that played in 'The Matrix'

Return multiple paths

```
MATCH  path = (:Person)-[:ACTED_IN]->(:Movie)<-[:DIRECTED]-(:Person {name:'Ron Howard'})
RETURN  path
```



Neo4j High Availability cluster



Neo4j advantages & disadvantages

- Advantages
  - Very fast in querying deeply connected data
  - Database model and domain model are similar
  - Powerful and simple query language
  - Easy to add new nodes, attributes and relationships
  - Cluster (scalability, failover and high availability)
- Disadvantages
  - Not for unconnected data
  - Not for unstructured data

SPRING BOOT NEO4J EXAMPLE

Domain classes + repository

```
@Node
public class Person {

  @Id @GeneratedValue private Long id;

  private String name;

  @Relationship(type = "TEAMMATE")
  public Set<Person> teammates= new HashSet<>();
```

```
public interface PersonRepository extends Neo4jRepository<Person, Long> {

  Person findByName(String name);
  List<Person> findByTeammatesName(String name);
}
```

Application(1/2)

```
@SpringBootApplication
@EnableNeo4jRepositories
public class PersonApplication implements CommandLineRunner{

  @Autowired
  PersonRepository personRepository;

  public static void main(String[] args) {
        SpringApplication.run(PersonApplication.class, args);
  }
```

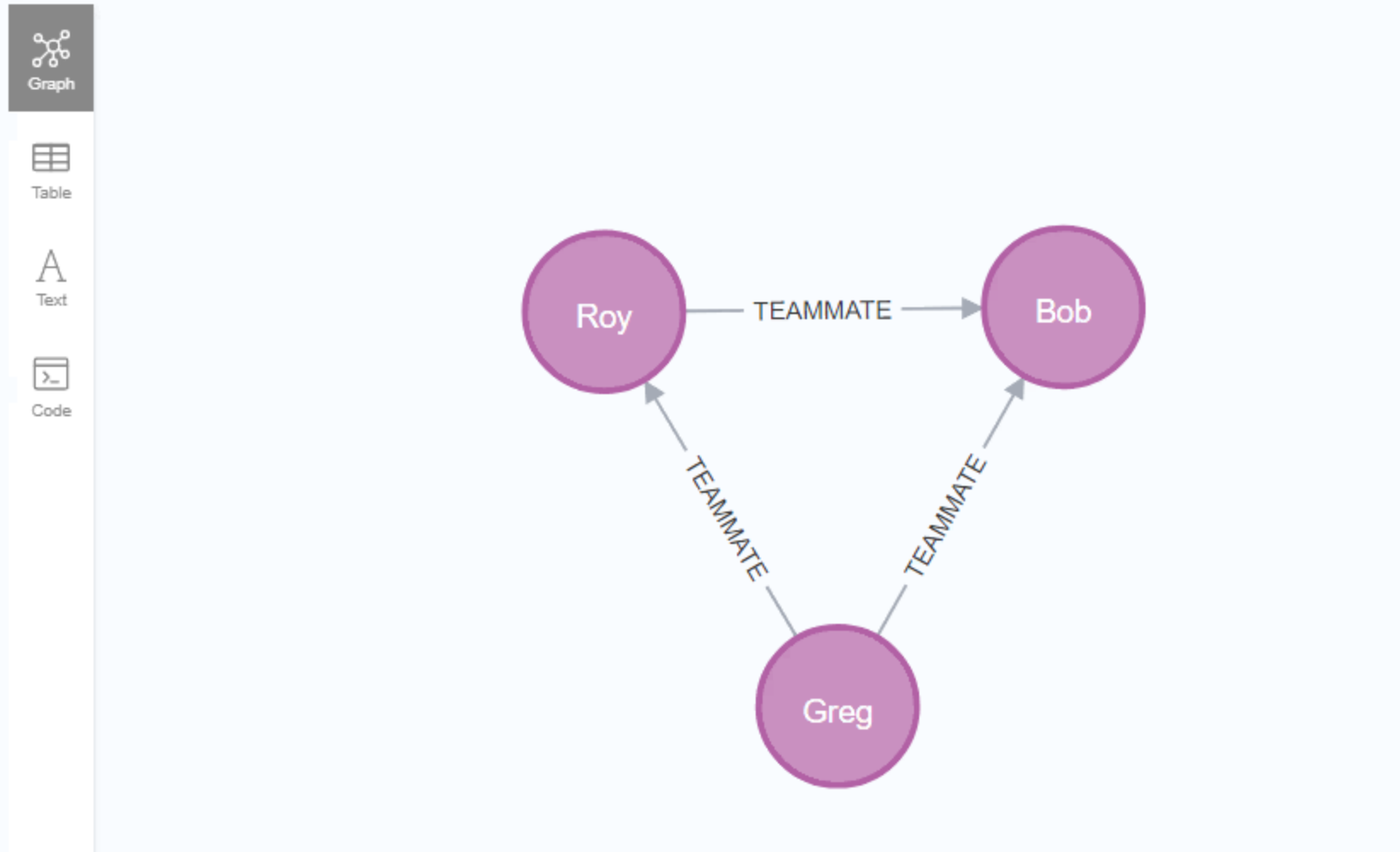
Application(2/2)

```
@Override
 public void run(String... args) throws Exception {
                personRepository.deleteAll();

                Person greg = new Person("Greg");
                Person roy = new Person("Roy");
                Person bob = new Person("Bob");
                greg.worksWith(roy);
                greg.worksWith(bob);
                roy.worksWith(bob);

                List<Person> team = Arrays.asList(greg, roy, bob);

                personRepository.save(greg);
                personRepository.save(roy);
                personRepository.save(bob);

                System.out.println("Lookup each person by name...");
                team.stream().forEach(person -> System.out.println(
                                personRepository.findByName(person.getName())));

                List<Person> teammates = personRepository.findByTeammatesName(bob.getName());
                System.out.println("The following have Bob as a teammate...");
                teammates.stream().forEach(person -> System.out.println( person.getName()));
 }
}
```

application.properties

```
spring.neo4j.uri=bolt://localhost:7687
spring.data.neo4j.username=neo4j
spring.data.neo4j.password=admin
```

Nodes


Graph
Table
Text
Code
Roy
TEAMMATE
Bob
TEAMMATE
TEAMMATE
Greg

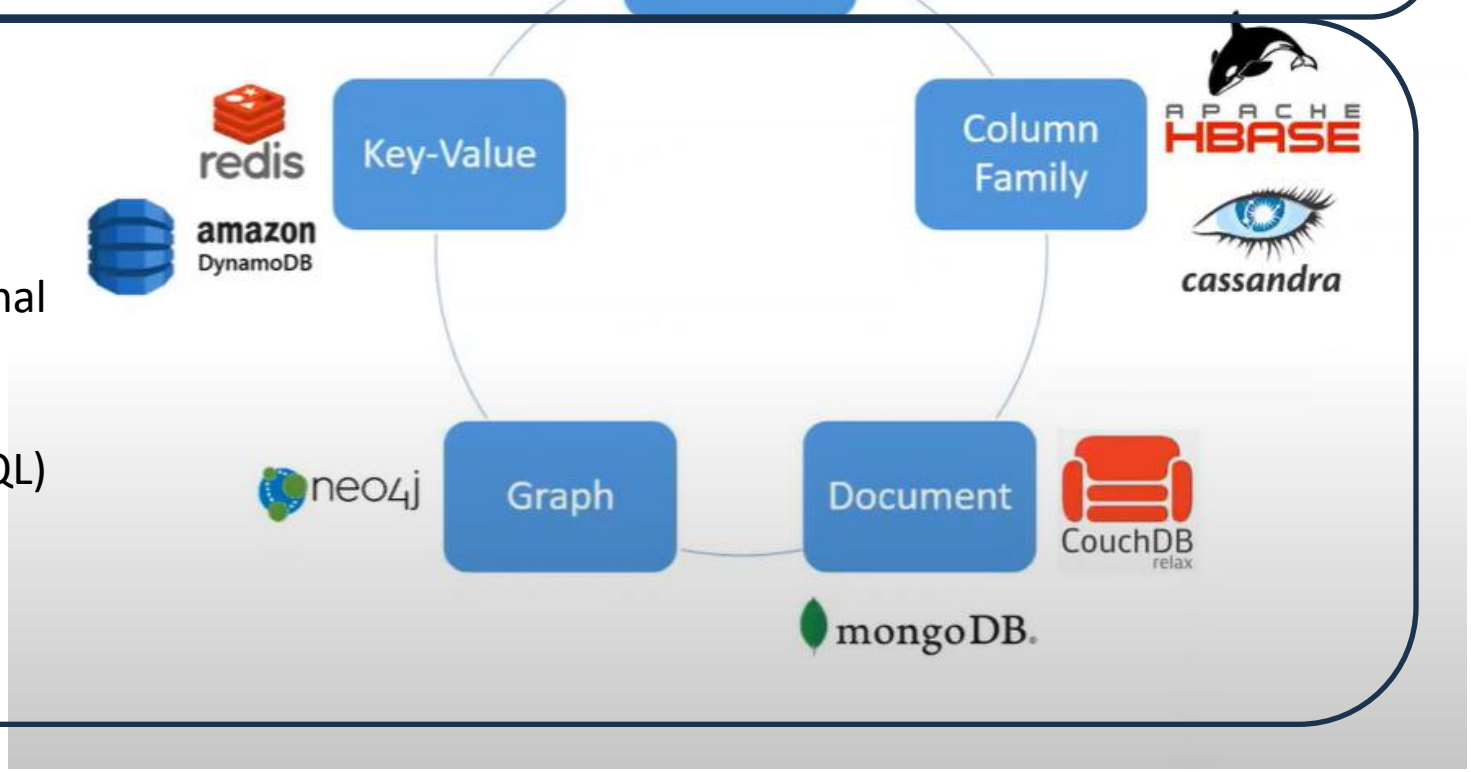
SUMMARY

Five different types of databases

Relational

Non-relational

NoSQL
(Not only SQL)



Relational database

- Does not scale well
- Hard to change
- Does not handle unstructured and semi-structured data

NoSQL databases

- Scales very well
- Easy to change, no fixed schema
- Handles unstructured and semi structured data

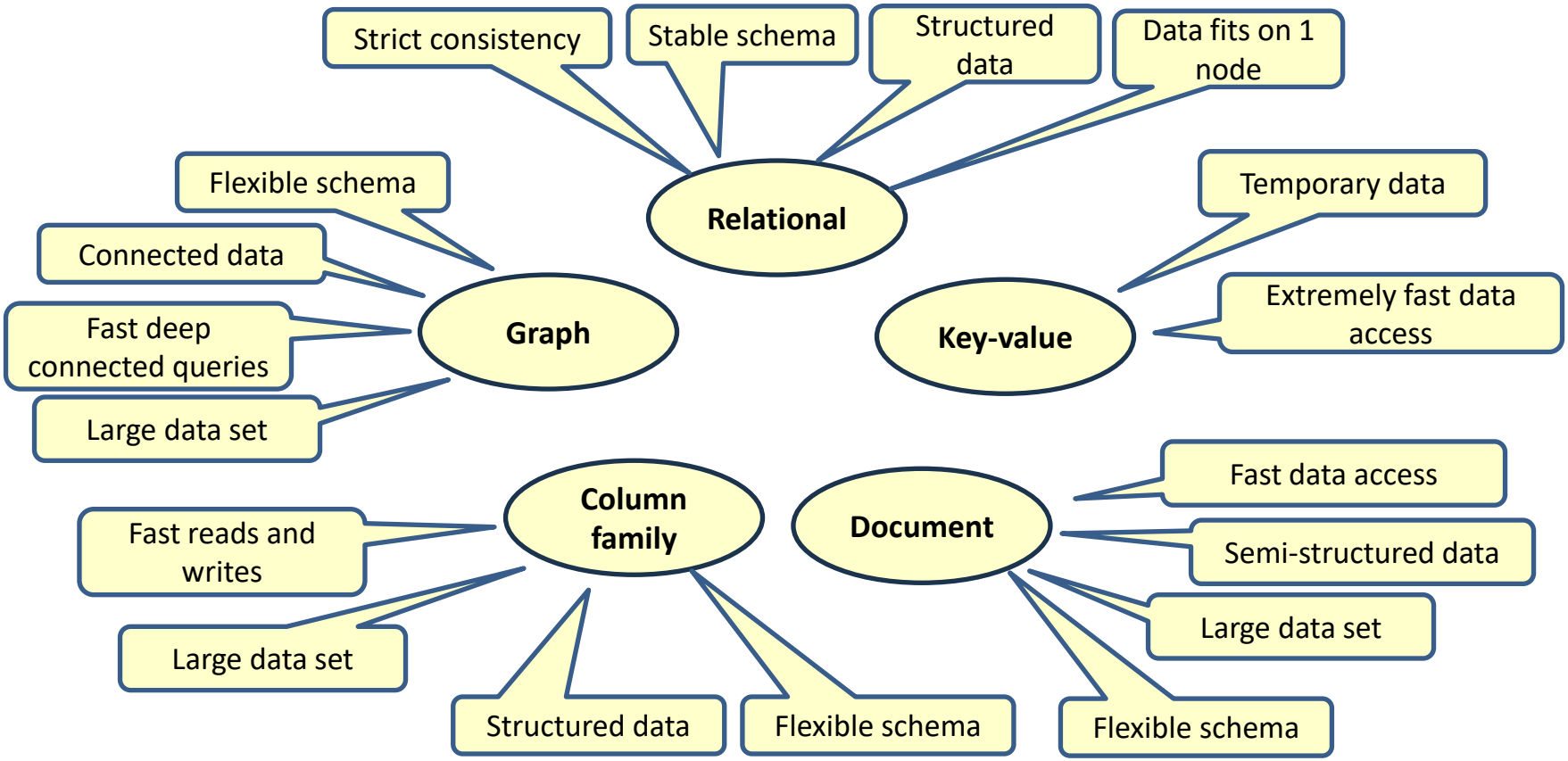
Comparing databases

	Strong points	Weak points
Relational	Structured data Wide adaption Strict consistency	Hard to scale Hard to change the schema Not good in unstructured data
Key-value	Extremely fast in-memory data	Not for permanent data storage No or limited query functionality
Document	Whole documents Fast reads and writes Unstructured data	Data duplication
Wide column	Very fast reads and writes Fast queries at scale Linear scalable	Data duplication Slow updates and deletes Not good in unstructured data
Graph	Connected data Fast queries at scale Flexible data structure	Not good for unconnected data Not good in unstructured data

Transactions

- Oracle:
  - One node: ACID
  - HA cluster with sharding: ACID
- Mongo, Neo4J:
  - One node: ACID
  - HA cluster with sharding: BASE
- Cassandra:
  - One node: Tunable consistency
  - HA cluster with sharding: Tunable consistency

Which database to use?



Key principle 6

- When your data does not fit on one node you automatically will get
  - Data duplication
- Data duplication over multiple nodes means eventual consistency