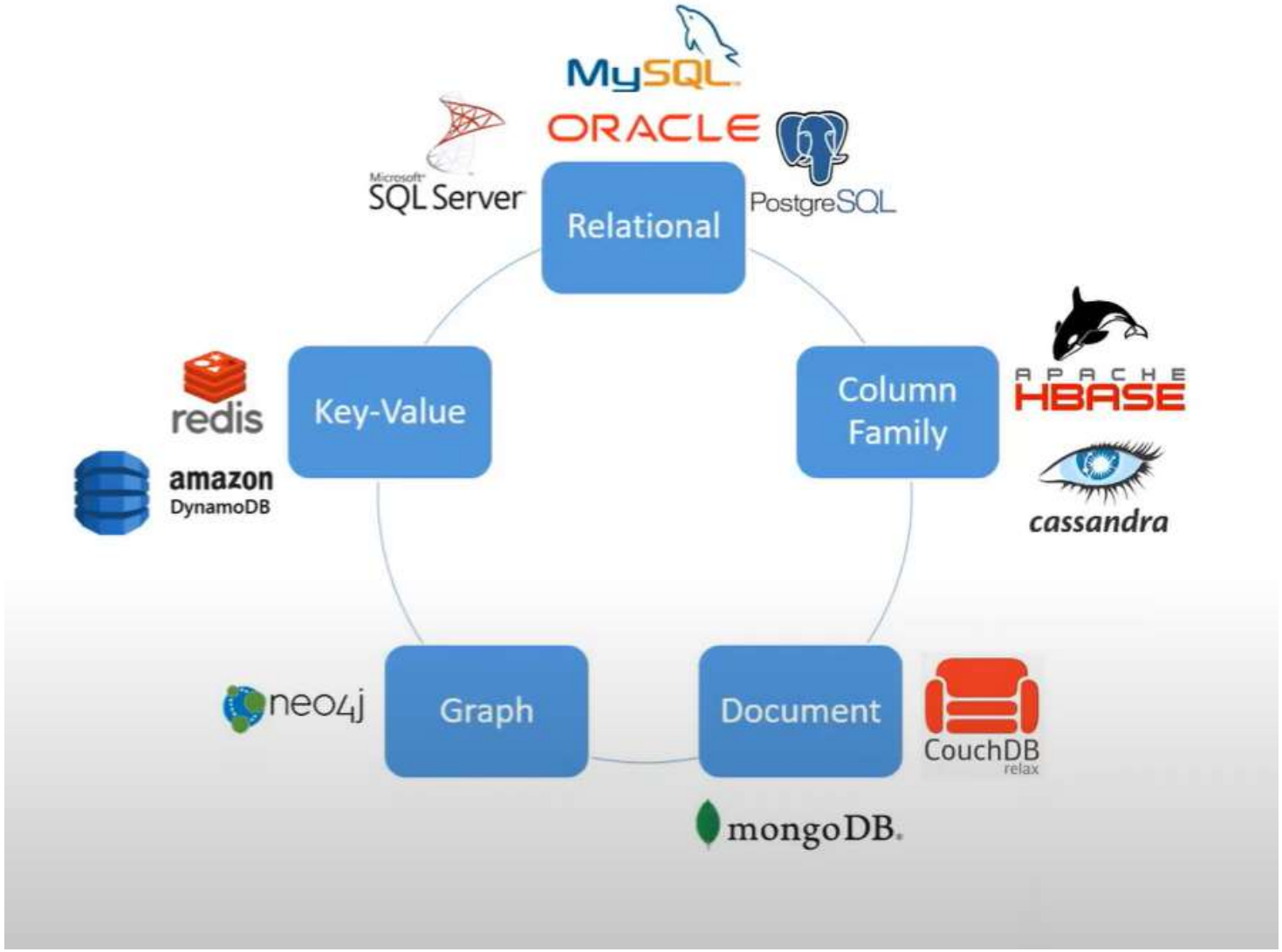
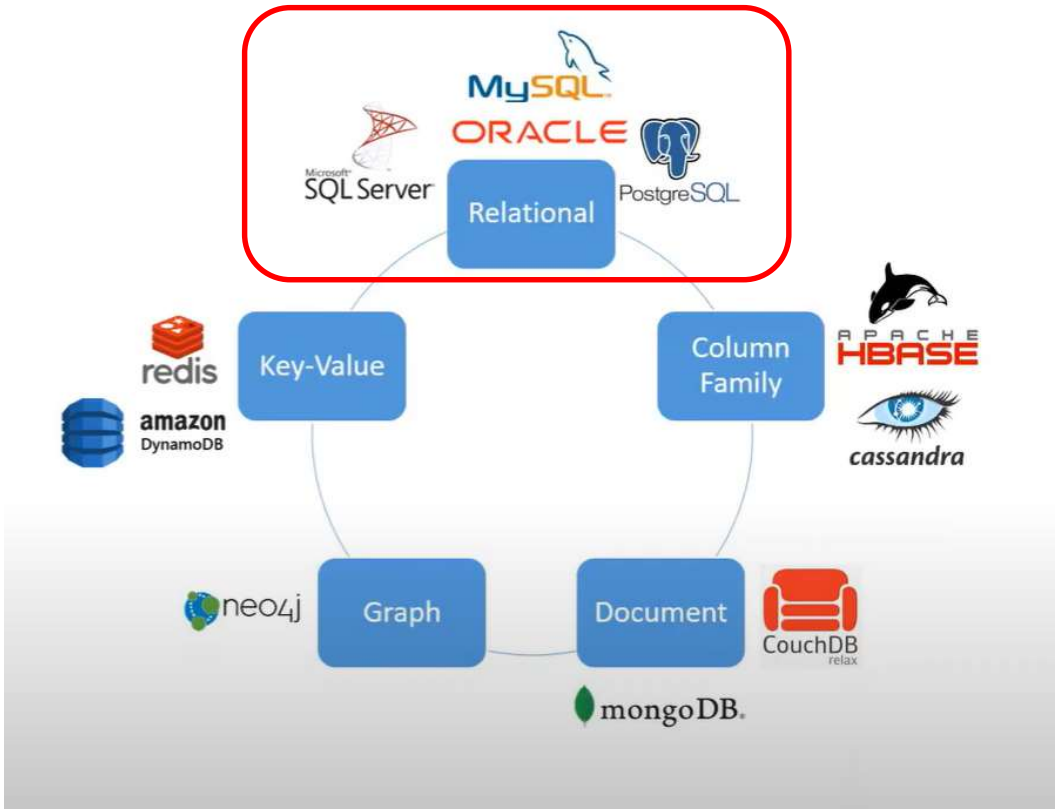
Lesson 4

DATABASES

Five different types of databases





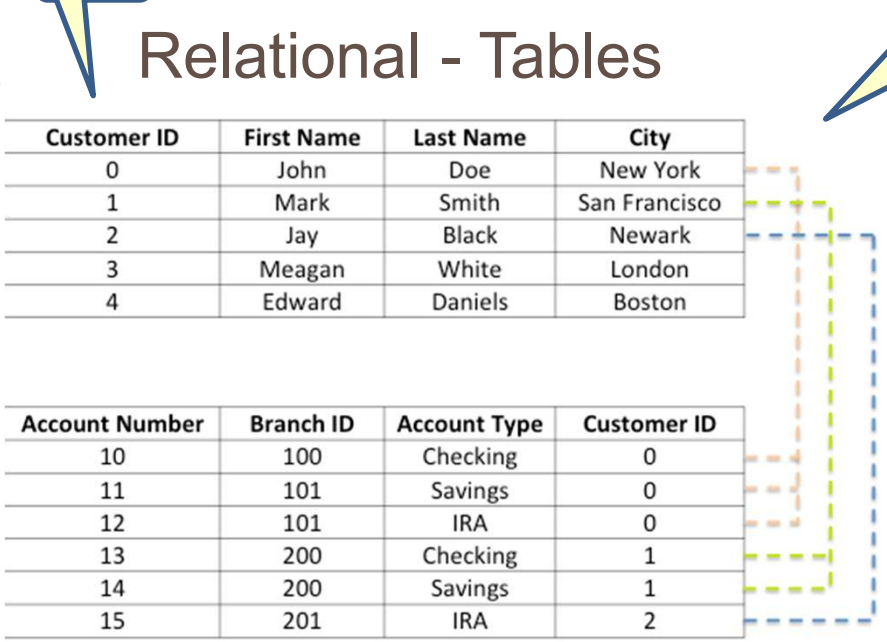
RELATIONAL DATABASE

Relational databases

Data is normalized: No duplication of data

Key

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



Foreign key

To find related data you need to JOIN tables together

SQL as language to manage data in the database

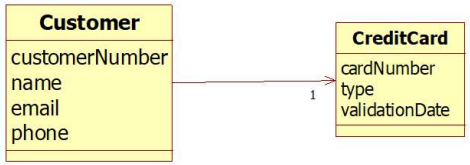
SPRING BOOT JPA EXAMPLE

Domain classes + repository

```java
@Entity
public class Customer {
  @Id
  private int customerNumber;
  private String name;
  private String email;
  private String phone;
  @ManyToOne(cascade =  CascadeType.ALL)
  private CreditCard creditCard;
```

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

```java
public interface CustomerRepository extends JpaRepository<Customer, Integer> {
   Customer findByPhone(String phone);
   Customer findByName(String name);
   List<Customer> findByNameOrPhone(String name, String phone);

   @Query("select c from Customer c where c.email= :email")
   Customer findCustomerWithEmail(String email);

   @Query("select c from Customer c where c.creditCard.type= :cctype")
   List<Customer>  findCustomerWithCreditCardType(String cctype);


}
```



Application(1/2)

```
@SpringBootApplication
@EnableJpaRepositories("repositories")
@EntityScan("domain")
public class CustomerApplication implements CommandLineRunner{

  @Autowired
  CustomerRepository customerrepository;

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

  @Override
  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);
```

Application(2/2)

```
//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);
    System.out.println("-----------find by phone ----------------");
    System.out.println(customerrepository.findByPhone("0622341678"));
    System.out.println("-----------find by email ----------------");
    System.out.println(customerrepository.findCustomerWithEmail("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);
    }

    System.out.println("-----------find by name ----------------");
    System.out.println(customerrepository.findByName("John doe"));

  }

}
```

application.properties

```
spring.datasource.url=jdbc:hsqldb:hsql://localhost/trainingdb
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver

spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.HSQLDialect

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

Tables

```
SELECT * FROM customer
```

CUSTOMER_NUMBER	EMAIL	NAME	PHONE	CREDIT_CARD_CARD_NUMBER
66	jj123@acme.com	James Johnson	068633452	99876549876
101	jd@gmail.com	John doe	0622341678	12324564321
109	jones@acme.com	John Jones	0624321234	657483342

```
SELECT * FROM credit_card
```

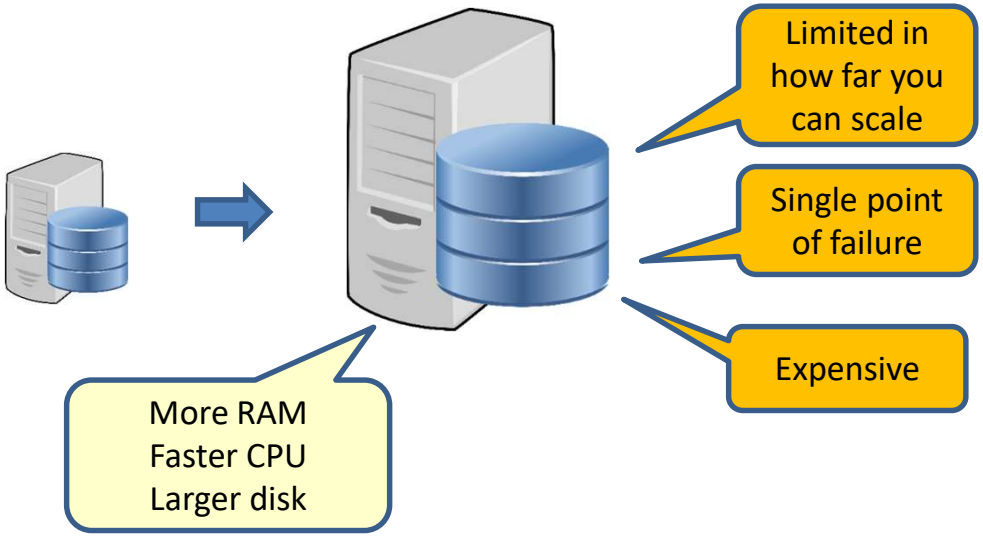
CARD_NUMBER	TYPE	VALIDATION_DATE
12324564321	Visa	11/23
657483342	Visa	09/23
99876549876	MasterCard	01/24

Database problems

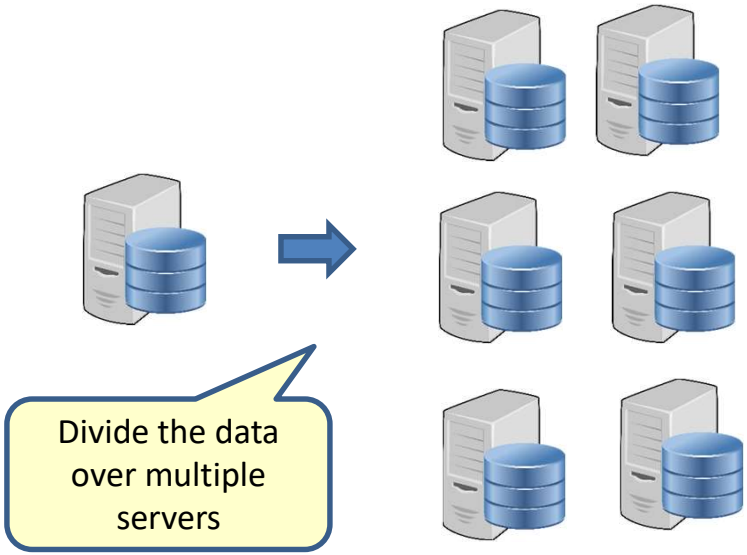
- Too much load
  - The one database node cannot handle the number of requests
- Too much data
  - The data does not fit anymore on one node

Database Scaling

- Vertical scaling



- Horizontal scaling



Distributed system

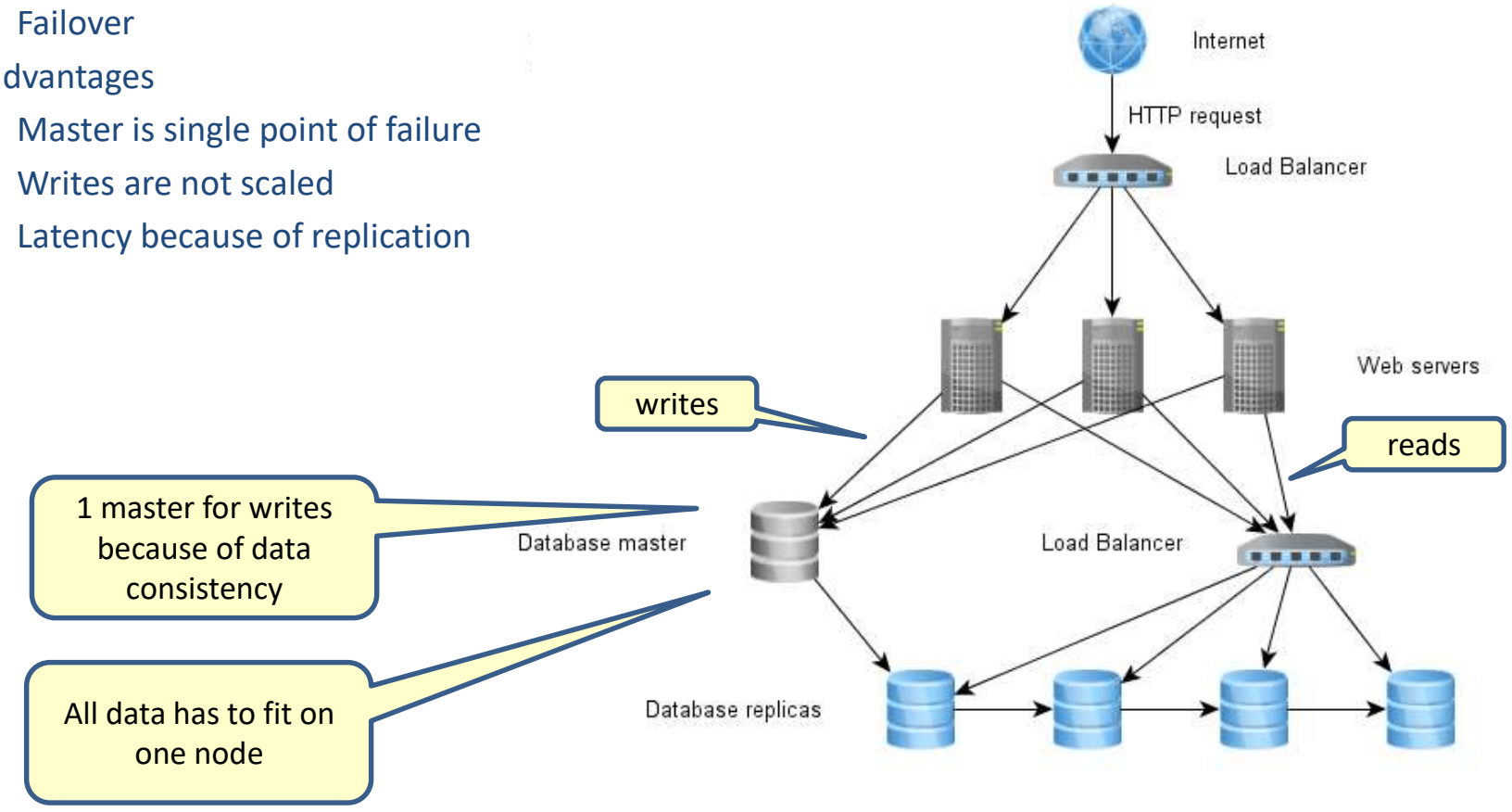
- Performance problem
  - Network calls are slow
- Reliability problem
  - Network failures

Solving database problems

- Too much load
  - Master-slave replication
- Too much data
  - Sharding

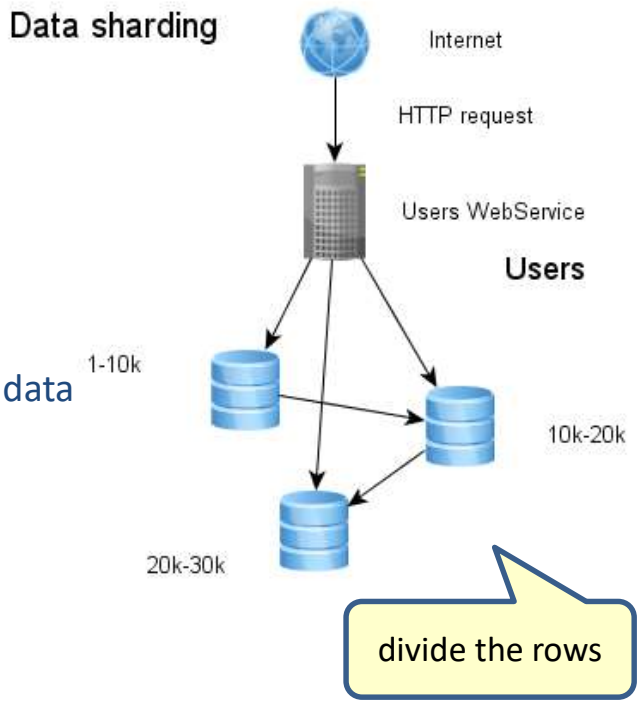
Master-slave replication

- Advantages
  - Reads can be load balanced over many databases
  - Failover
- Disadvantages
  - Master is single point of failure
  - Writes are not scaled
  - Latency because of replication



Sharding

- Advantage
  - We can scale the write operations
- Disadvantages
  - Complexity
    - Queries
    - Balancing of the shards
  - Performance
    - If you need to access multiple databases to get your data



When to use what?

- Master-slave replication
  - When you have too much load
  - More reads than writes
- Sharding
  - When you have to much data

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

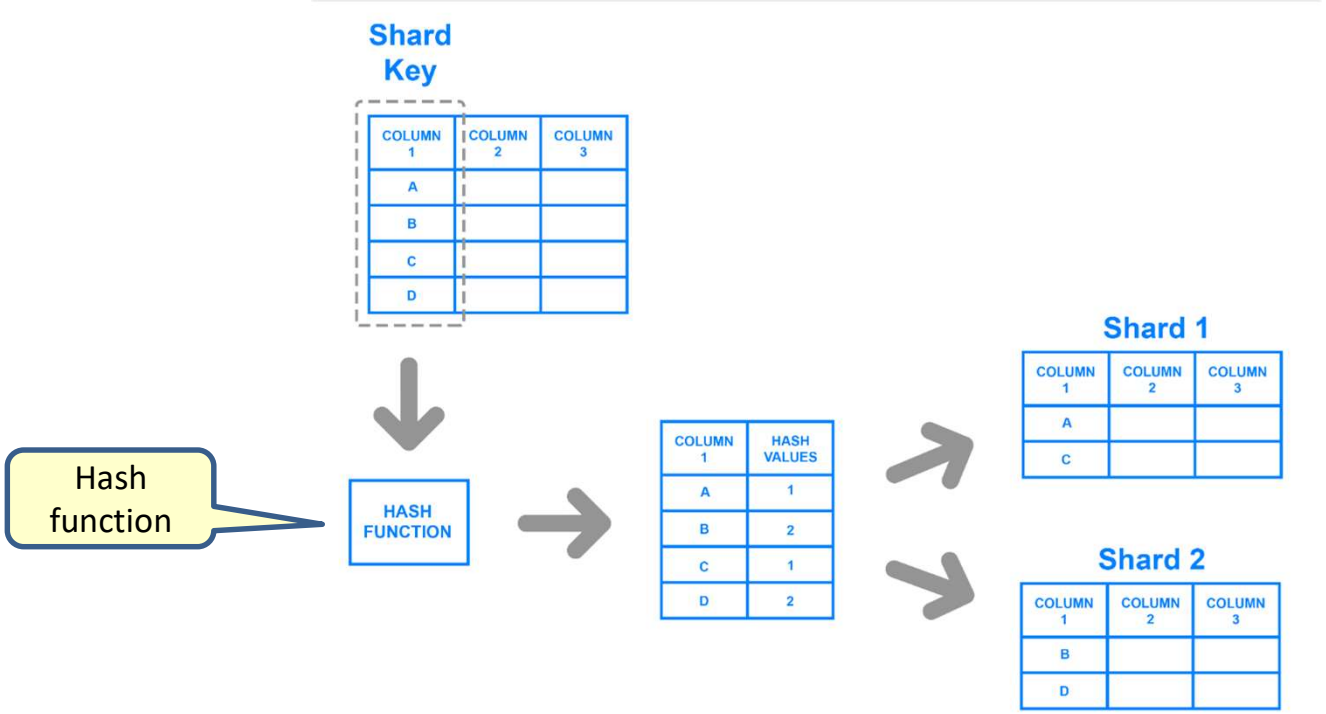
HP1

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

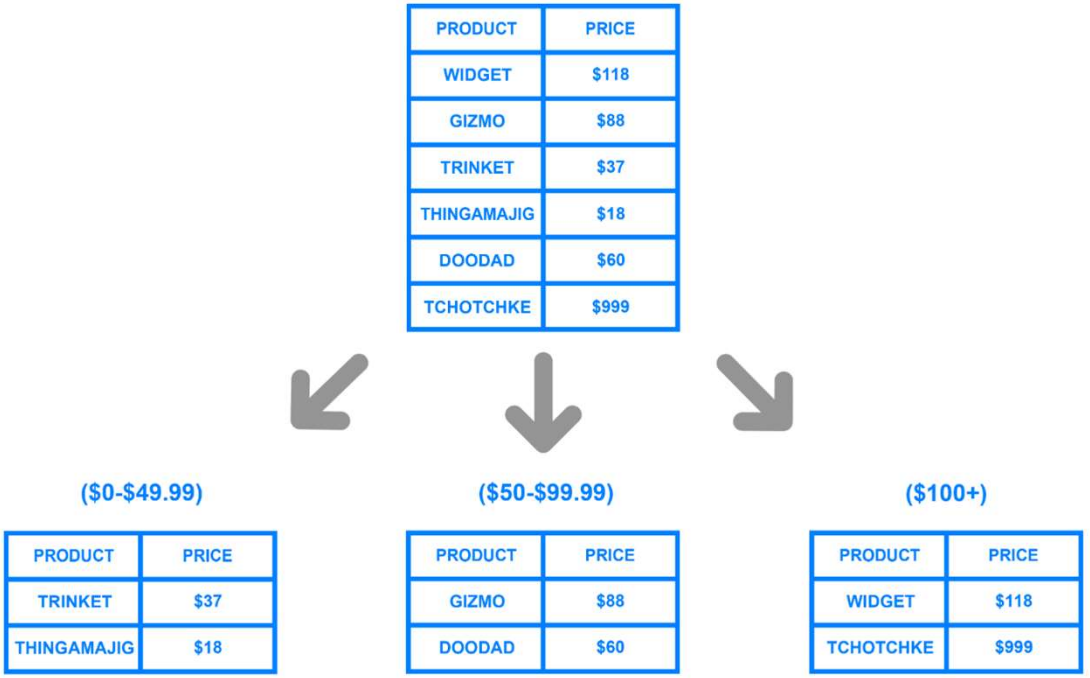
HP2

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

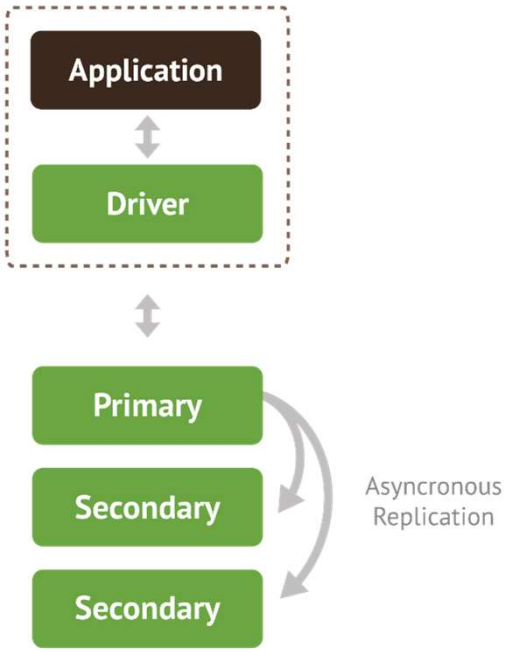
Key based sharding



Range based sharding



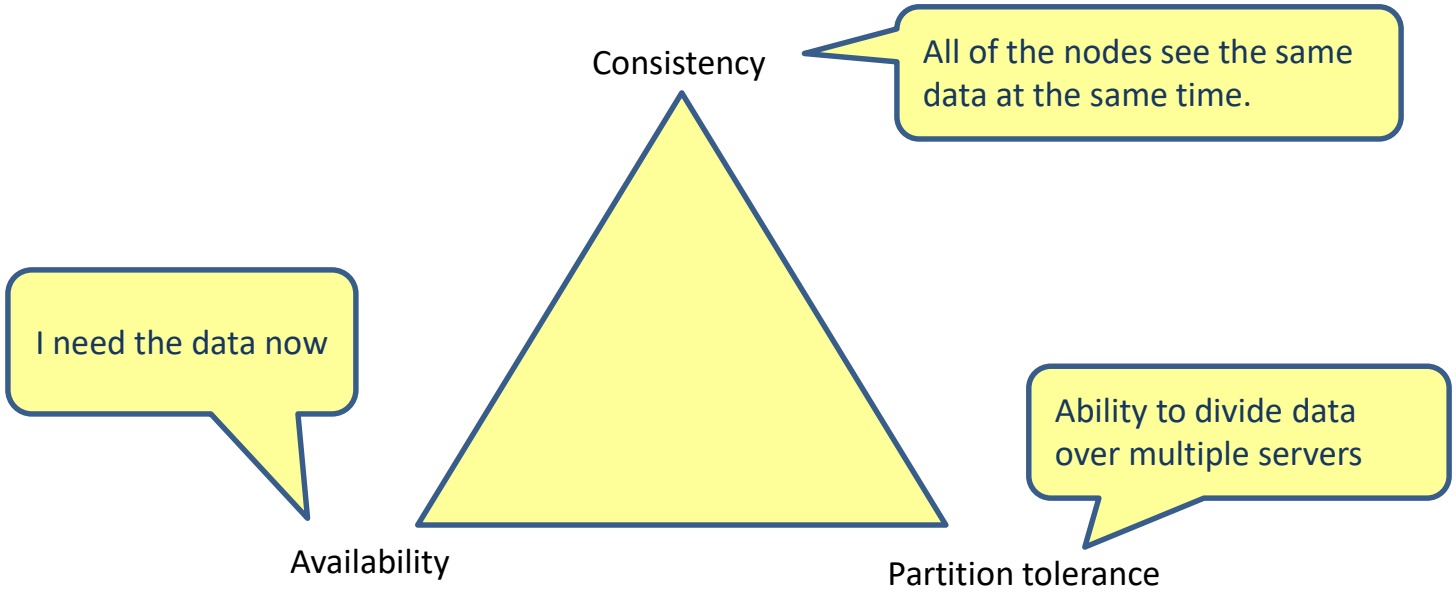
Replica Sets



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

Brewer‘s CAP Theorem

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

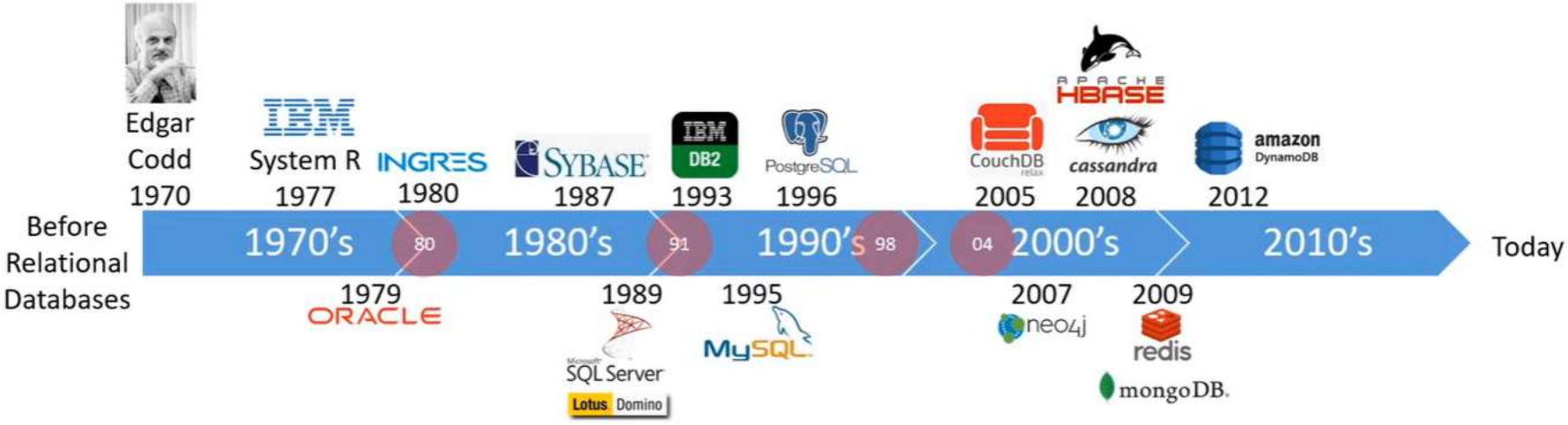


Consistency

- 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

Today's requirements on databases

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



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

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

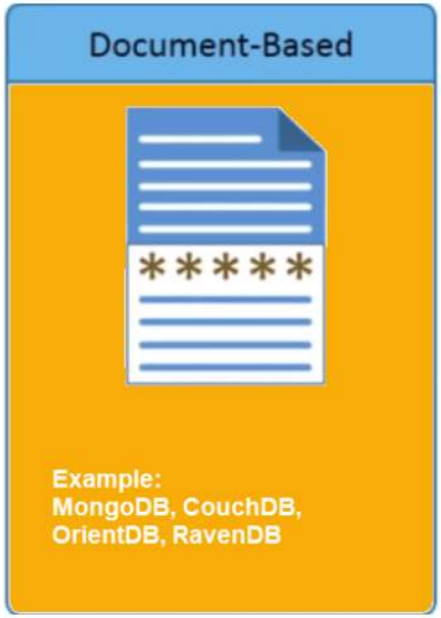
ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.

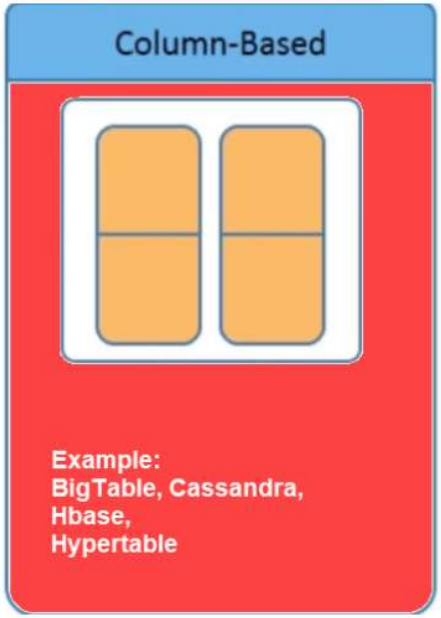
A relational database does not handle unstructured and semi structured data very well

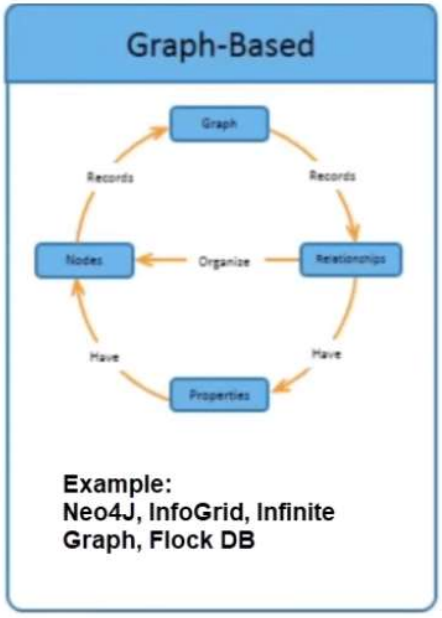
NoSQL characteristics

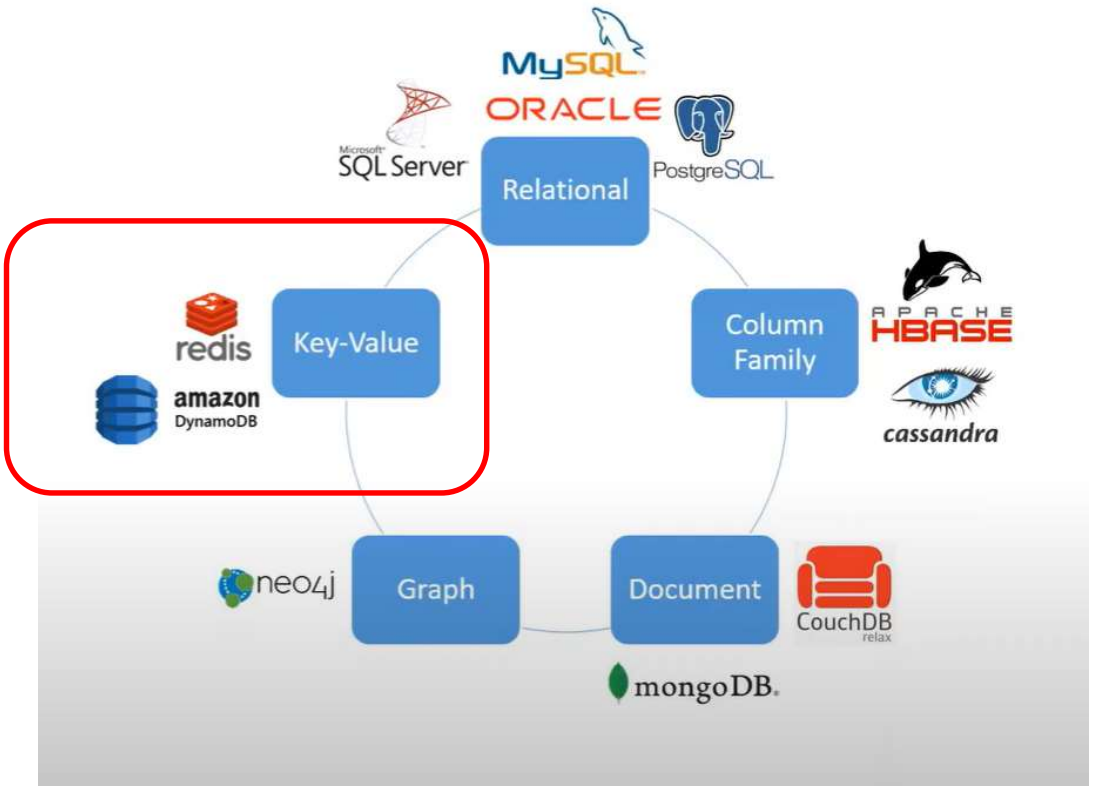
- Key-value store
- No fixed schema
- Can scale (almost) unlimited
  - Eventual consistency

Types of NoSQL databases





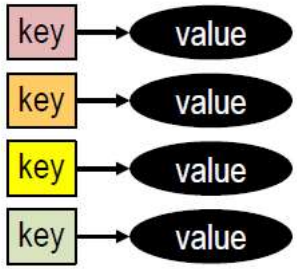




KEY VALUE STORE

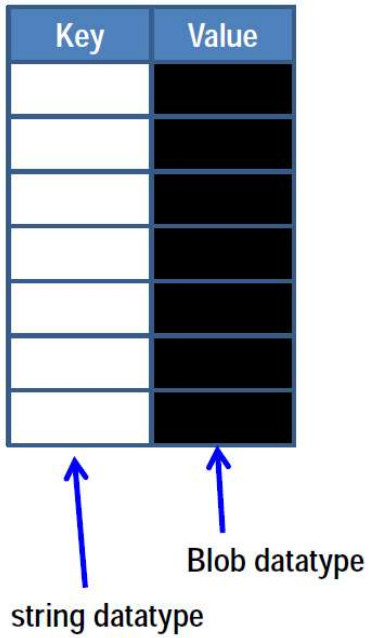
Key-value store

- One key -> one value



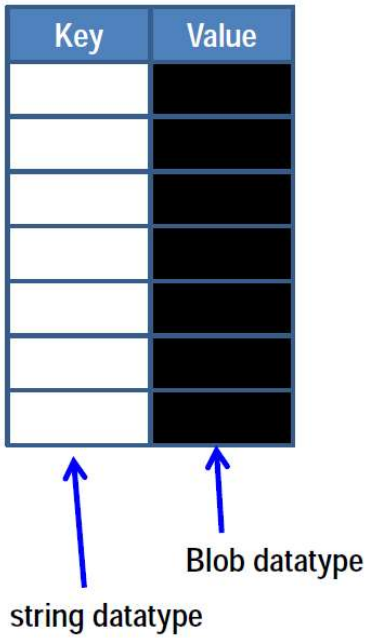
Key-value store

- 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

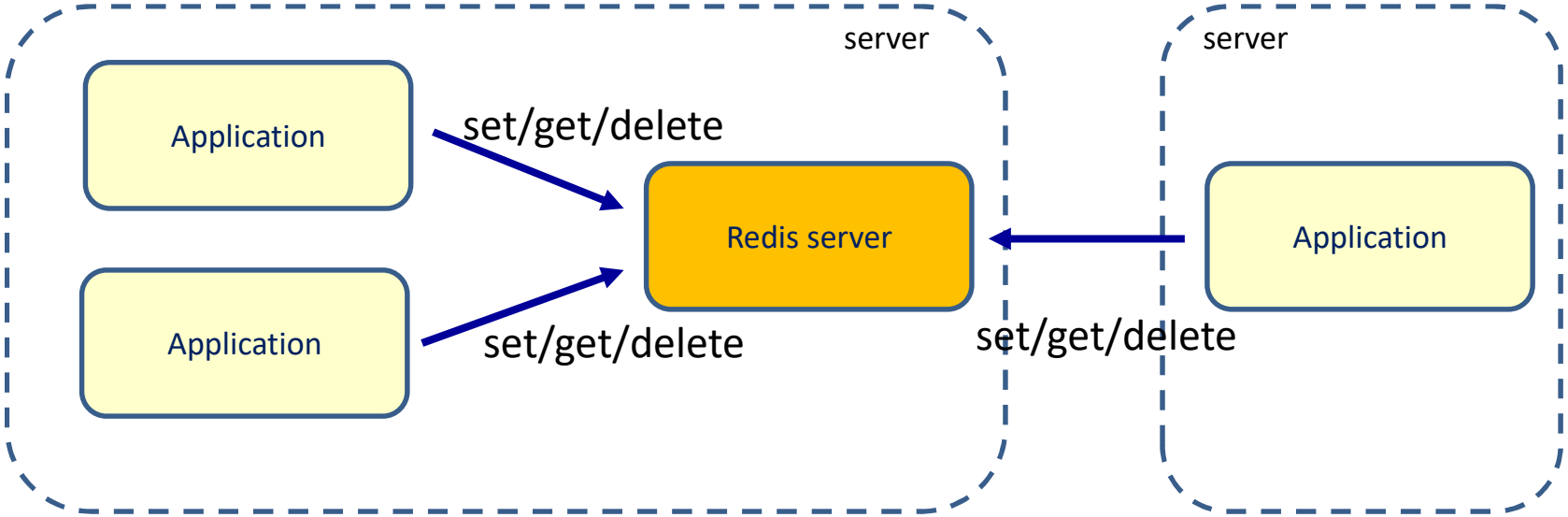
- **Pros:**
  - Very fast
  - Scalable
  - Simple API
    - Put(key, value)
    - Get(key)
    - Delete(key)
- **Cons:**
  - No way to query based on the content of the value



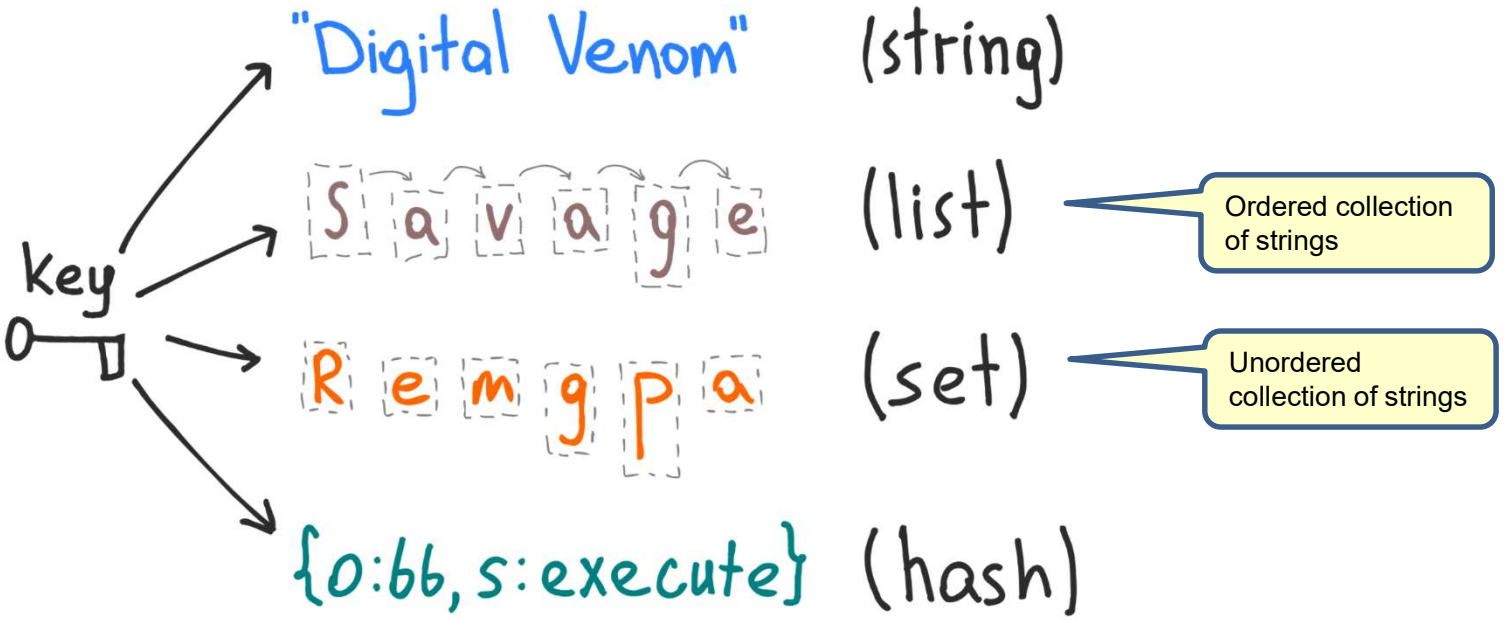
Redis

- **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



Redis data types

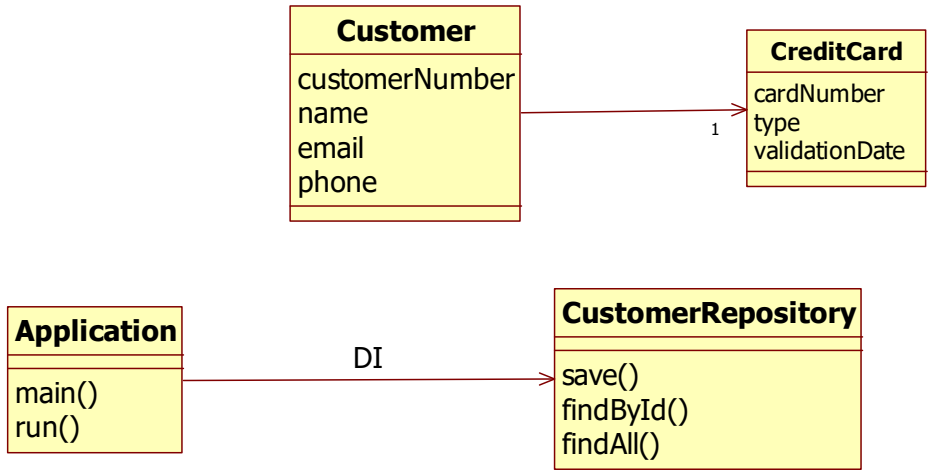


SPRING BOOT REDIS EXAMPLE

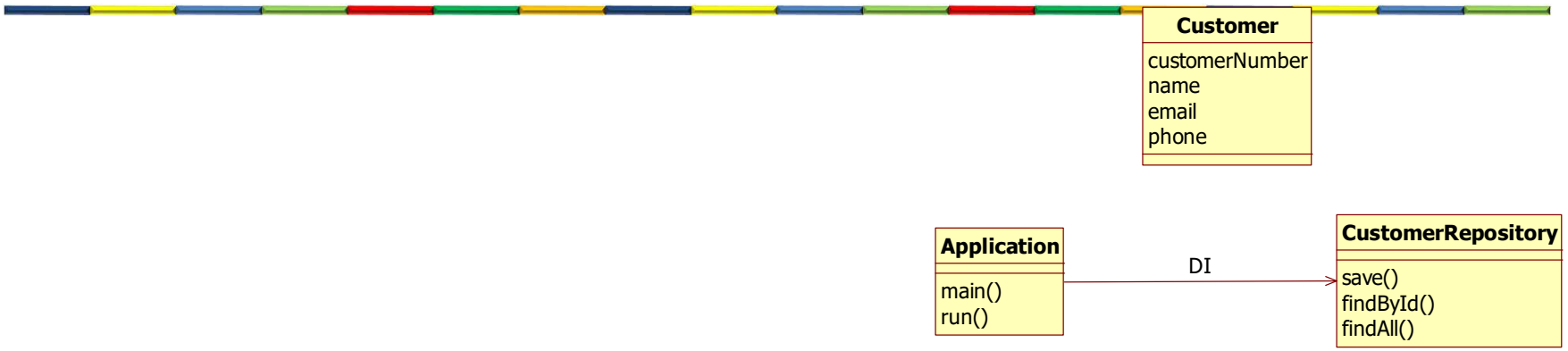
Domain classes

```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
```

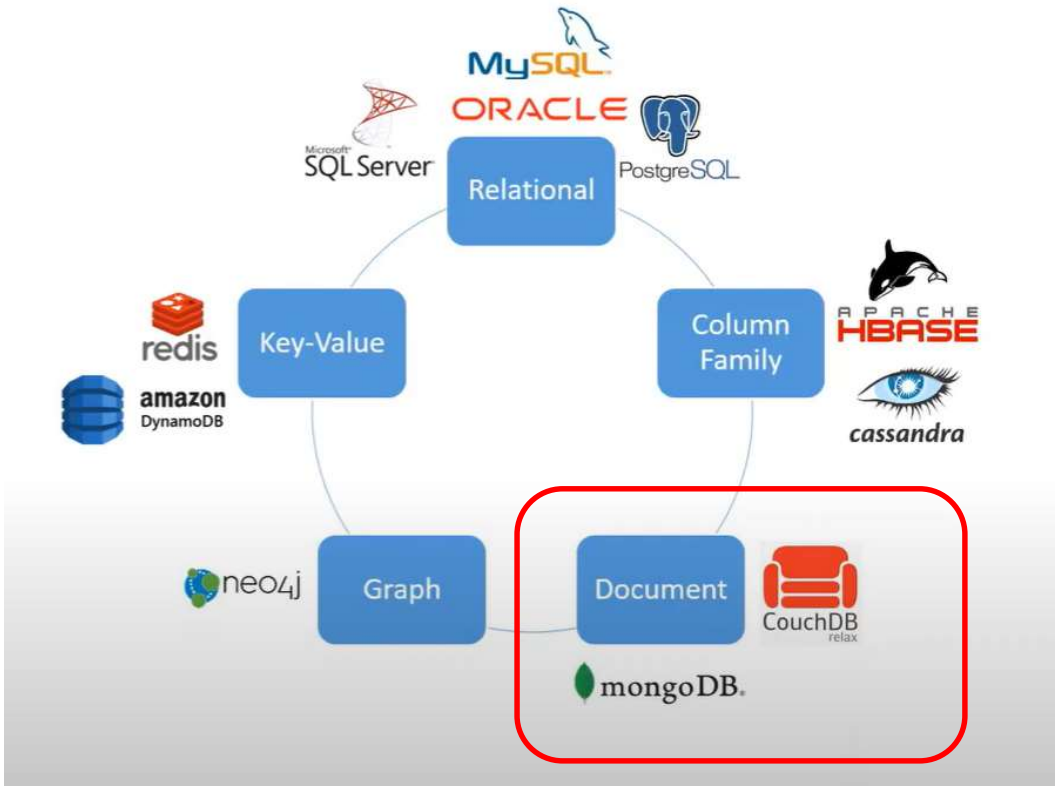
Application

```
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());
}
```

Redis explorer

```
>> keys *

1) "Customer:101"
```

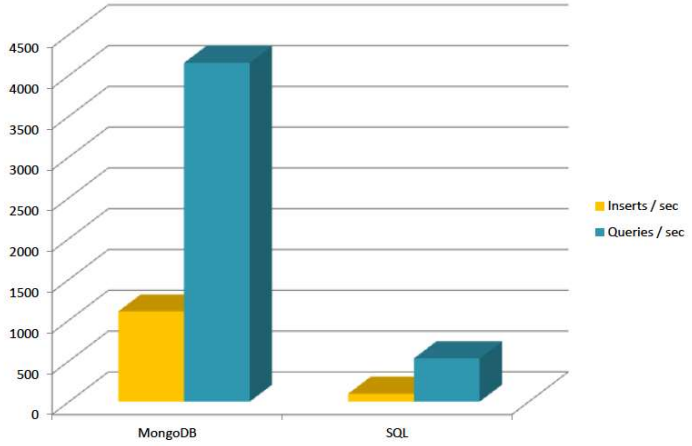
```
>> hgetall Customer:101

 1) "_class"
 2) "customers.app1.Customer"
 3) "customerNumber"
 4) "101"
 5) "name"
 6) "John doe"
 7) "email"
 8) "jd@gmail.com"
 9) "phone"
10) "0622341678"
11) "creditCard.cardNumber"
12) "12324564321"
13) "creditCard.type"
14) "Visa"
15) "creditCard.validationDate"
16) "11/23"
```



DOCUMENT DATABASE

MongoDB

- Document database
- Fast
- Can handle large datasets



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: […]
    } ]
}
```

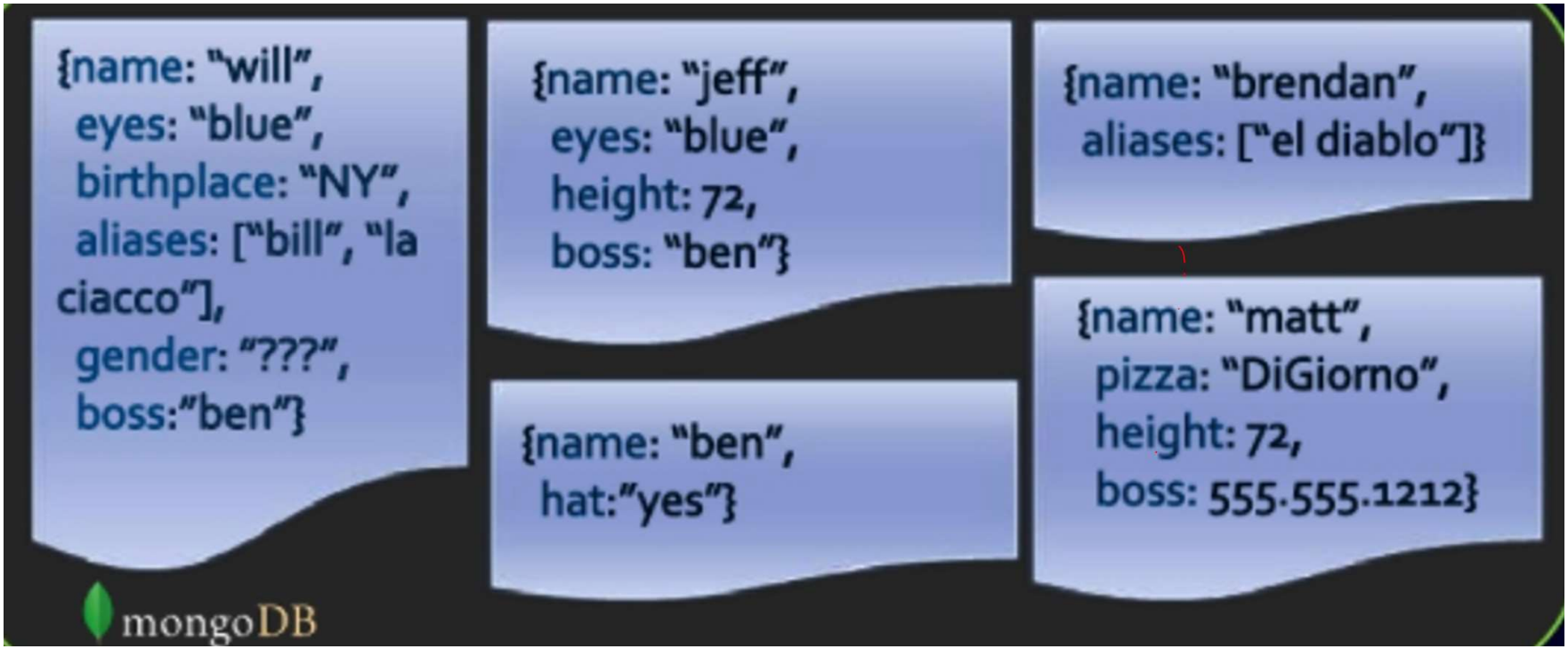
BSON

```
{ author: 'joe',
 created : new Date('03/28/2009'),
 title : 'Yet another blog post',
 text : 'Here is the text...',
 tags : [ 'example', 'joe' ],
 comments : [
              { author: 'jim',
                comment: 'I disagree'
              },
              { author: 'nancy',
                comment: 'Good post'
              }
 ]
}
```

Remember it is stored in binary formats (BSON)

```
"\x16\x00\x00\x00\x02hello\x00
 \x06\x00\x00\x00world\x00\x00"

"1\x00\x00\x00\x04BSON\x00&\x00
 \x00\x00\x020\x00\x08\x00\x00
 \x00awesome\x00\x011\x00333333
 \x14@\x102\x00\xc2\x07\x00\x00
 \x00\x00"
```

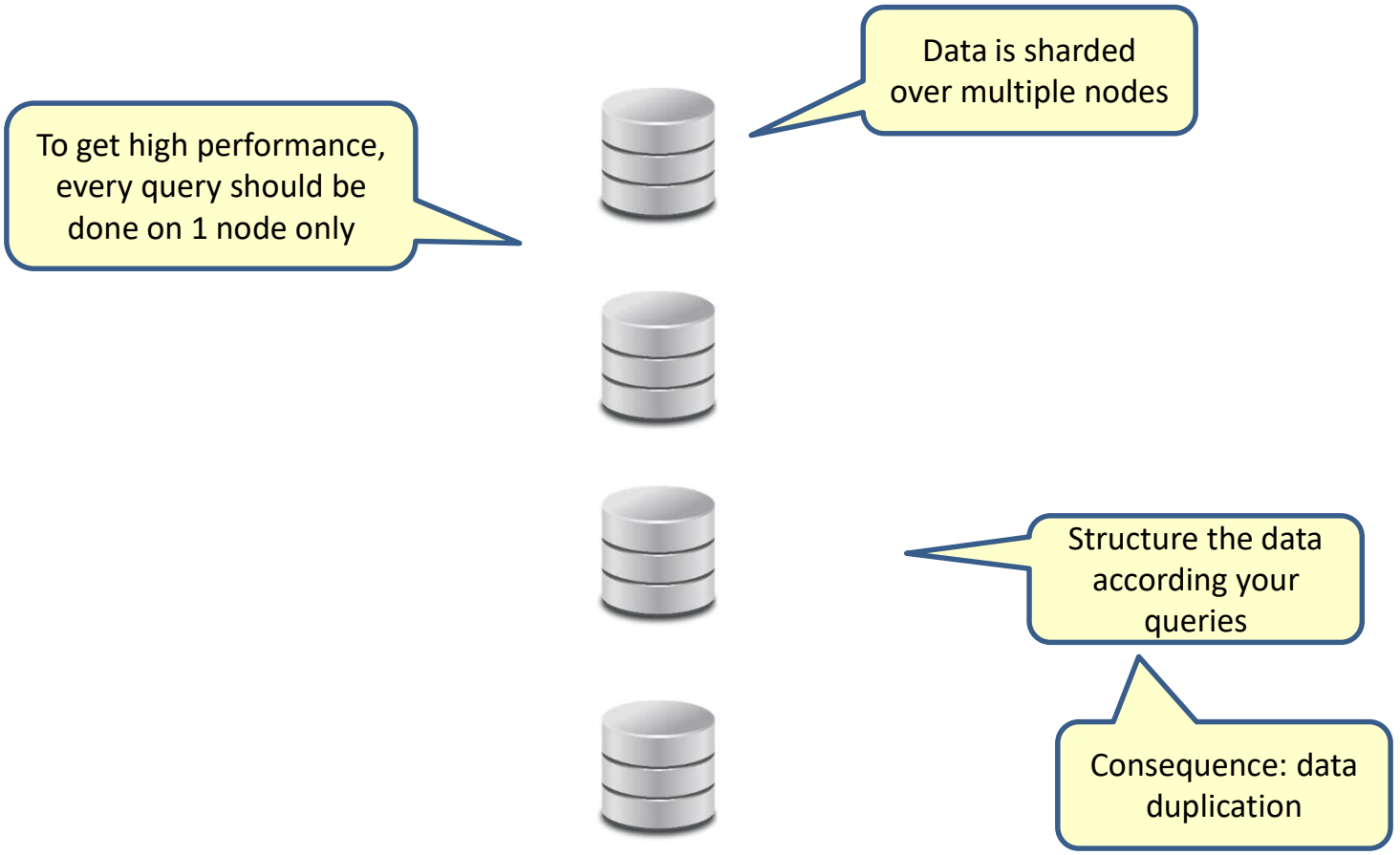
Schema free



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?

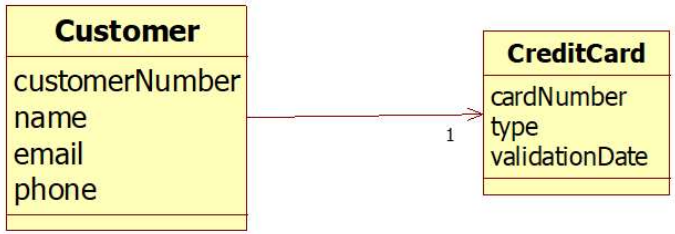


SPRING BOOT MONGO EXAMPLE

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);
}
```

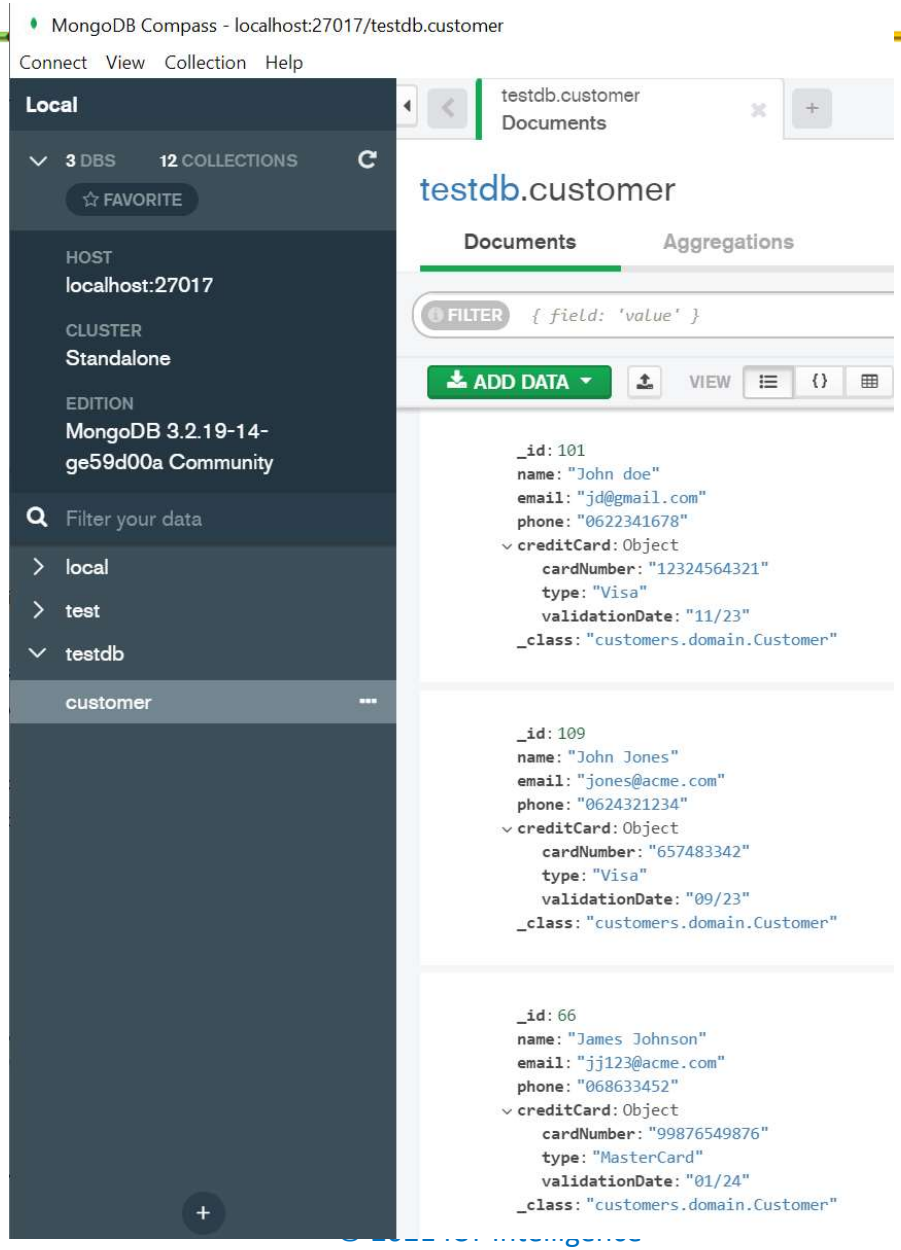
Application (1/2)

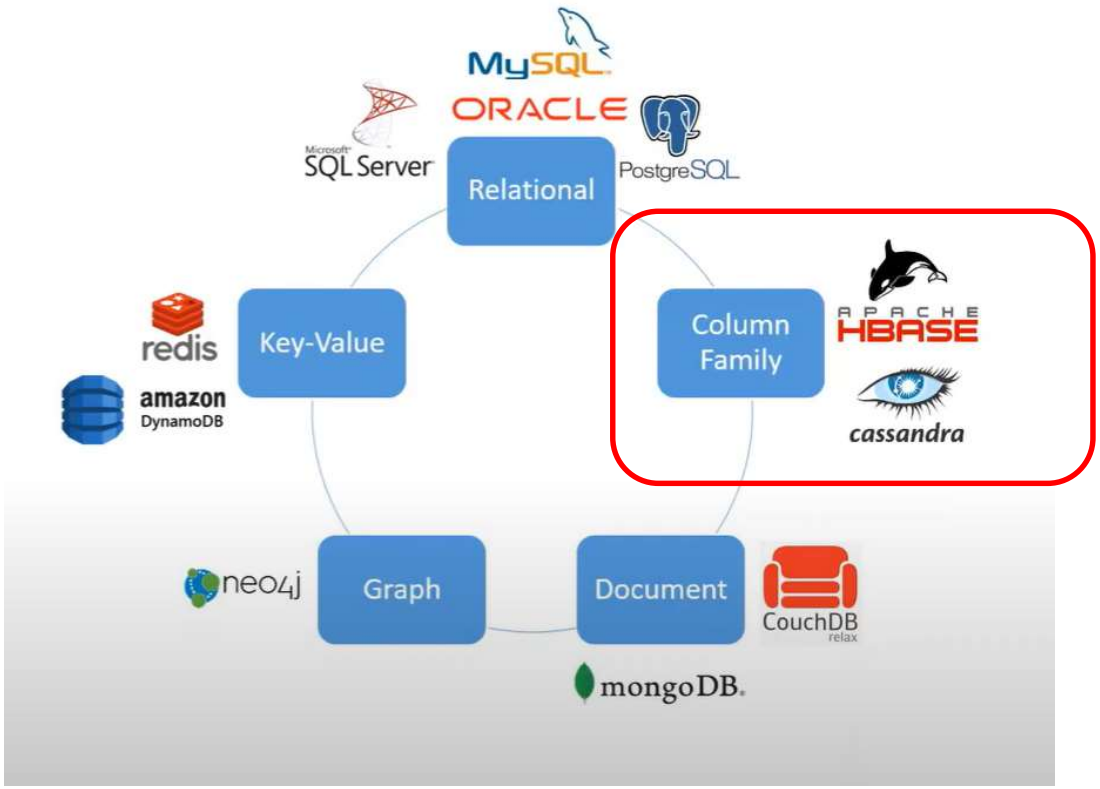
```
public void run(String... args) throws Exception {
 // create customer
Customer customer = new Customer(101,"John doe", "johnd@acme.com", "0622341678");
CreditCard creditCard = new CreditCard("12324564321", "Visa", "11/23");
customer.setCreditCard(creditCard);
customerRepository.save(customer);
customer = 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

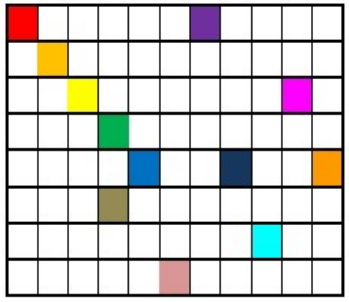




COLUMN FAMILY DATABASE

Column family

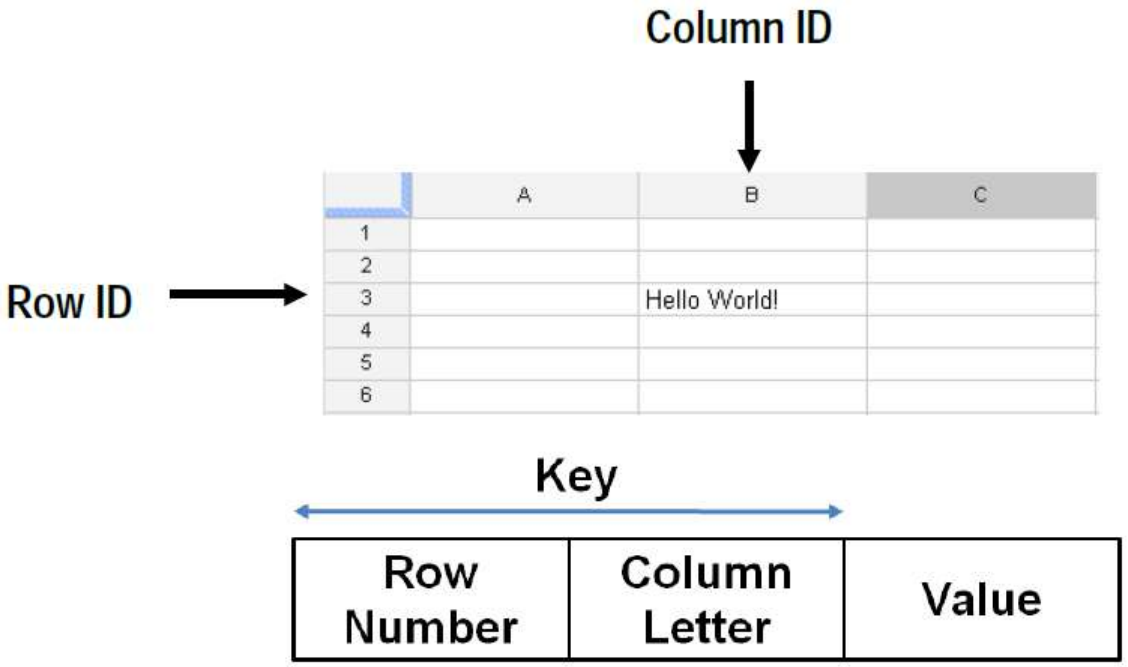
- Place data in a certain column
- Ideal for high-variability data sets
- Colum families allow to query all columns that have a specific property or properties
- Allow new columns to be inserted without doing an "alter table"



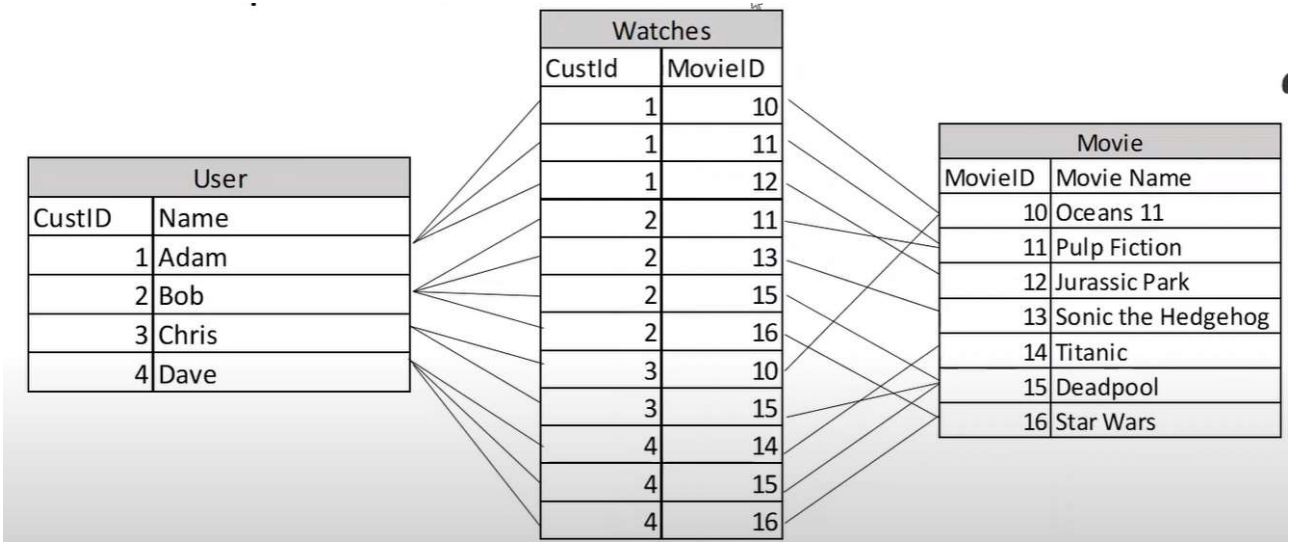
- Players
  - Cassandra
  - HBase
  - Google BigTable

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

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

Cassandra partitioning

customerNr	firstName	lastName
1	Frank	Brown
2	Bob	Johnson
3	John	Jackson
4	Frank	Young
5	Sue	Jones

customerNr	firstName	lastName
1	Frank	Brown
2	Bob	Johnson

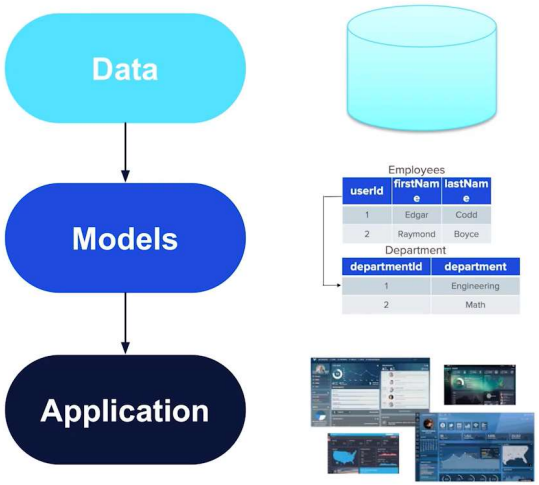
customerNr	firstName	lastName
3	John	Jackson
5	Sue	Jones

customerNr	firstName	lastName
4	Frank	Young

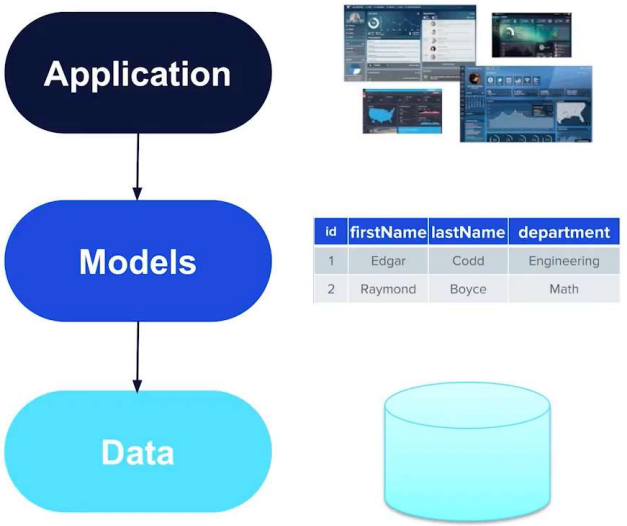
No Joins

Data modeling

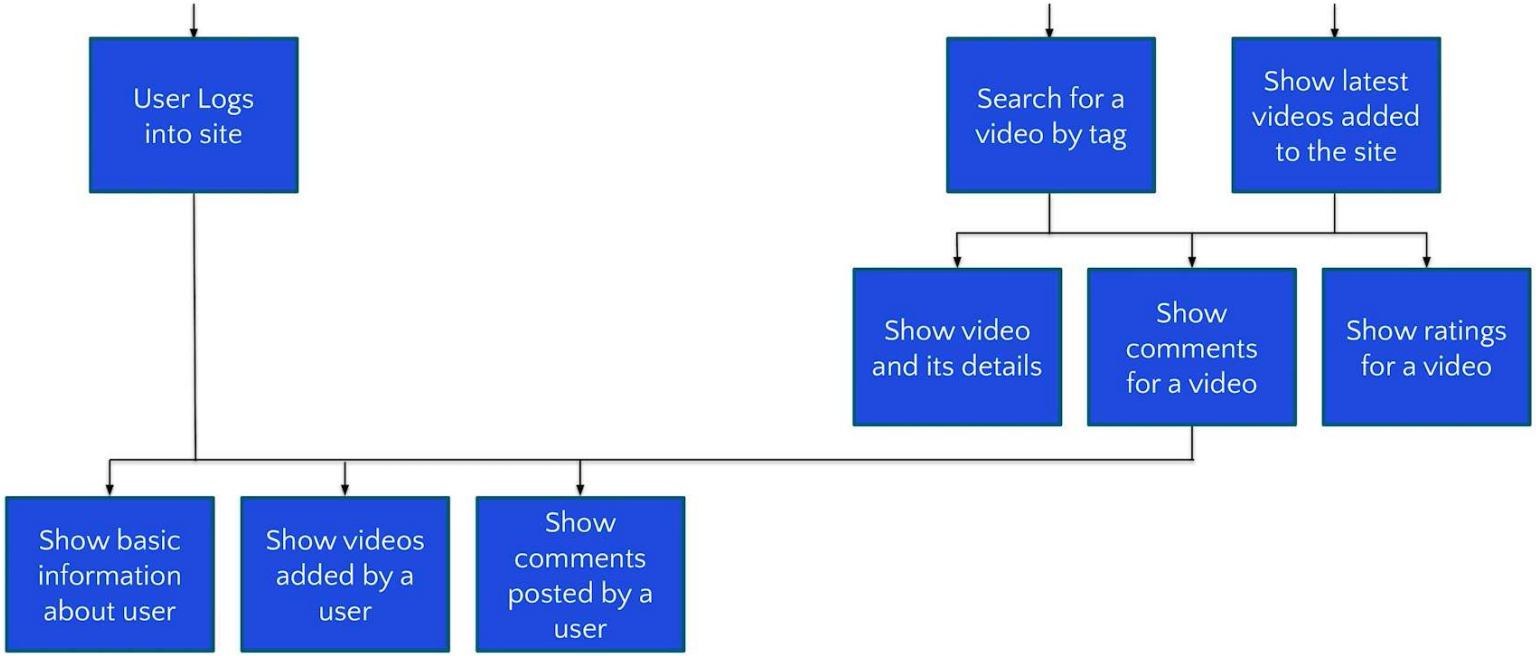
- RDBMS



- Cassandra



Video application workflow



Queries that support the workflow

Users

User Logs into site

Find user by email address

Show basic information about user

Find user by id

Comments

Show comments for a video

Find comments by video (latest first)

Show comments posted by a user

Find comments by user (latest first)

Ratings

Show ratings for a video

Find ratings by video

Relational way

- Single Users table with all user data and an Id Primary Key
- Add an index on email address to allow queries by email

Find user by email address

Find user by id

Cassandra way

User Logs into site

Find user by email address

```
CREATE TABLE user_credentials (
    email text,
    password text,
    userid uuid,
    PRIMARY KEY (email)
);
```

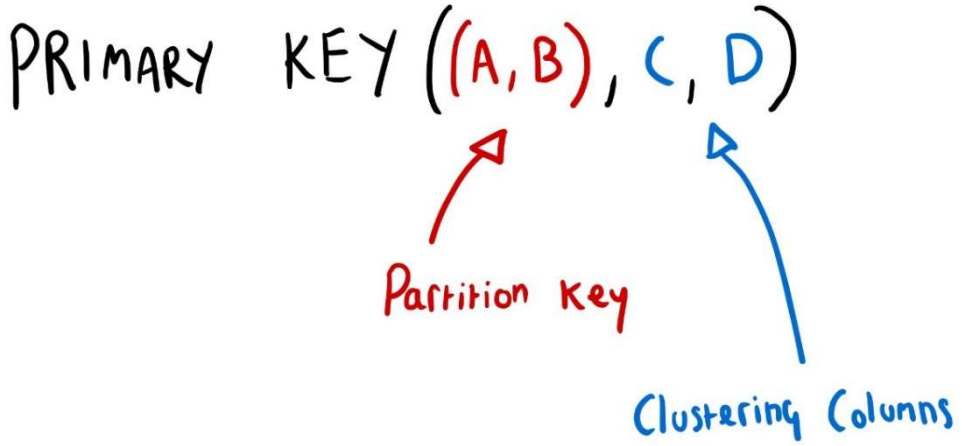
Show basic information about user

Find user by id

```
CREATE TABLE users (
    userid uuid,
    firstname text,
    lastname text,
    email text,
    created_date timestamp,
    PRIMARY KEY (userid)
);
```

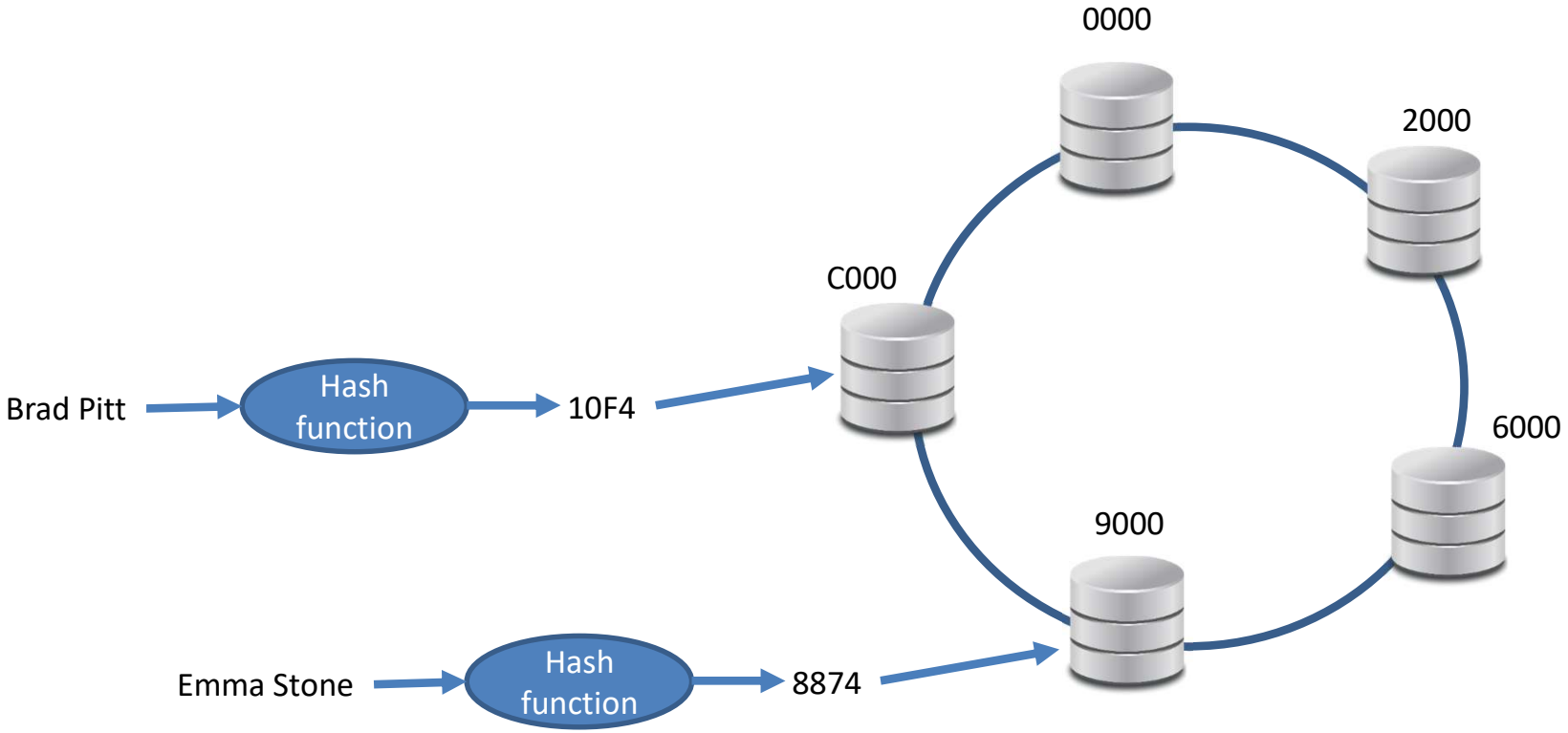
Cassandra keys

- Primary key
  - Partition key
  - Clustering columns
- Data types



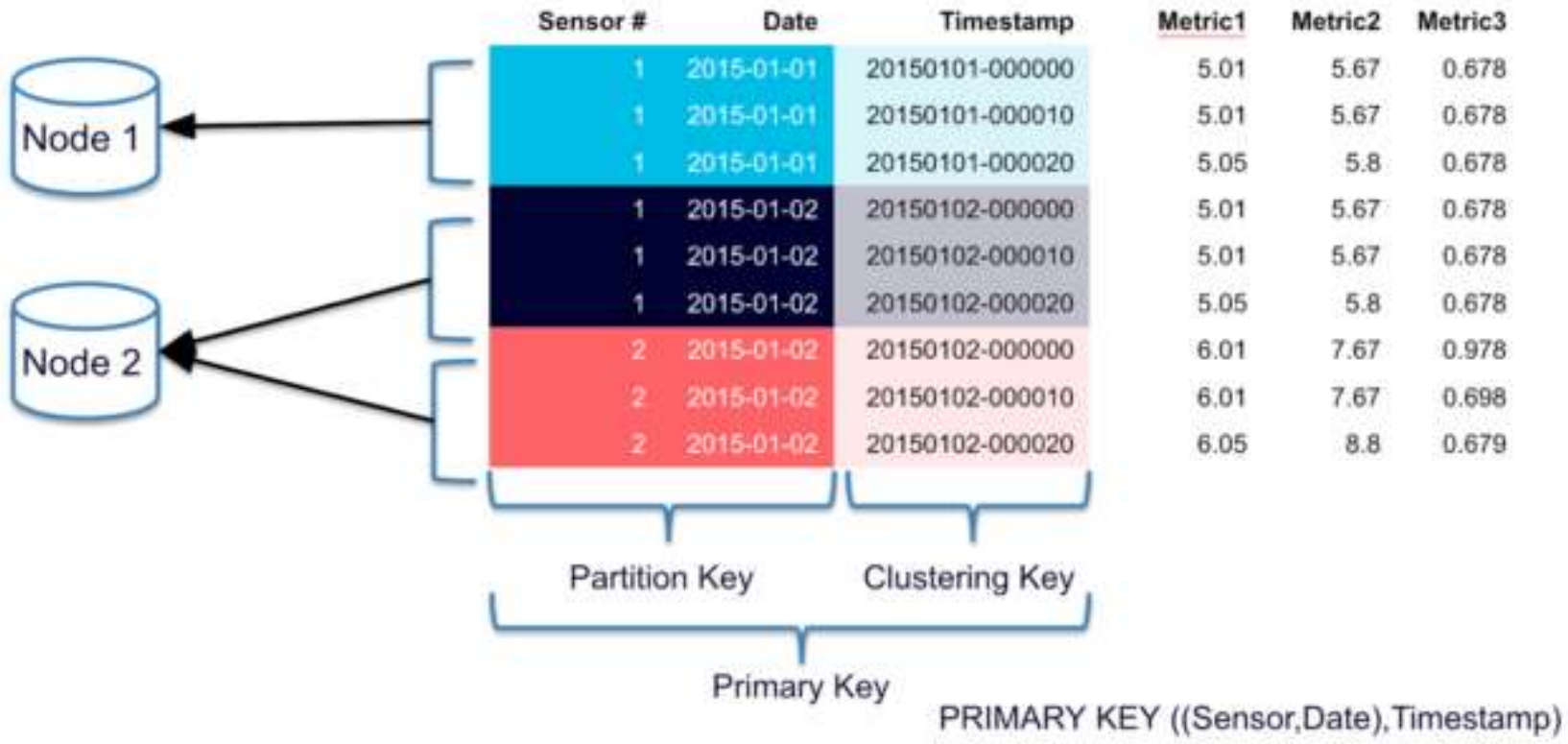
Partition key

- Decides where to place this record



Clustering columns

- Make the whole PK unique



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); ↩
```

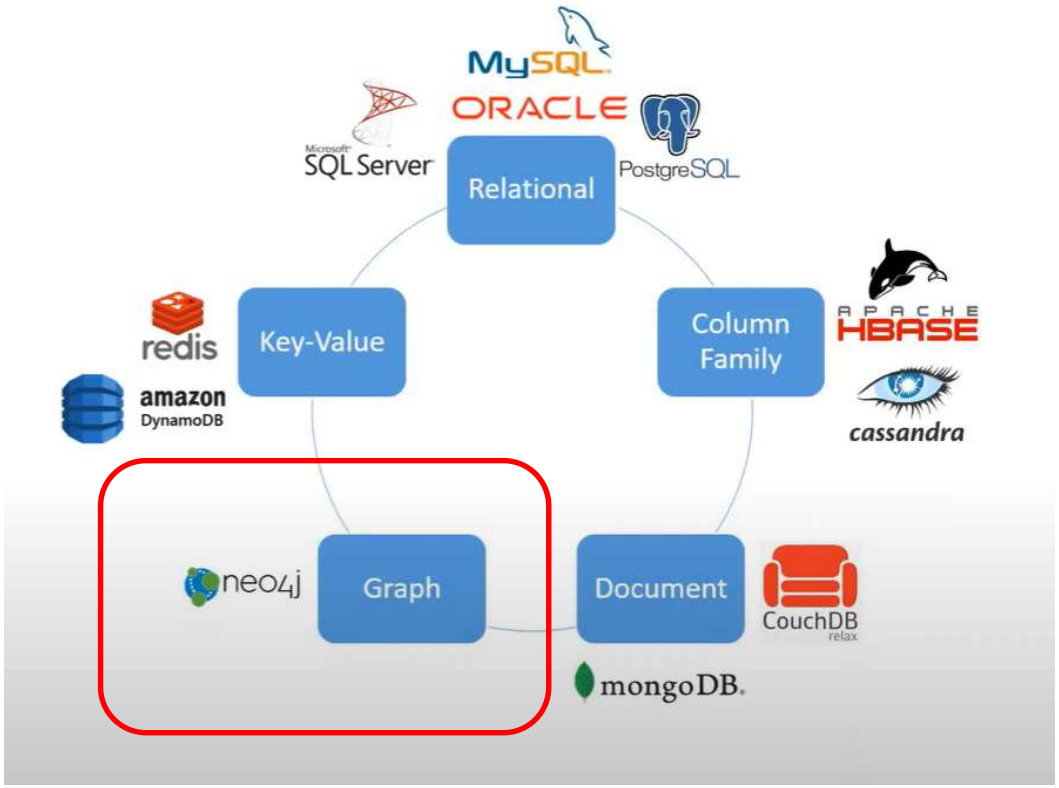
Data duplication

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

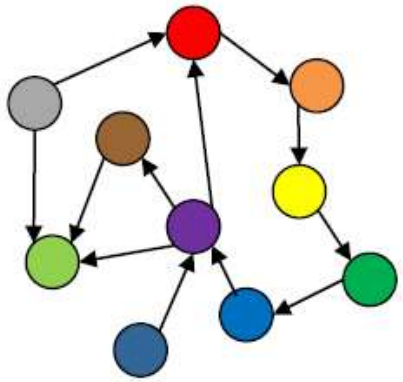
email	watched_on	title	year
joe@datastax.com	2020-04-28	Edward Scissorhands	1990
joe@datastax.com	2020-03-08	Alice in Wonderland	2010
joe@datastax.com	2020-02-13	Toy Story 3	2010
joe@datastax.com	2020-01-22	Despicable Me	2010
joe@datastax.com	2019-12-30	Alice in Wonderland	1951
jen@datastax.com	2011-10-01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



GRAPH DATABASE

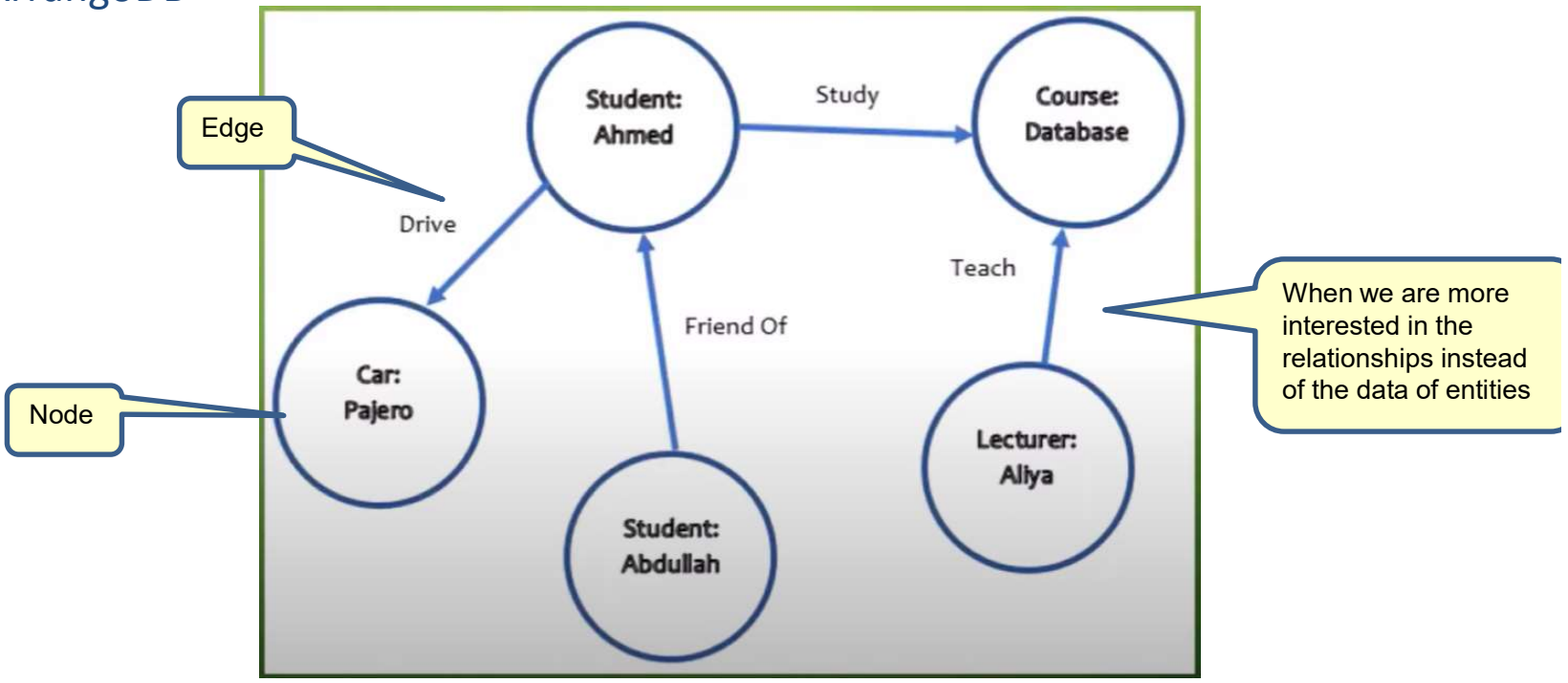
Graph

- Store entities as nodes with properties
- Store relationships as edges with properties
- Used when the relationship and relationships types between items are critical
- A query is like traversing the graph
  - Very fast



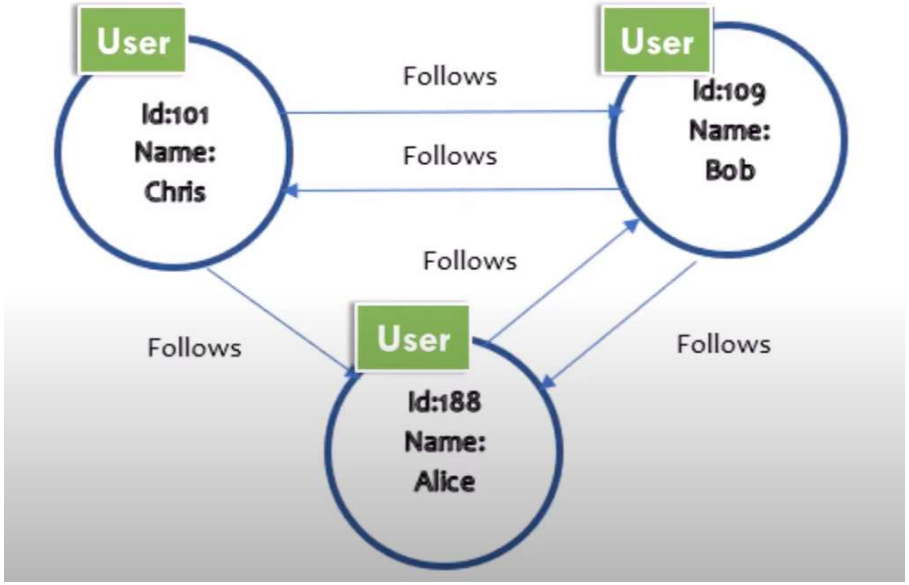
Graph database

- Players
  - Neo4J
  - OrientDB
  - ArrangoDB

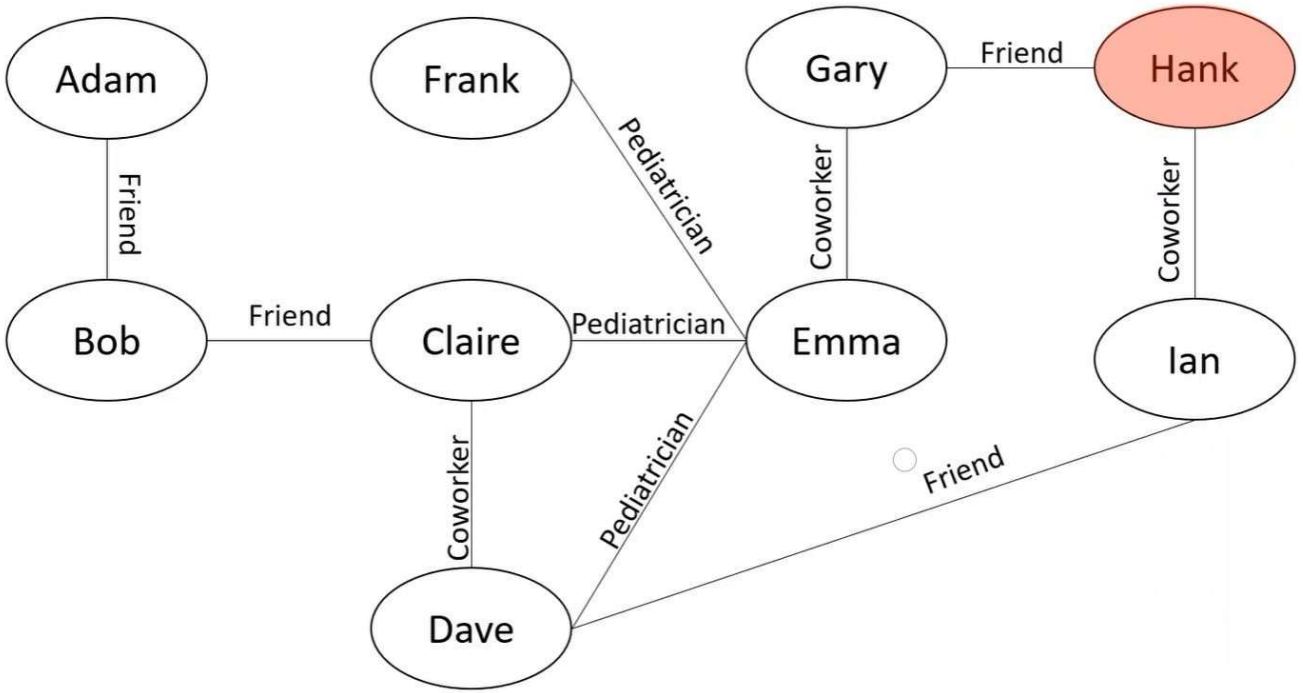


Graph DB use cases

- Social networks
- Knowledge graphs
- Fraud investigation
- Contact tracing



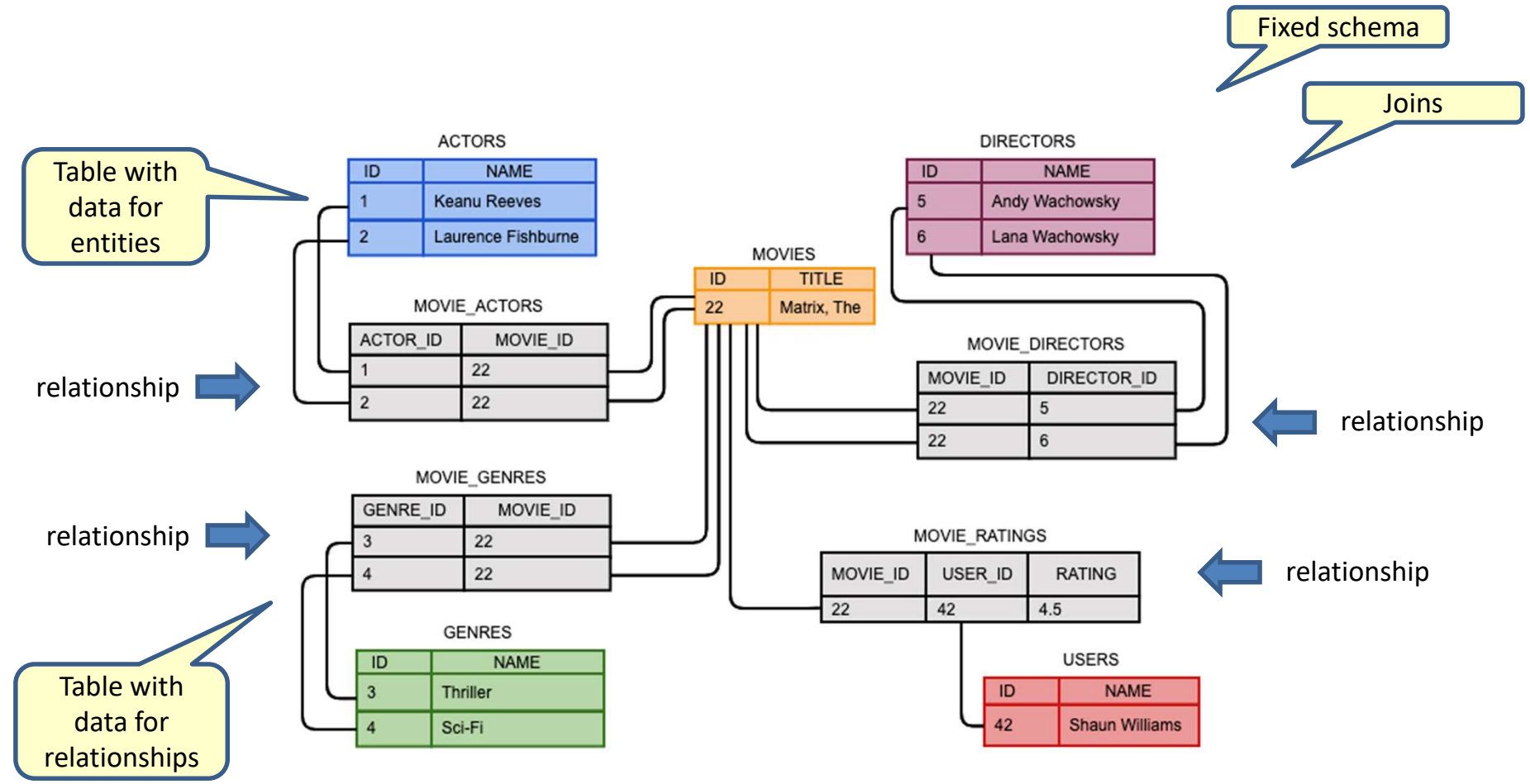
Graphs



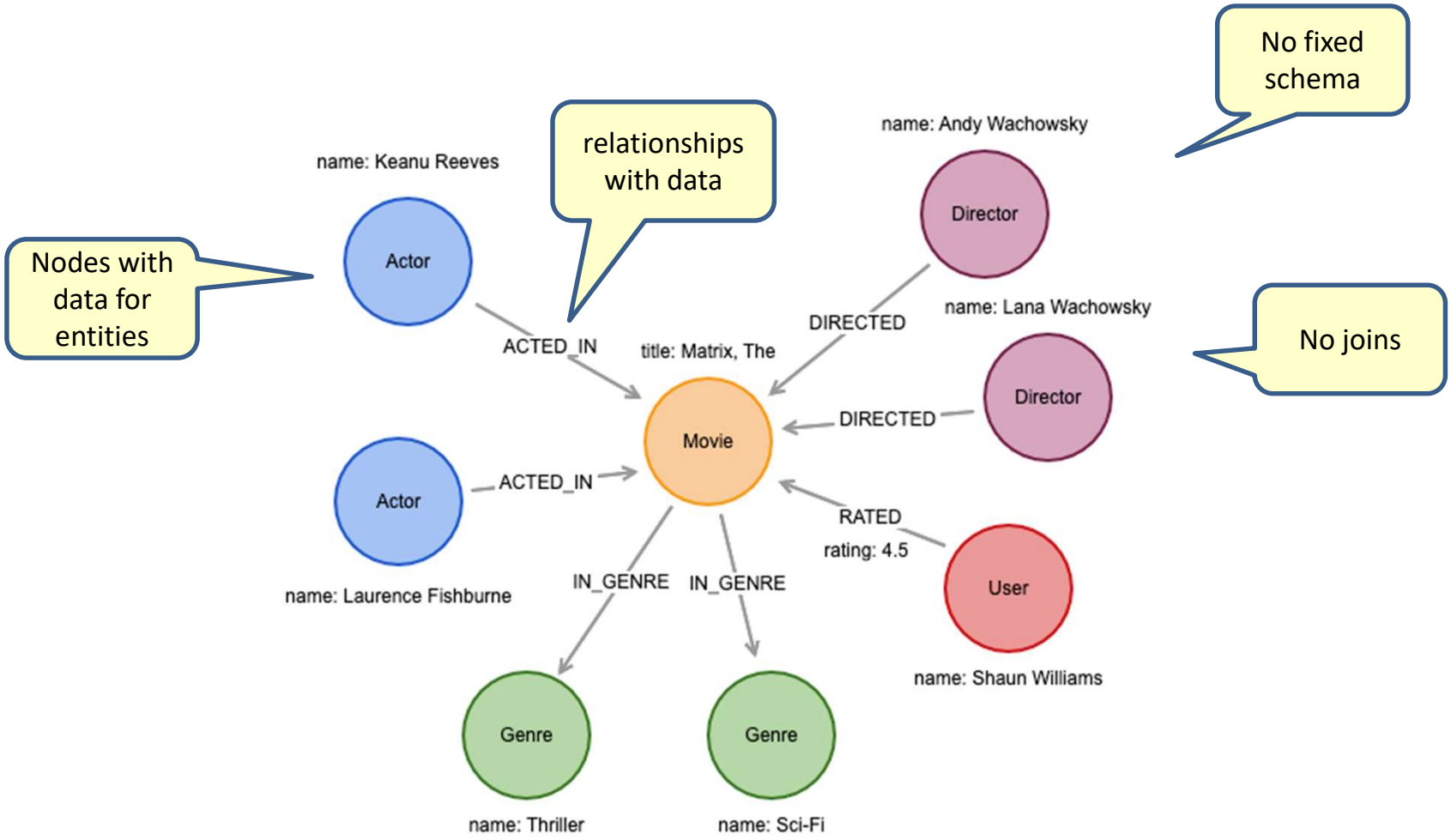
Graph database

- 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

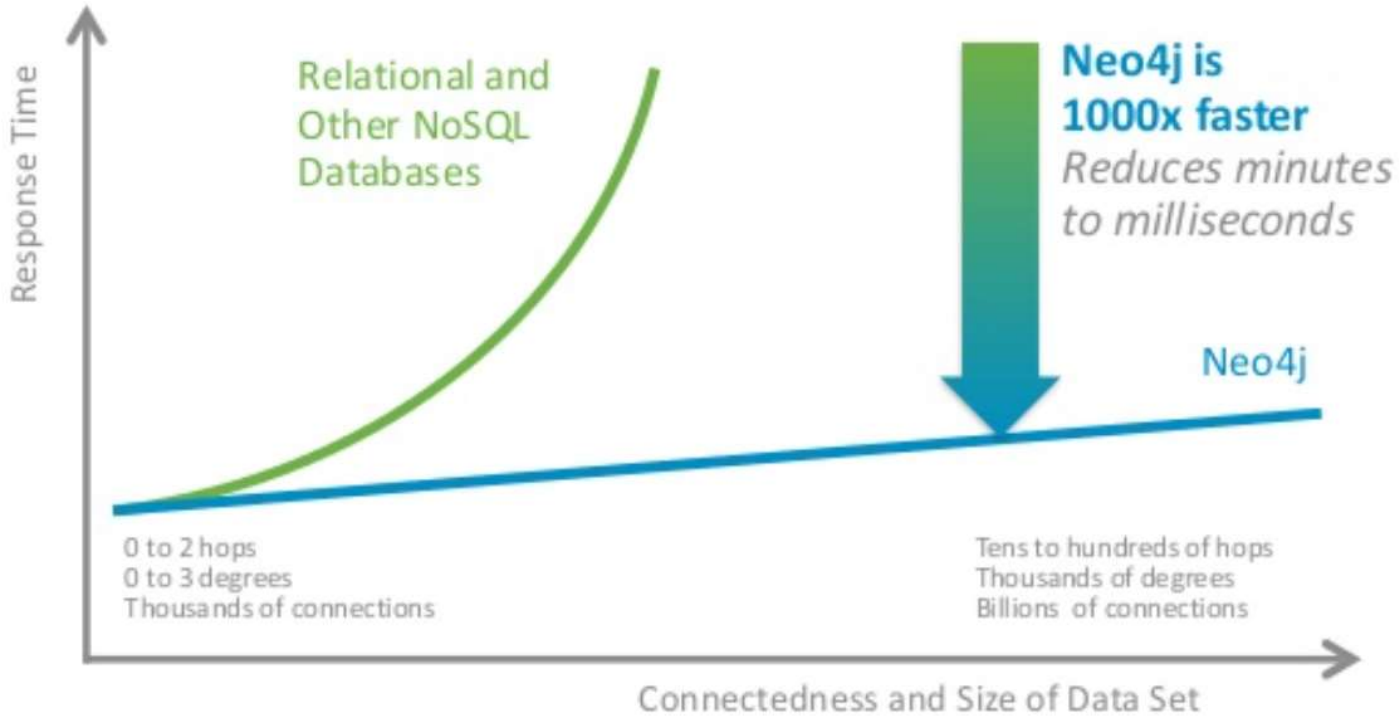
Relational database



Graph database: NEO4J



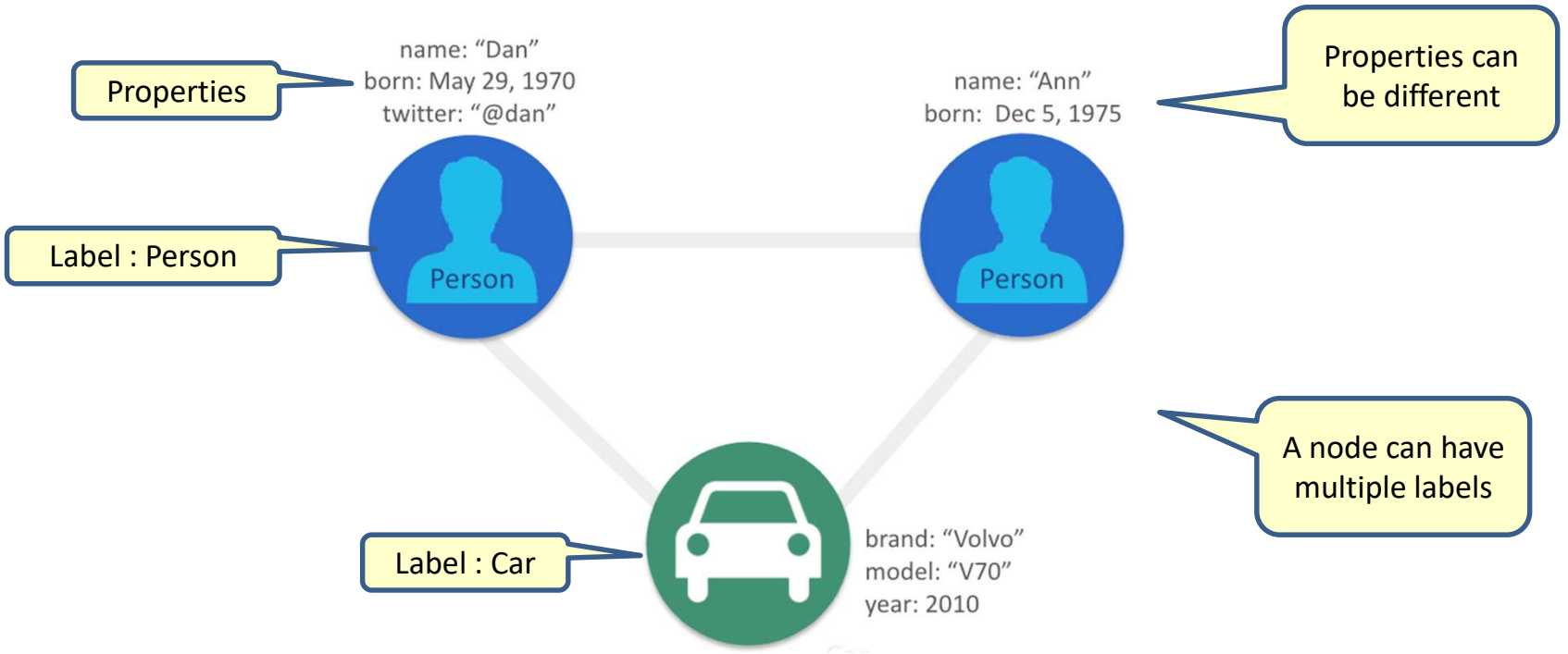
Query performance



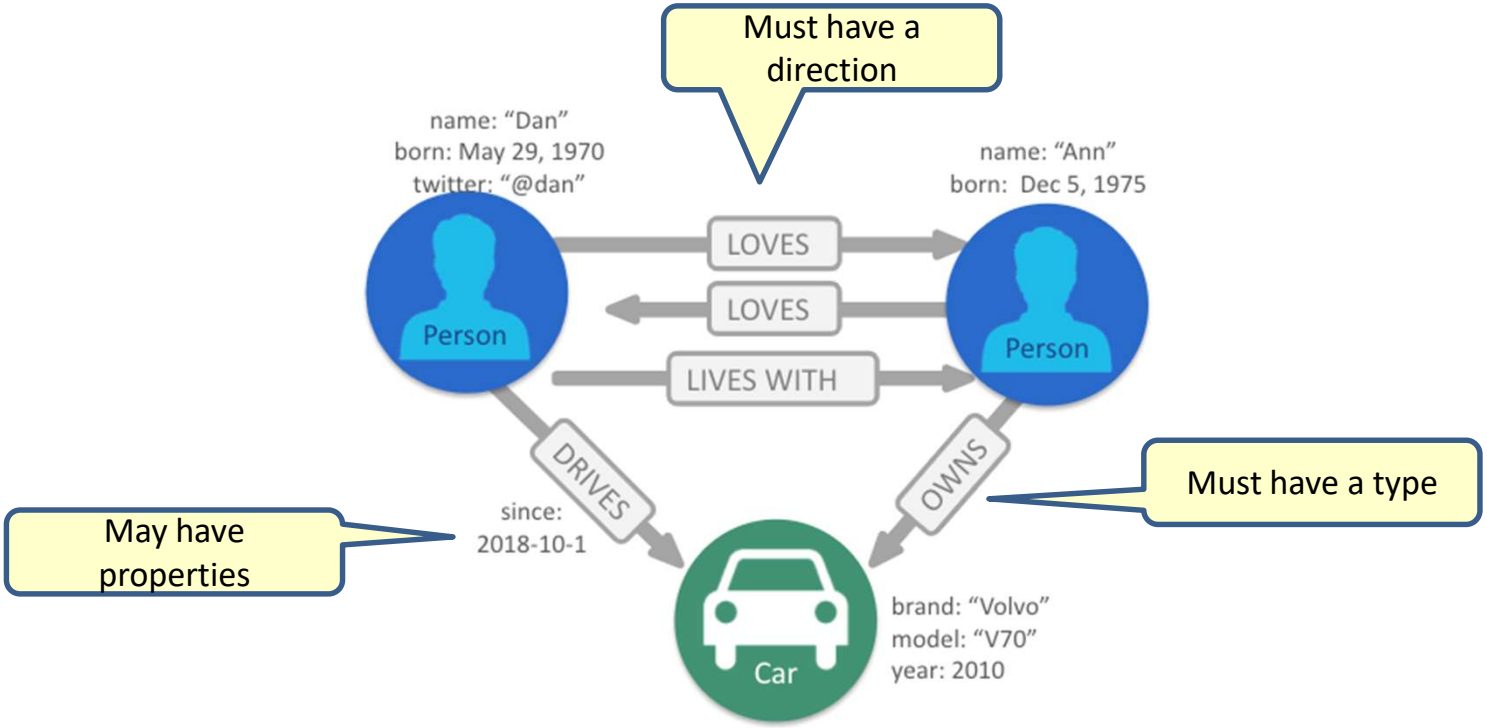
Where are graph databases used?

					
Personalization	**Fraud Detection**	**Network Operations**	**Master Data Management**	**Knowledge Graph**	**Identity and Access Management**
❑ Real-time product recommendations ❑ C360 ❑ Marketing data platform ❑ Customer Journey Analytics	❑ Anti-money laundering ❑ Insider trading ❑ Know your customer	❑ Vulnerability ❑ Impact analysis ❑ Logistics	❑ MetaData, ❑ Data lineage	❑ Artificial intelligence ❑ Machine learning ❑ Natural language processing ❑ Chatbot ❑ Human capital management	❑ Entitlement ❑ Offers ❑ Provisioning

Nodes

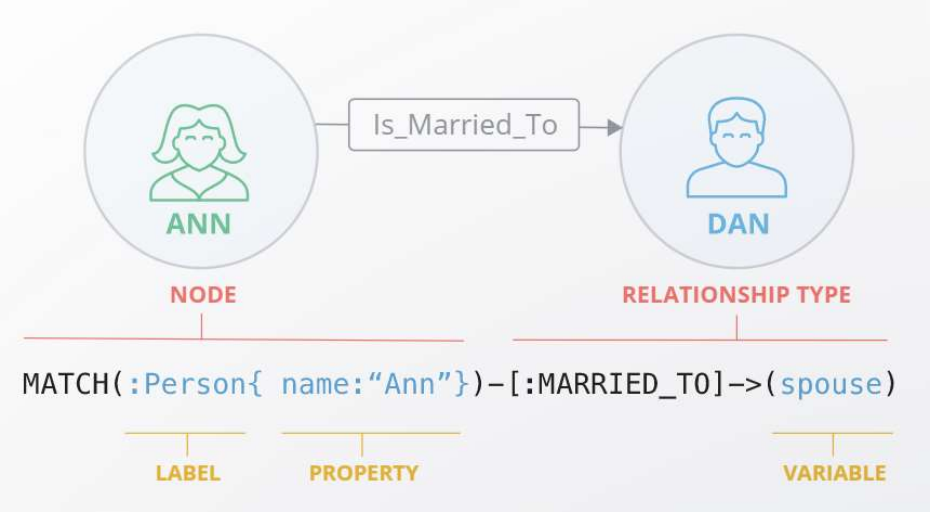


Relationships



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
```

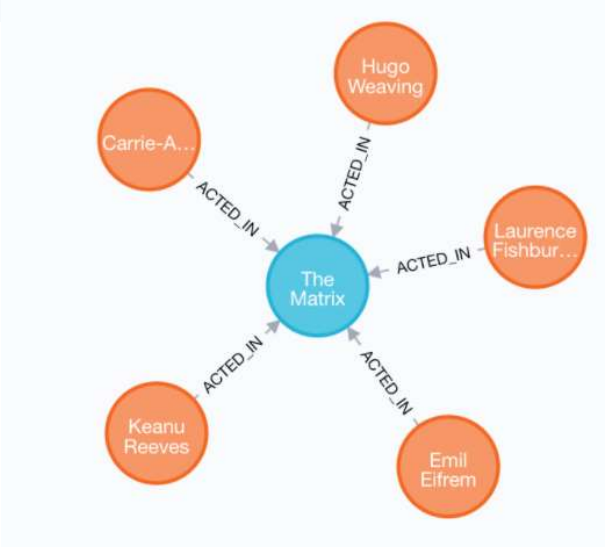
Retrieve all Person nodes born in 1970

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

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

Queries on relationships

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

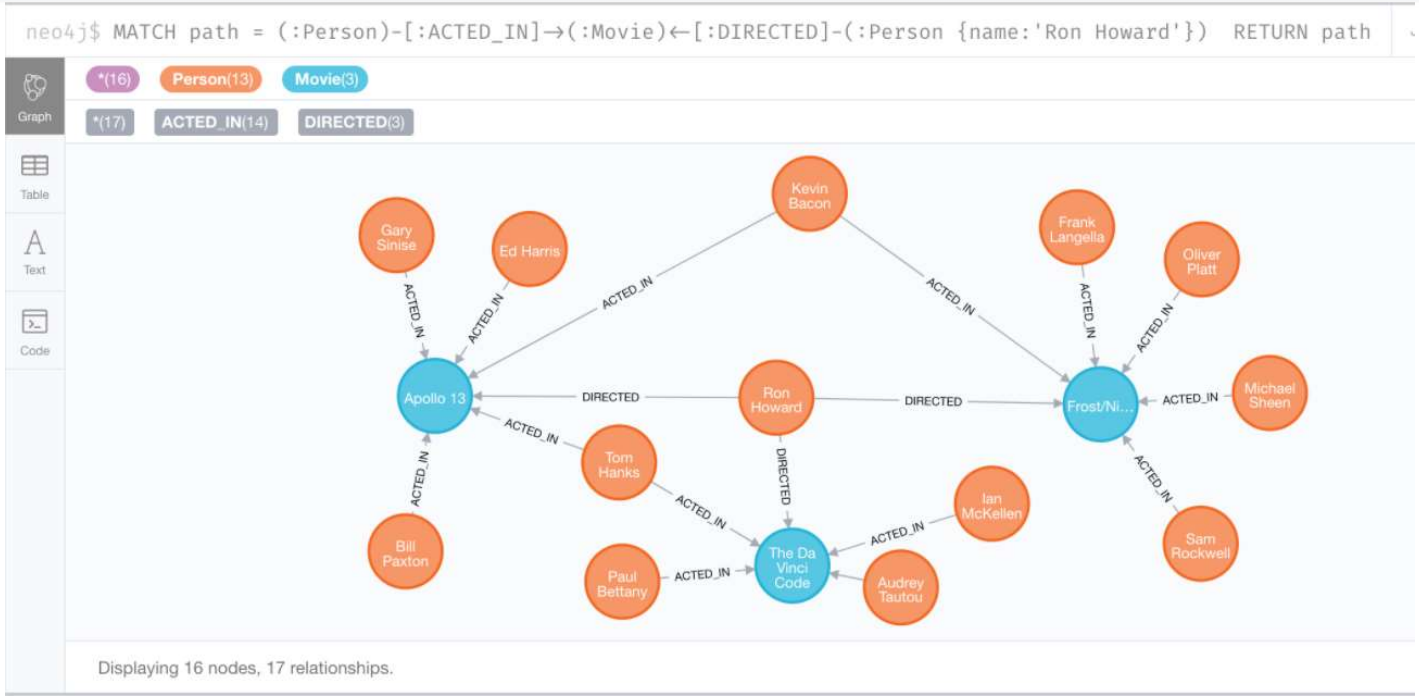


Give all actors that played in 'The Matrix'

92

Return multiple paths

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



COMPARING DATABASES

Relational database

- Advantages
  - Strict consistent transactions
  - Referential integrity
  - SQL standard
  - Well known
  - Mature
- Disadvantages
  - Not good in unstructured or semi structured data
  - Schema is fixed
  - Writes don’t scale well

When to use a relational database

- When your data is structured
- When you need strict consistent transactions
- When you do not need write scalability
- When your schema is stable
- When the size of the dataset fits on 1 node

When not to use a relational database

- When your data is unstructured or semi structured
- When you need write scalability
- When your schema changes very often
- When you need full text search
- When you need fast queries that span many tables
- When the size of the dataset does not fit on 1 node

Redis (key value store)

- Advantages
  - Super fast insert, update, delete, search of temporary data (cache)
- Disadvantages
  - You cannot query the value
  - Amount of data limited by memory size of server
  - Not transactional

When to use Redis

- Fast data access of temporary data that fits in memory
- Caching

Redis real world use cases

- Store session data
- Real time recommendation and advertising
- Caching

MongoDB (document database)

- Advantages
  - Build in scalability
  - Can handle a lot of data
  - No fixed schema
  - Fast insert, update, delete and retrieval of documents
- Disadvantages
  - Data duplication
  - Not good in handling references
  - No cross documents queries

When to use a document database

- When you are interested in document instead of relationships
  - When you don’t need cross document queries
- When your data is semi structured
  - Structure can change a lot
  - Structure can differ
- When you have a lot of data that does not fit on one node
- When you need fast data access

When not to use a document database

- When you are interested in relationships between entities
  - When you need cross entity queries
- When you need ACID transactions
- When data duplication is a problem

Cassandra (Column family database)

- Advantages
  - Very scalable
  - Fast data retrieval
  - No fixed schema
  - Can handle large data sets
- Disadvantages
  - Data duplication
  - Eventual consistency
  - Slower insert, update and delete actions

When to use a column family database

- When you need to scale your writes
- When you need cross entity queries
- When you need fast search queries
- When your data structure changes a lot
- When you have a lot of data that does not fit on one node
- When you need fast data access

When not to use a column family database

- When your queries change a lot
- When you need ACID transactions
- When data duplication is a problem

Neo4J (graph database)

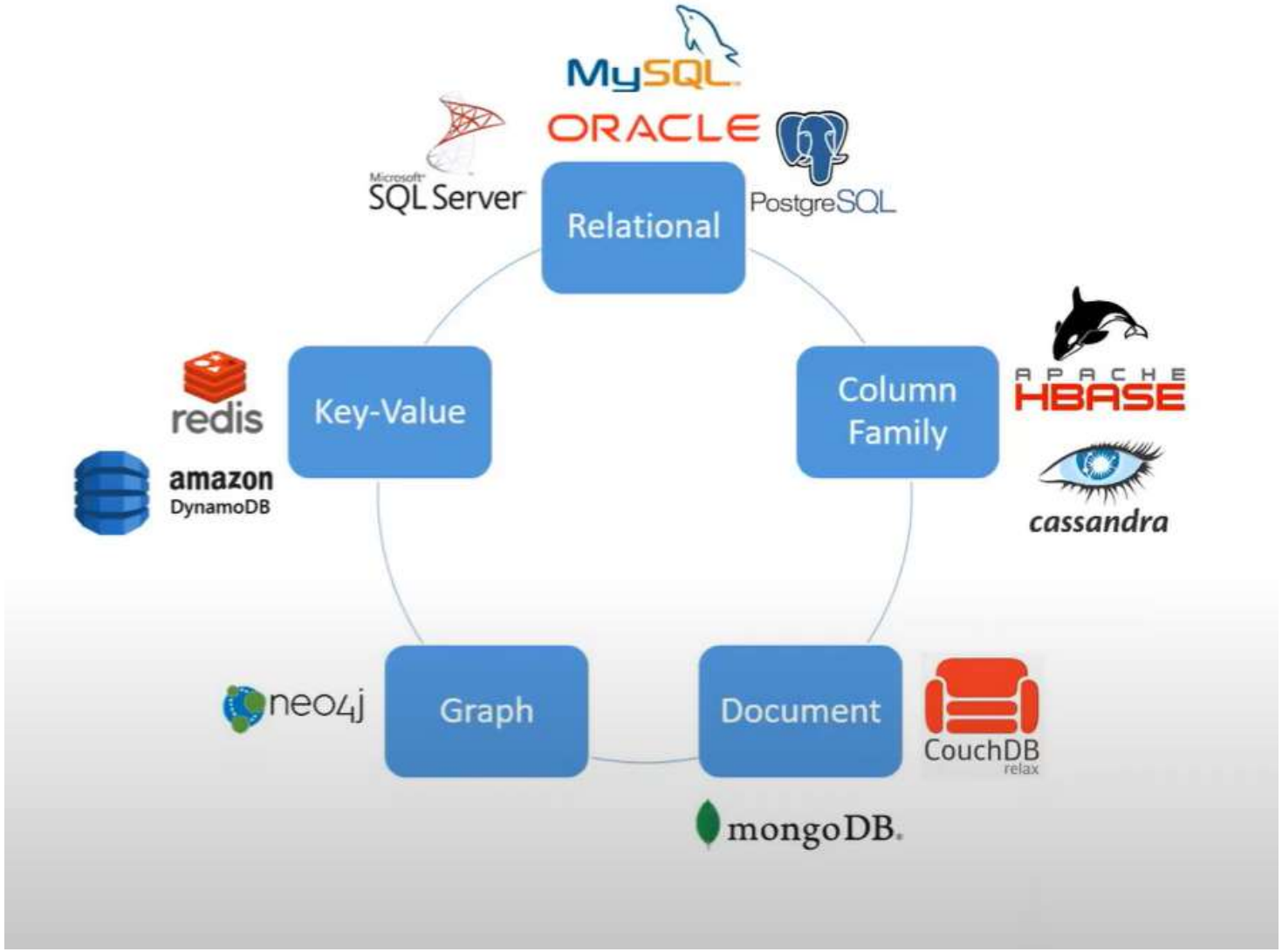
- Advantages
  - Very fast in querying deeply connected data that has many relationships
  - Database model and domain model are similar
  - Powerful and simple query language
  - Easy to add new nodes, attributes and relationships
- Disadvantages
  - Does not work well for data that is not connected

Neo4J (graph database)

- When to use a graph database?
  - When you are interested in relationships between entities
- When not to use a graph database?
  - When you are interested entities and not relationships

SUMMARY

Five different types of databases



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

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