NoSQL Databases

Principles of Database Management
Lemahieu et. Al.
Cambridge University Press, 2018

Introduction

- The NoSQL movement
- Key–value stores
- Tuple and document stores
- Column-oriented databases
- Graph-based databases
- Other NoSQL categories

The NoSQL Movement

- RDBMSs put a lot of emphasis on keeping data consistent.
  - Entire database is consistent at all times (ACID)
- Focus on consistency may hamper flexibility and scalability
- As the data volumes or number of parallel transactions increase, capacity can be increased by
  - Vertical scaling: extending storage capacity and/or CPU power of the database server
  - Horizontal scaling: multiple DBMS servers being arranged in a cluster

The NoSQL Movement

- RDBMSs are not good at extensive horizontal scaling
  - Coordination overhead because of focus on consistency
  - Rigid database schemas
- Other types of DBMSs needed for situations with massive volumes, flexible data structures, and where scalability and availability are more important ➔ NoSQL databases

The NoSQL Movement

- NoSQL databases
  - Describes databases that store and manipulate data in formats other than tabular relations, i.e., non-relational databases (NoREL)
- NoSQL databases aim at near-linear horizontal scalability by distributing data over a cluster of database nodes for the sake of performance as well as availability
- Eventual consistency: the data (and its replicas) will become consistent at some point in time after each transaction

The NoSQL Movement

	Relational Databases	NoSQL Databases
Data paradigm	Relational tables	Key–value (tuple) based Document based Column based Graph based XML, object based Others: time series, probabilistic, etc.
Distribution	Single-node and distributed	Mainly distributed
Scalability	Vertical scaling, harder to scale horizontally	Easy to scale horizontally, easy data replication
Openness	Closed and open source	Mainly open source
Schema role	Schema-driven	Mainly schema-free or flexible schema
Query language	SQL as query language	No or simple querying facilities, or special-purpose languages
Transaction mechanism	ACID: Atomicity, Consistency, Isolation, Durability	BASE: Basically Available, Soft state, Eventual consistency
Feature set	Many features (triggers, views, stored procedures, etc.)	Simple API
Data volume	Capable of handling normal-sized datasets	Capable of handling huge amounts of data and/or very high frequencies of read/write requests

Key–Value Stores

- Key–value-based database stores data as (key, value) pairs
  - Keys are unique
  - Hash map, or hash table or dictionary

Key–Value Stores

```
import java.util.HashMap;
import java.util.Map;
public class KeyValueStoreExample {
	public static void main(String... args) {
		// Keep track of age based on name
		Map<String, Integer> age_by_name = new HashMap<>();

		// Store some entries
		age_by_name.put("wilfried", 34);
		age_by_name.put("seppe", 30);
		age_by_name.put("bart", 46);
		age_by_name.put("jeanne", 19);

		// Get an entry
		int age_of_wilfried = age_by_name.get("wilfried");
		System.out.println("Wilfried's age: " + age_of_wilfried);

		// Keys are unique
		age_by_name.put("seppe", 50); // Overrides previous entry
	}
}
```

Key–Value Stores

- Keys (e.g., “bart”, “seppe”) are hashed by means of a so-called **hash function**
  - A hash function takes an arbitrary value of arbitrary size and maps it to a key with a fixed size, which is called the hash value
  - Each hash can be mapped to a space in computer memory

Key
wilfried
seppe
bart
jeanne

Hash function →

Hash	Key
01	(wilfried,34)
03	(seppe,30)
07	(bart,46)
08	(jeanne,19)

Key–Value Stores

- NoSQL databases are built with horizontal scalability support in mind
- Distribute hash table over different locations
- Assume we need to spread our hashes over three servers
  - Hash every key (“wilfried”, “seppe”) to a server identifier
  - index(hash) = mod(hash, nrServers) + 1

Key–Value Stores

Key
wilfried
seppe
bart
jeanne

Hash function

Hash	**Index** (= mod 3 + 1)
01	2
03	1
07	2
08	3

Server #1	**Entries**
03	(seppe,30)

Server #2	**Entries**
01	(wilfried,34)
07	(bart,46)

Server #3	**Entries**
08	(jeanne,19)

Sharding!

Key–Value Stores

- Example: Memcached
  - Implements a distributed memory-driven hash table (i.e., a key–value store), which is put in front of a traditional database to speed up queries by caching recently accessed objects in RAM
  - Caching solution

Key–Value Stores

```
import java.util.ArrayList;
import java.util.List;
import net.spy.memcached.AddrUtil;
import net.spy.memcached.MemcachedClient;

public class MemCachedExample {
 public static void main(String[] args) throws Exception {
 List<String> serverList = new ArrayList<String>() {
 {
 this.add("memcachedserver1.servers:11211");
 this.add("memcachedserver2.servers:11211");
 this.add("memcachedserver3.servers:11211");
 }
 };
```

Key–Value Stores

```java
MemcachedClient memcachedClient = new MemcachedClient(
AddrUtil.getAddresses(serverList));

// ADD adds an entry and does nothing if the key already exists
// Think of it as an INSERT
// The second parameter (0) indicates the expiration - 0 means no expiry
memcachedClient.add("marc", 0, 34);
memcachedClient.add("seppe", 0, 32);
memcachedClient.add("bart", 0, 66);
memcachedClient.add("jeanne", 0, 19);


// SET sets an entry regardless of whether it exists
// Think of it as an UPDATE-OR-INSERT
memcachedClient.add("marc", 0, 1111); // <- ADD will have no effect
memcachedClient.set("jeanne", 0, 12); // <- But SET will
```

Key–Value Stores

```
 // REPLACE replaces an entry and does nothing if the key does not exist
 // Think of it as an UPDATE
 memcachedClient.replace("not_existing_name", 0, 12); // <- Will have no effect
 memcachedClient.replace("jeanne", 0, 10);

 // DELETE deletes an entry, similar to an SQL DELETE statement
 memcachedClient.delete("seppe");

 // GET retrieves an entry
 Integer age_of_marc = (Integer) memcachedClient.get("marc");
 Integer age_of_short_lived = (Integer) memcachedClient.get("short_lived_name");
 Integer age_of_not_existing = (Integer) memcachedClient.get("not_existing_name");
 Integer age_of_seppe = (Integer) memcachedClient.get("seppe");
 System.out.println("Age of Marc: " + age_of_marc);
 System.out.println("Age of Seppe (deleted): " + age_of_seppe);
 System.out.println("Age of not existing name: " + age_of_not_existing);
 System.out.println("Age of short lived name (expired): " + age_of_short_lived);

 memcachedClient.shutdown();

 }
}
```

Key–Value Stores

- Request coordination
- Consistent hashing
- Replication and redundancy
- Eventual consistency
- Stabilization
- Integrity constraints and querying

Request Coordination

- In many NoSQL implementations (e.g., Cassandra, Google's BigTable, Amazon's DynamoDB), all nodes implement the same functionality and are all able to perform the role of request coordinator
- Need for membership protocol
  - Dissemination
    - Based on periodic, pairwise communication
  - Failure detection

Consistent Hashing

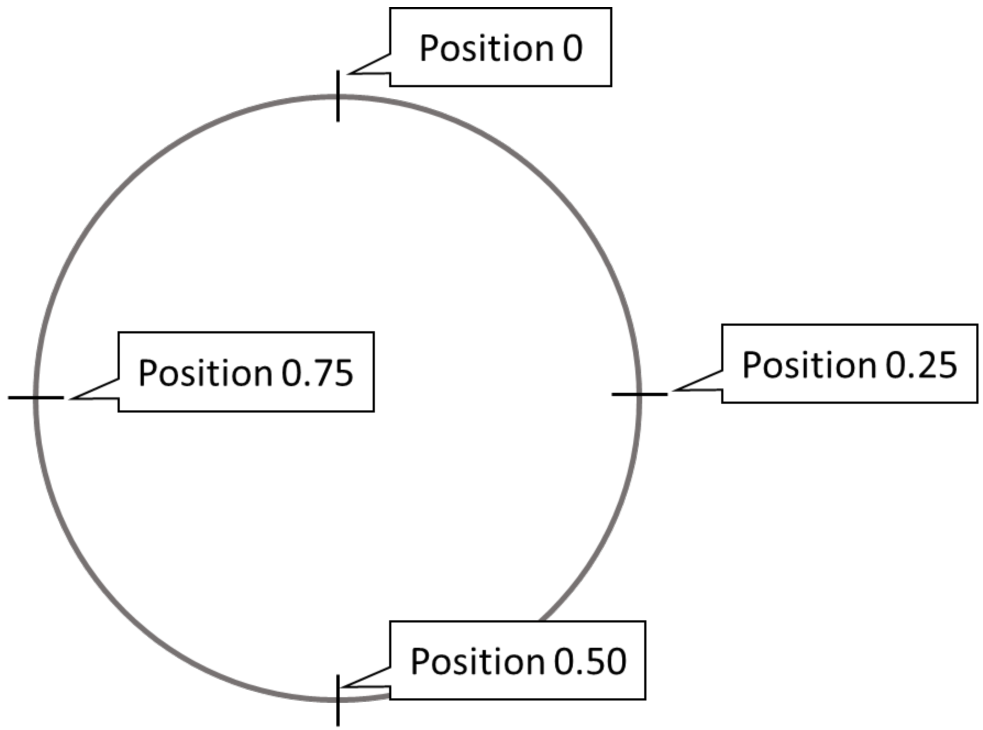
- **Consistent hashing** schemes are often used, which avoid having to remap each key to a new node when nodes are added or removed
- Suppose we have a situation in which ten keys are distributed over three servers (n = 3) with the following hash function:
  - $h(key) = key$ modulo n

Consistent Hashing

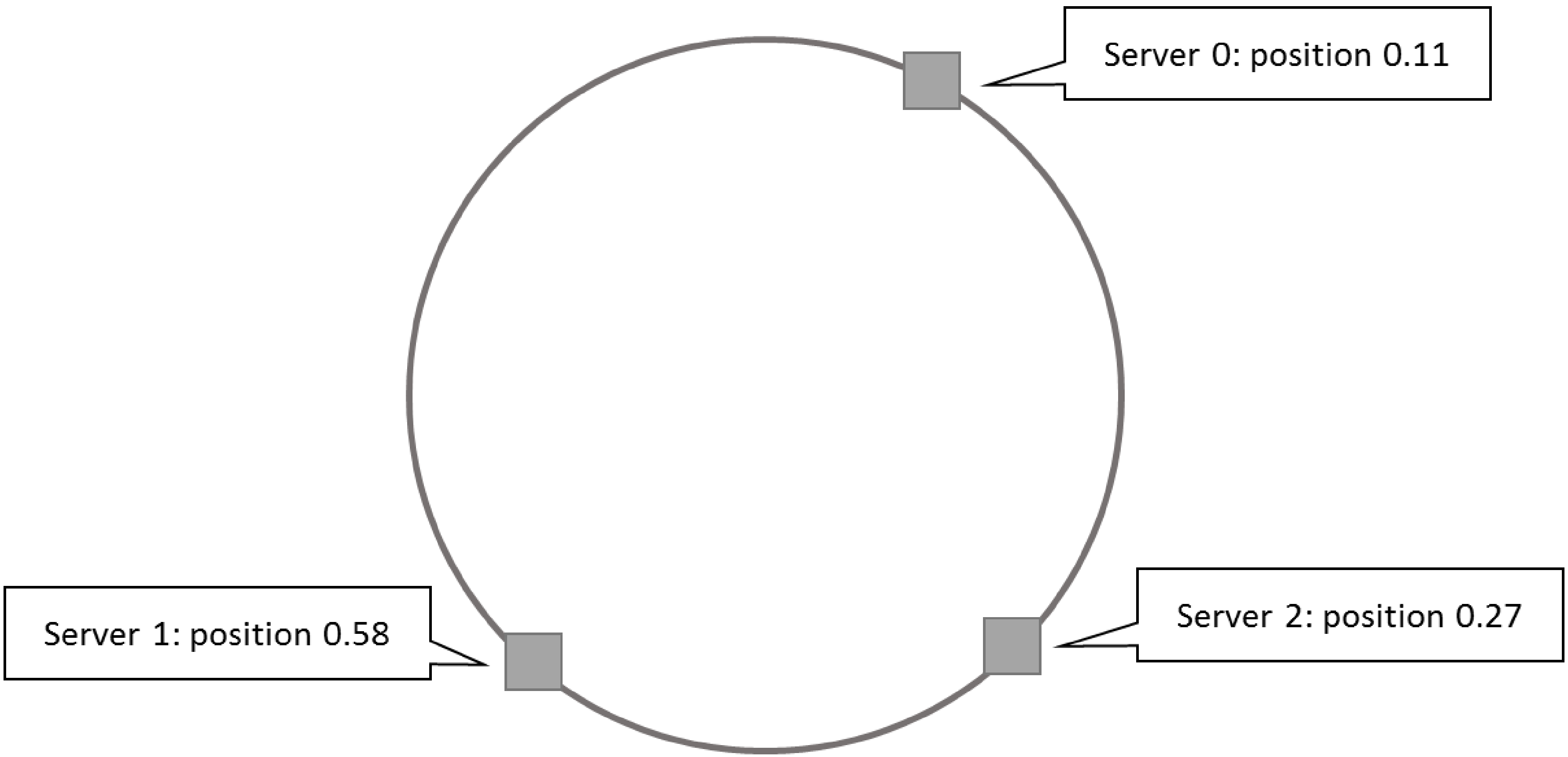
	n		
key	3	2	4
0	0	0	0
1	1	1	1
2	2	0	2
3	0	1	3
4	1	0	0
5	2	1	1
6	0	0	2
7	1	1	3
8	2	0	0
9	0	1	1

Consistent Hashing

- At the core of a consistent hashing setup is a so-called **“ring”-topology**, which is basically a representation of the number range [0,1]:

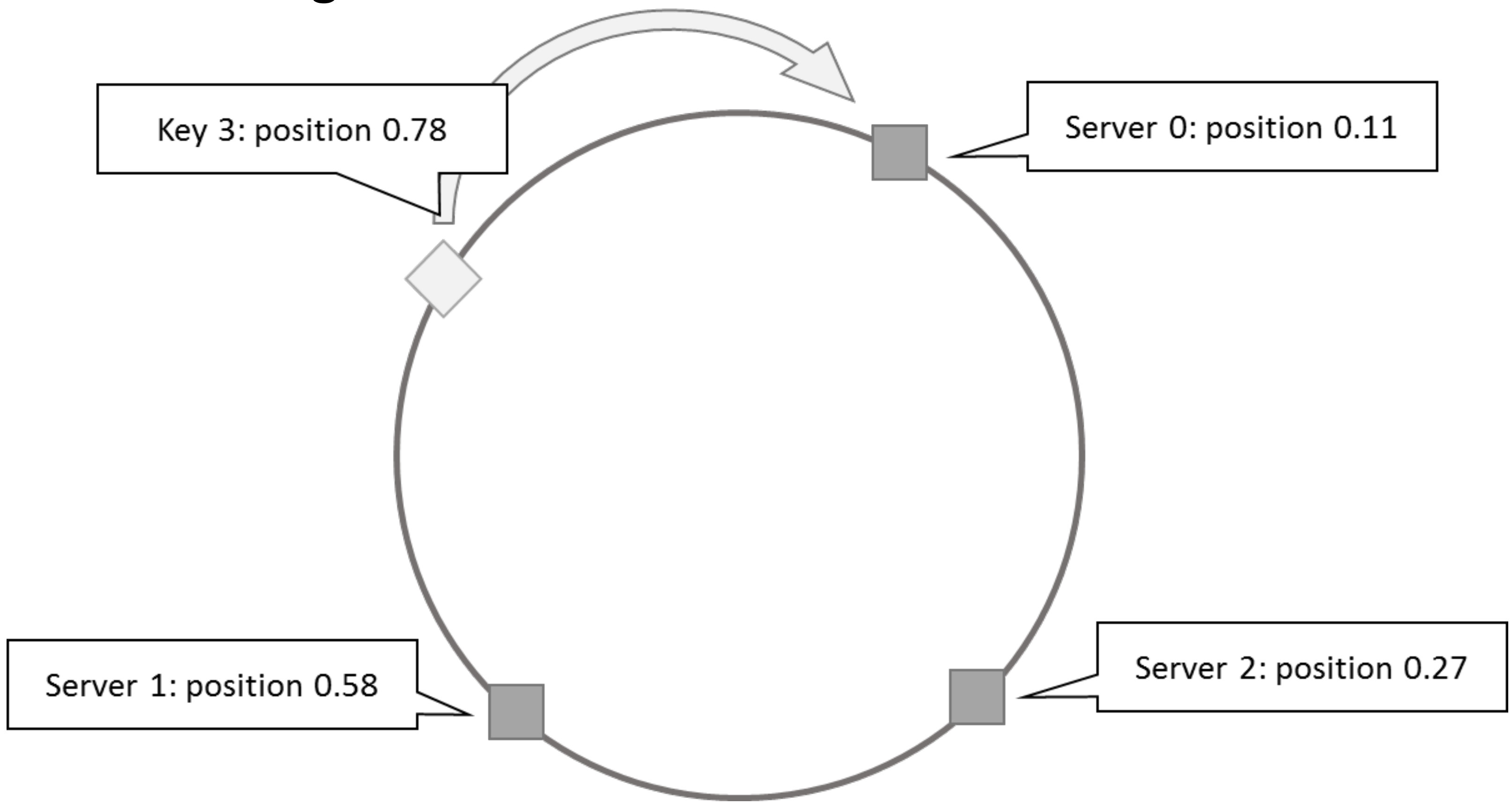


Consistent Hashing



Consistent Hashing

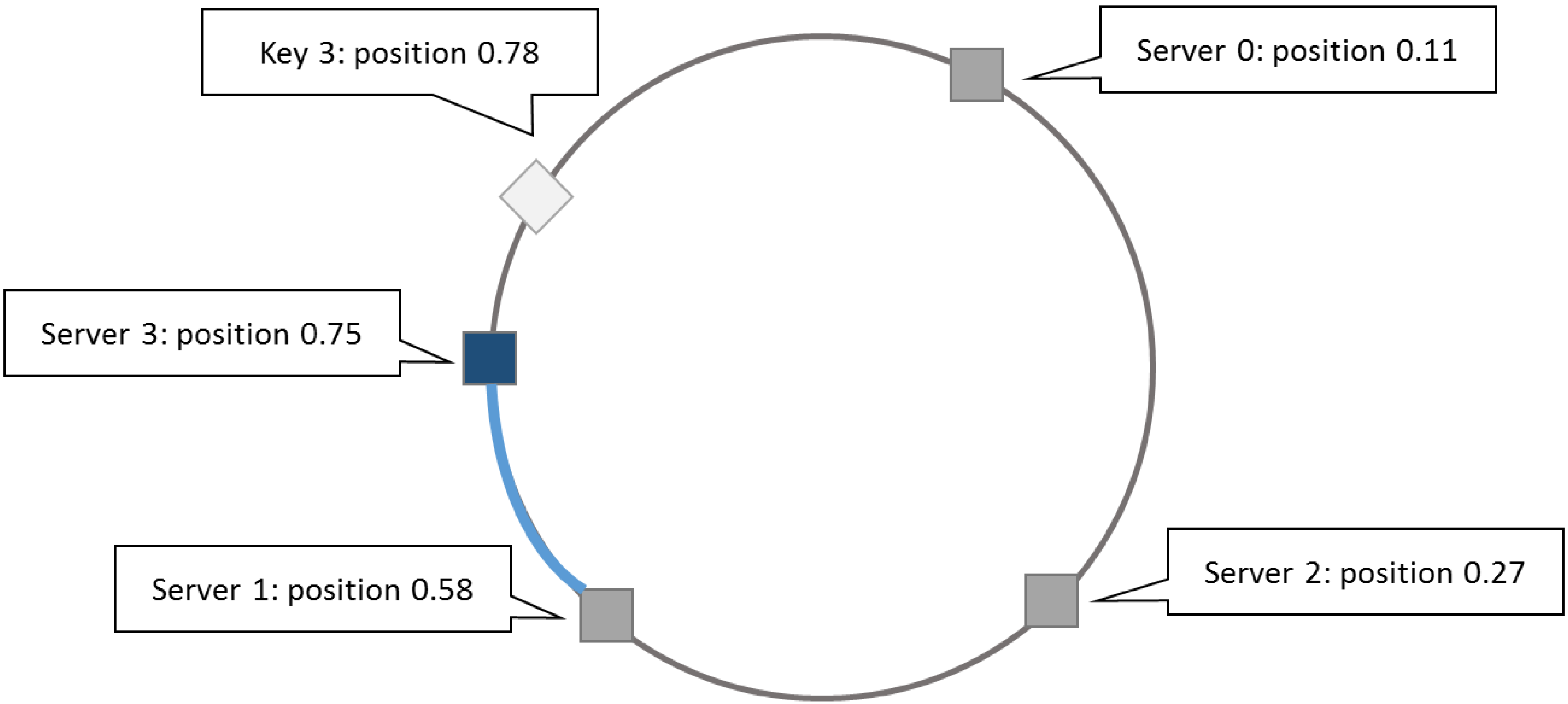
- Hash each key to a position on the ring, and store the actual key–value pair on the first server that appears clockwise of the hashed point on the ring



Consistent Hashing

- Because of the uniformity property of a “good” hash function, roughly 1/*n* of key–value pairs will end up being stored on each server
- Most of the key–value pairs will remain unaffected in the event that a machine is added or removed

Consistent Hashing



Replication and Redundancy

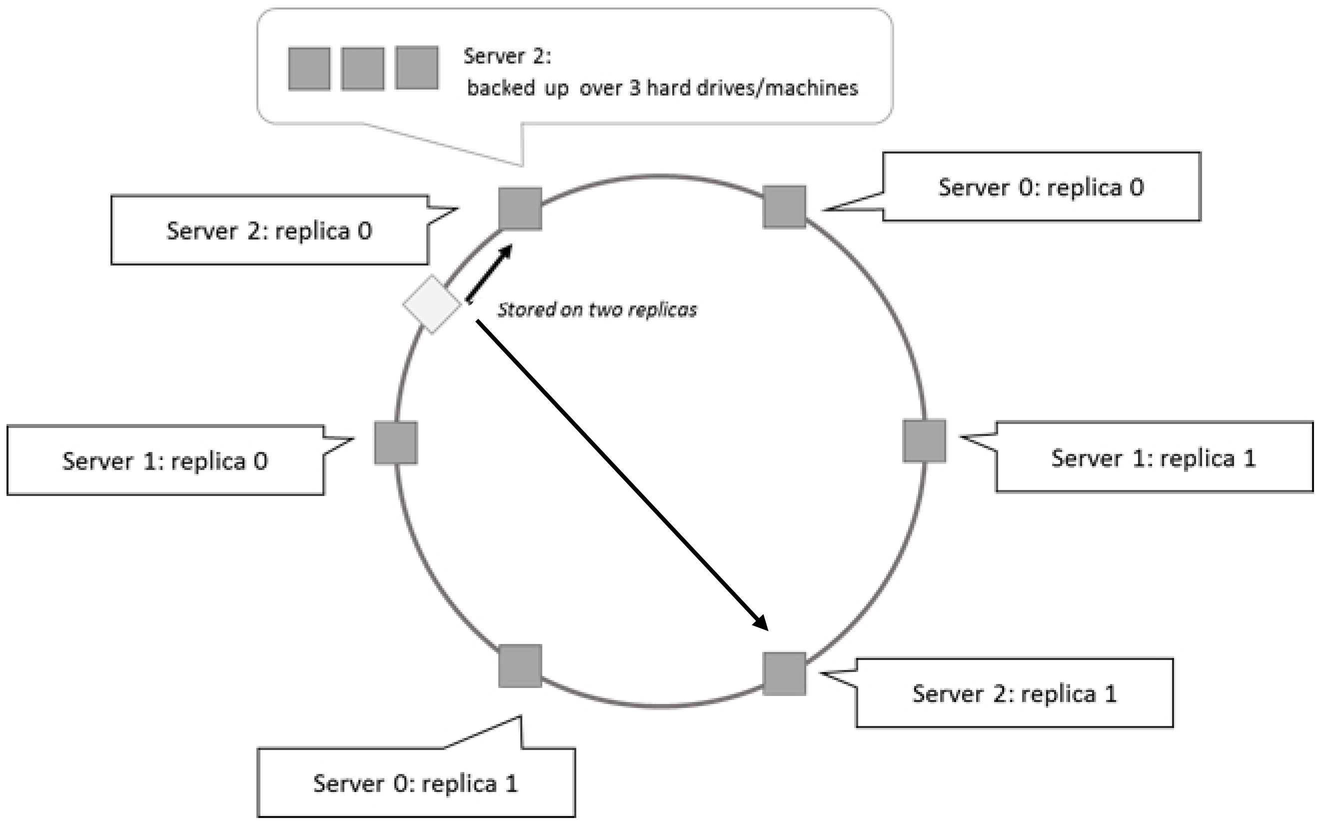
- Problems with consistent hashing:
  - If two servers end up being mapped close to one another, one of these nodes will end up with few keys to store
  - In the case that a server is added, all of the keys moved to this new node originate from just one other server
- Instead of mapping a server *s* to a single point on our ring, we map it to multiple positions, called **replicas**
- For each physical server *s*, we hence end up with *r* (the number of replicas) points on the ring
- Note: each of the replicas still represents the same physical instance (↔ redundancy)
  - Virtual nodes

Replication and Redundancy

- To handle data replication or redundancy, many vendors extend the consistent hashing mechanism so that key–value pairs are duplicated across multiple nodes
  - e.g., by storing the key–value pair on two or more nodes clockwise from the key's position on the ring

Replication and Redundancy

- It is also possible to set up a full redundancy scheme in which each node itself corresponds to multiple physical machines, each storing a fully redundant copy of the data



Eventual Consistency

- Membership protocol does not guarantee that every node is aware of every other node *at all times*
  - it will reach a consistent state over time
- State of the network might not be perfectly consistent at any moment in time, though will become eventually consistent at a future point in time
- Many NoSQL databases guarantee so-called **eventual consistency**

Eventual Consistency

- Most NoSQL databases follow the **BASE** principle
  - Basically Available, Soft state, Eventual consistency
- **CAP theorem** states that a distributed computer system cannot guarantee the following three properties at the same time:
  - Consistency (all nodes see the same data at the same time)
  - Availability (guarantees that every request receives a response indicating a success or failure result)
  - Partition tolerance (the system continues to work even if nodes go down or are added)

Eventual Consistency

- Most NoSQL databases sacrifice the consistency part of CAP in their setup, instead striving for eventual consistency
- The full BASE acronym stands for:
  - Basically Available: NoSQL databases adhere to the availability guarantee of the CAP theorem
  - Soft state: the system can change over time, even without receiving input
  - Eventual consistency: the system will become consistent over time

Stabilization

- The operation which repartitions hashes over nodes in case nodes are added or removed is called **stabilization**
- If a consistent hashing scheme is being applied, the number of fluctuations in the hash–node mappings will be minimized.

Integrity Constraints and Querying

- Key–value stores represent a very diverse gamut of systems
- Full-blown DBMSs versus caches
- Only limited query facilities are offered
  - e.g. put and set
- Limited to no means to enforce structural constraints
  - DBMS remains agnostic to the internal structure
- No relationships, referential integrity constraints, or database schema can be defined

Tuple and Document Stores

- A **tuple store** is similar to a key–value store, with the difference that it does not store pairwise combinations of a key and a value, but instead stores a unique key together with a vector of data
- Example:
  - `marc -> ("Marc", "McLast Name", 25, "Germany")`
- No requirement to have the same length or semantic ordering (schema-less!)

Tuple and Document Stores

- Various NoSQL implementations do, however, permit organizing entries in semantical groups (aka collections or tables)
- Examples:
  - `Person:marc -> ("Marc", "McLast Name", 25, "Germany")`
  - `Person:harry -> ("Harry", "Smith", 29, "Belgium")`

Tuple and Document Stores

- **Document stores** store a collection of attributes that are labeled and unordered, representing items that are semi-structured
- Example:

```
{
 Title = "Harry Potter"
 ISBN = "111-1111111111"
 Authors = [ "J.K. Rowling" ]
 Price = 32
 Dimensions = "8.5 x 11.0 x 0.5"
 PageCount = 234
 Genre = "Fantasy"
}
```

Tuple and Document Stores

- Most modern NoSQL databases choose to represent documents using JSON

```
{
    "title": "Harry Potter",
    "authors": ["J.K. Rowling", "R.J. Kowling"],
    "price": 32.00,
    "genres": ["fantasy"],
    "dimensions": {
        "width": 8.5,
        "height": 11.0,
        "depth": 0.5
    },
    "pages": 234,
    "in_publication": true,
    "subtitle": null
}
```

Tuple and Document Stores

- Items with keys
- Filters and queries
- Complex queries and aggregation with MapReduce
- SQL after all …

Items with Keys

- Most NoSQL document stores will allow you to store items in tables (collections) in a schema-less manner, but will enforce that a primary key be specified
  - e.g. Amazon’s DynamoDB, MongoDB (_id)
- A primary key will be used as a partitioning key to create a hash and determine where the data will be stored

Filters and Queries

```
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoDatabase;
import java.util.ArrayList;
import static com.mongodb.client.model.Filters.*;
import static java.util.Arrays.asList;

public class MongoDBExample {
 public static void main(String... args) {
 MongoClient mongoClient = new MongoClient();
 MongoDatabase db = mongoClient.getDatabase("test");

 // Delete all books first
 db.getCollection("books").deleteMany(new Document());
// Add some books
 db.getCollection("books").insertMany(new ArrayList<Document>() {{
 add(getBookDocument("My First Book", "Wilfried", "Lemahieu", 12, new String[]{"drama"}));
 add(getBookDocument("My Second Book", "Seppe", “vanden Broucke", 437, new String[]{"fantasy", "thriller"}));
 add(getBookDocument("My Third Book", "Seppe", “vanden Broucke", 200, new String[]{"educational"}));
 add(getBookDocument("Java Programming", "Bart", "Baesens", 100, new String[]{"educational"}));
 }});
```

Filters and Queries

```
// Perform query
FindIterable<Document> result = db.getCollection("books").find(
and( eq("author.last_name", “vanden Broucke"),
eq("genres", "thriller"),
gt("nrPages", 100)));

for (Document r : result) {
System.out.println(r.toString());
// Increase the number of pages:
db.getCollection("books").updateOne(
new Document("_id", r.get("_id")),
new Document("$set",
new Document("nrPages", r.getInteger("nrPages") + 100)));
}
mongoClient.close();}

        public static Document getBookDocument(String title,
         String authorFirst, String authorLast,
         int nrPages, String[] genres) {
         return new Document("author", new Document()
         .append("first_name", authorFirst)
         .append("last_name", authorLast))
         .append("title", title)
         .append("nrPages", nrPages)
         .append("genres", asList(genres));}}
```

Filters and Queries

```
Document{{_id=567ef62bc0c3081f4c04b16c,
author=Document{{first_name=Seppe, last_name=vanden Broucke}},
title=My Second Book, nrPages=437, genres=[fantasy, thriller]}}
```

Filters and Queries

```
// Perform aggregation query
AggregateIterable<Document> result = db.getCollection("books")
         .aggregate(asList(
                  new Document("$group",
                           new Document("_id", "$author.last_name")
                                    .append("page_sum", new Document("$sum",
"$nrPages")))));

for (Document r : result) {
 System.out.println(r.toString());
}
```

```
Document{{_id=Lemahieu, page_sum=12}}
Document{{_id=Vanden Broucke, page_sum=637}}
Document{{_id=Baesens, page_sum=100}}
```

Filters and Queries

- Queries can still be slow because every filter (such as “author.last_name = Baesens”) entails a complete collection or table scan
- Most document stores can define a variety of indexes
  - unique and non-unique indexes
  - compound indexes
  - geospatial indexes
  - text-based indexes

Complex Queries and Aggregation with MapReduce

- Document stores do not support relations
- **First approach**: embedded documents

```
{
        "title": "Databases for Beginners",
        "authors": ["J.K. Sequel", "John Smith"],
        "pages": 234
}
```

BUT: Data duplication!

```
{
        "title": "Databases for Beginners",
        "authors": [
 {"first_name": "Jay Kay", "last_name": "Sequel", "age": 54},
 {"first_name": "John", "last_name": "Smith", "age": 32}
],
        "pages": 234
}
```

Complex Queries and Aggregation with MapReduce

- **Second approach**: create two collections

```
book collection:
{
     "title": "Databases for Beginners",
     "authors": ["Jay Kay Rowling", "John Smith"],
     "pages": 234
}

authors collection:
{
     "_id": "Jay Kay Rowling",
     "age": 54
}
```

BUT: Need to resolve complex relational queries in application code!

Complex Queries and Aggregation with MapReduce

- **Third Approach**: MapReduce
  - A map-reduce pipeline starts from a series of key–value pairs (k1,v1) and maps each pair to one or more output pairs
  - The output entries are shuffled and distributed so that all output entries belonging to the same key are assigned to the same worker (e.g., physical machines)
  - Workers then apply a reduce function to each group of key–value pairs having the same key, producing a new list of values per output key
  - The resulting, final outputs are then (optionally) sorted per key k2 to produce the final outcome

Complex Queries and Aggregation with MapReduce

- Example: get a summed count of pages for books per genre
- Create a list of input key–value pairs

k1	**v1**
1	{genre: education, nrPages: 120}
2	{genre: thriller, nrPages: 100}
3	{genre: fantasy, nrPages: 20}
…	…

- Map function is a simple conversion to a genre–nrPages key–value pair

```
function map(k1, v1)
   emit output record (v1.genre, v1.nrPages)
end function
```

Complex Queries and Aggregation with MapReduce

- Workers have produced the following three output lists, with the keys corresponding to genres

Worker 1	
k2	**v2**
education	120
thriller	100
fantasy	20

Worker 2	
k2	**v2**
drama	500
education	200

Worker 3	
k2	**v2**
education	20
fantasy	10

- A working operation will be started per unique key k2, for which its associated list of values will be reduced
  - e.g., (education,[120,200,20]) will be reduced to its sum, 340

```
function reduce(k2, v2_list)
    emit output record (k2, sum(v2_list))
end function
```

Complex Queries and Aggregation with MapReduce

- Final output looks like this:

k2	**v3**
education	340
thriller	100
drama	500
fantasy	30

- Can be sorted based on k2 or v3

Complex Queries and Aggregation with MapReduce

- Suppose we would now like to retrieve an average page count per book for each genre
- Reduce function becomes

```
function reduce(k2, v2_list)
    emit output record (k2, sum(v2_list) / length(v2_list))
end function
```

- After mapping the input list, workers produce the following three output lists

Worker 1	
k2	**v2**
education	120
thriller	100
fantasy	20

Worker 2	
k2	**v2**
drama	500
education	200

Worker 3	
k2	**v2**
education	20
fantasy	10

Complex Queries and Aggregation with MapReduce

- Average as follows

k2	v3
education	(20 + 50 + 50 + 100 + 100 + 20) / 6 = 56.67
thriller	100 / 1 = 100.00
drama	(100 + 200 + 200) / 3 = 166.67
fantasy	(20 + 10) / 3 = 10.00

- Note: reduce operation can happen more than once, and can already start before all mapping operations have finished!
  – Need to ensure that results are correct by possibly rewriting map and reduce functions!

Complex Queries and Aggregation with MapReduce

- Example: count the number of occurrences per word in a document

```
function map(document_name, document_text)
    for each word in document_text do
            emit output record (word, 1)
    repeat
end function

function reduce(word, partial_counts)
    emit output record (word, sum(partial_counts))
end function
```

Complex Queries and Aggregation with MapReduce

- Example: return the average number of pages per genre, but now taking into account that books can have more than one genre associated with them (in MongoDB)

Complex Queries and Aggregation with MapReduce

```java
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.client.MongoDatabase;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import static java.util.Arrays.asList;

public class MongoDBAggregationExample {
        public static Random r = new Random();

 public static void main(String... args) {
 MongoClient mongoClient = new MongoClient();
 MongoDatabase db = mongoClient.getDatabase("test");

 setupDatabase(db);
 for (Document r : db.getCollection("books").find())
        System.out.println(r);

 mongoClient.close();}
```

Complex Queries and Aggregation with MapReduce

```
public static void setupDatabase(MongoDatabase db) {
        db.getCollection("books").deleteMany(new Document());

        String[] possibleGenres = new String[] {
                          "drama", "thriller", "romance", "detective",
                          "action", "educational", "humor", "fantasy" };

for (int i = 0; i < 100; i++) {
         db.getCollection("books").insertOne(
                          new Document("_id", i)
        .append("nrPages", r.nextInt(900) + 100)
        .append("genres",
getRandom(asList(possibleGenres), r.nextInt(3) + 1)));
}
}
```

Complex Queries and Aggregation with MapReduce

```
public static List<String> getRandom(List<String> els, int number) {
        List<String> selected = new ArrayList<>();
        List<String> remaining = new ArrayList<>(els);
        for (int i = 0; i < number; i++) {
                int s = r.nextInt(remaining.size());
                selected.add(remaining.get(s));
                remaining.remove(s);
        }
        return selected;
 }
}
```

```
Document{{_id=0, nrPages=188, genres=[action, detective, romance]}}
Document{{_id=1, nrPages=976, genres=[romance, detective, humor]}}
Document{{_id=2, nrPages=652, genres=[thriller, fantasy, action]}}
Document{{_id=3, nrPages=590, genres=[fantasy]}}
Document{{_id=4, nrPages=703, genres=[educational, drama, thriller]}}
Document{{_id=5, nrPages=913, genres=[detective]}}
…
```

Complex Queries and Aggregation with MapReduce

- Manual construction of the aggregation query looks as follows:

```
public static void reportAggregate(MongoDatabase db) {
	Map<String, List<Integer>> counts = new HashMap<>();
	for (Document r : db.getCollection("books").find()) {
		for (Object genre : r.get("genres", List.class)) {
			if (!counts.containsKey(genre.toString()))
				counts.put(genre.toString(), new ArrayList<Integer>());
			counts.get(genre.toString()).add(r.getInteger("nrPages"));
		}
	}
	for (Entry<String, List<Integer>> entry : counts.entrySet()) {
		System.out.println(entry.getKey() + " --> AVG = " +
				sum(entry.getValue()) / (double) entry.getValue().size());
	}
}

private static int sum(List<Integer> value) {
	int sum = 0;
	for (int i : value) sum += i;
	return sum;
}
```

```
romance --> AVG = 497.39285714285717
drama --> AVG = 536.88
detective --> AVG = 597.1724137931035
humor --> AVG = 603.5357142857143
fantasy --> AVG = 540.0434782608696
educational --> AVG = 536.1739130434783
action --> AVG = 398.9032258064516
thriller --> AVG = 513.5862068965517
```

Complex Queries and Aggregation with MapReduce

- If the list of genres is known beforehand, we can optimize by performing the aggregation per genre directly in MongoDB itself:

```
public static void reportAggregate(MongoDatabase db) {
        String[] possibleGenres = new String[] {
                              "drama", "thriller", "romance", "detective",
                              "action", "educational", "humor", "fantasy" };

        for (String genre : possibleGenres) {
                   AggregateIterable<Document> iterable =
                    db.getCollection("books").aggregate(asList(
                              new Document("$match", new Document("genres", genre)),
                              new Document("$group", new Document("_id", genre)
                                         .append("average", new Document("$avg", "$nrPages")))));

                   for (Document r : iterable) {
                              System.out.println(r);
                   }
        }
}
```

```
Document{{_id=drama, average=536.88}}
Document{{_id=thriller, average=513.5862068965517}}
Document{{_id=romance, average=497.39285714285717}}
Document{{_id=detective, average=597.1724137931035}}
Document{{_id=action, average=398.9032258064516}}
...
```

Complex Queries and Aggregation with MapReduce

- Assume now that we have millions of books in our database and we do not know the number of genres beforehand ➔ use MapReduce
- Map in MongoDB:

```
function() {
 // No arguments, use "this" to refer to the
 // local document item being processed
 emit(key, value);
}
```

- Reduce in MongoDB:

```
function(key, values) {
 return result;
}
```

Complex Queries and Aggregation with MapReduce

- **Map function**

```
function() {
    var nrPages = this.nrPages;
    this.genres.forEach(function(genre) {
            emit(genre, {average: nrPages, count: 1});
    });
}
```

- **Reduce function**

```
function(genre, values) {
    var s = 0;
    var newc = 0;
    values.forEach(function(curAvg) {
            s += curAvg.average * curAvg.count;
            newc += curAvg.count;
    });
    return {average: (s / newc), count: newc};
}
```

Complex Queries and Aggregation with MapReduce

```
public static void reportAggregate(MongoDatabase db) {
         String map = "function() { " +
                            " var nrPages = this.nrPages; " +
                            " this.genres.forEach(function(genre) { " +
                            " emit(genre, {average: nrPages, count: 1}); " +
                            " }); " +
                            "} ";
         String reduce = "function(genre, values) { " +
                            " var s = 0; var newc = 0; " +
                            " values.forEach(function(curAvg) { " +
                            " s += curAvg.average * curAvg.count; " +
                            " newc += curAvg.count; " +
                            " }); " +
                            " return {average: (s / newc), count: newc}; " +
                            "} ";

         MapReduceIterable<Document> result = db.getCollection("books")
 .mapReduce(map, reduce);
         for (Document r : result)
                  System.out.println(r);}
```

Complex Queries and Aggregation with MapReduce

Document{{_id=action, value=Document{{average=398.9032258064516, count=31.0}}}}

Document{{_id=detective, value=Document{{average=597.1724137931035, count=29.0}}}}

Document{{_id=drama, value=Document{{average=536.88, count=25.0}}}}

Document{{_id=educational, value=Document{{average=536.1739130434783, count=23.0}}}}

Document{{_id=fantasy, value=Document{{average=540.0434782608696, count=23.0}}}}

Document{{_id=humor, value=Document{{average=603.5357142857143, count=28.0}}}}

Document{{_id=romance, value=Document{{average=497.39285714285717, count=28.0}}}}

Document{{_id=thriller, value=Document{{average=513.5862068965517, count=29.0}}}}

SQL After All

- GROUP BY-style SQL queries are convertible to an equivalent map–reduce pipeline
- Many document store implementations express queries using an SQL interface
- Couchbase also allows defining foreign keys and performing join operations

```
SELECT books.title, books.genres,
authors.name
FROM books
JOIN authors ON KEYS books.authorId
```

SQL After All

- Many RDBMS vendors start implementing NoSQL by the following:
  - Focusing on horizontal scalability and distributed querying
  - Dropping schema requirements
  - Support for nested data types or allowing storing JSON directly in tables
  - Support for map–reduce operations
  - Support for special data types, such as geospatial data

Column-Oriented Databases

- A **column-oriented DBMS** is a database management system that stores data tables as sections of columns of data
- Useful if:
  - Aggregates are regularly computed over large numbers of similar data items
  - Data are sparse, i.e., columns with many null values
- Can also be an RDBMS, key–value, or document store

Column-Oriented Databases

- Example

Id	Genre	Title	Price	Audiobook price
1	fantasy	My first book	20	30
2	education	Beginners guide	10	null
3	education	SQL strikes back	40	null
4	fantasy	The rise of SQL	10	null

- Row-based databases are not efficient at performing operations that apply to the entire dataset
  - Need indexes which add overhead

Column-Oriented Databases

- In a column-oriented database, all values of a column are placed together on disk

  Genre: fantasy:1,4 education:2,3

  Title: My first book:1 Beginners guide:2 SQL strikes back:3 The rise of SQL:4

  Price: 20:1 10:2,4 40:3

  Audiobook price: 30:1

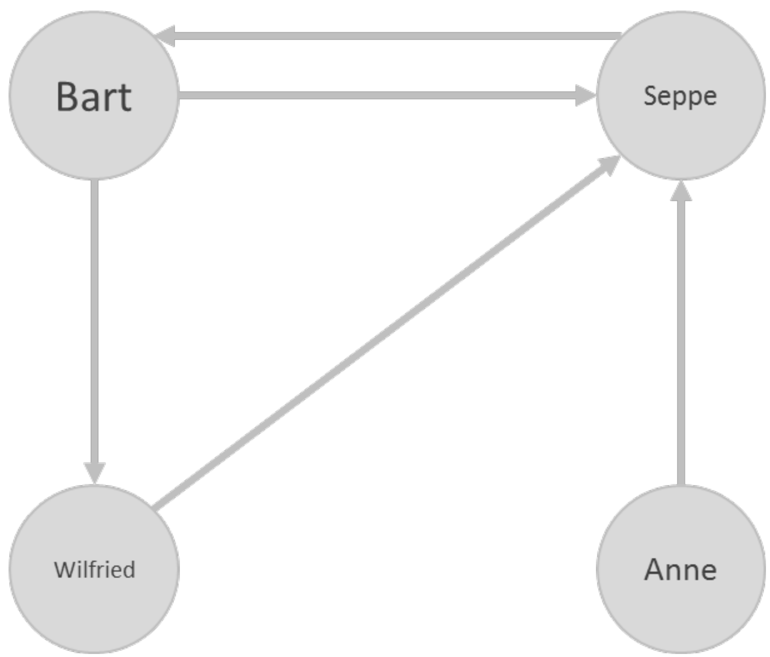
- A column matches the structure of a normal index in a row-based system
- Operations such as find all records with price equal to 10 can now be executed directly
- Null values do not take up storage space anymore

Column-Oriented Databases

- Disadvantages
  – Retrieving all attributes pertaining to a single entity becomes less efficient
  – Join operations will be slowed down
- Examples
  – Google BigTable, Cassandra, HBase, and Parquet

Graph-Based Databases

- **Graph databases** apply graph theory to the storage of information of records
- Graphs consist of **nodes** and **edges**



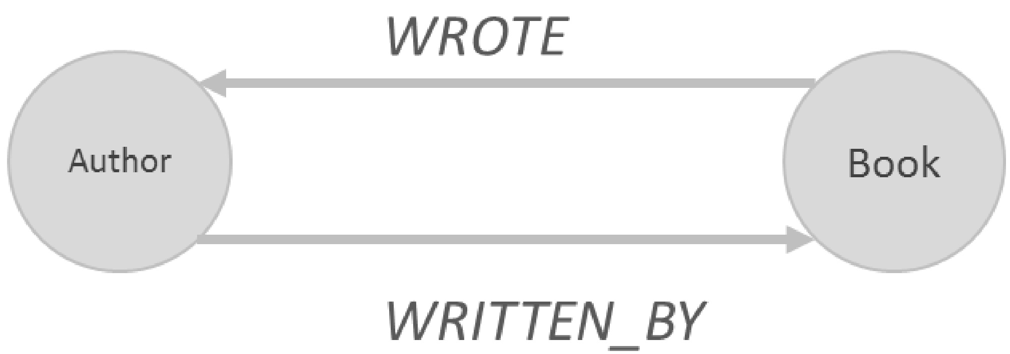
Graph-Based Databases

- One-to-one, one-to-many, and many-to-many structures can easily be modeled in a graph
- Consider the N–M relationship between books and authors
- RDBMS needs three tables: Book, Author and Books_Authors
- SQL query to return all book titles for books written by a particular author would look like this:

```
SELECT title
FROM books, authors, books_authors
WHERE author.id = books_authors.author_id
 AND books.id = books_authors.book_id
 AND author.name = "Bart Baesens"
```

Graph-Based Databases

- In a graph database (using **Cypher query language** from Neo4j)




```
MATCH (b:Book)<-[:WRITTEN_BY]-(a:Author)
WHERE a.name = "Bart Baesens"
RETURN b.title
```

Graph-Based Databases

- A graph database is a hyper-relational database, in which JOIN tables are replaced by more interesting and semantically meaningful relationships that can be navigated and/or queried using graph traversal based on graph pattern matching.

Graph-Based Databases

- Cypher Overview (Neo4j)
- Exploring a social graph

Cypher Overview

- Cypher is a declarative, text-based query language, containing many similar operations as SQL
- Contains a special **MATCH** clause to match those patterns using symbols that look like graph symbols as drawn on a whiteboard
- Nodes are represented by parentheses, representing a circle: `()`
- Nodes can be labeled in case they need to be referred to elsewhere, and be further filtered by their type, using a colon: `(b:Book)`
- Edges are drawn using either `--` or `-->`, representing a unidirectional line or an arrow representing a directional relationship, respectively

Cypher Overview

- Relationships can be filtered by putting square brackets in the middle:
  `(b:Book)<-[:WRITTEN_BY]-(a:Author)`

Cypher Overview

```
MATCH (b:Book)
RETURN b;

MATCH (b:Book)
RETURN b
ORDER BY b.price DESC
LIMIT 20;

MATCH (b:Book)
WHERE b.title = "Beginning Neo4j"
RETURN b;

MATCH (b:Book {title:"Beginning Neo4j"})
RETURN b;
```

Cypher Overview

- JOIN clauses are expressed using direct relational matching

```
MATCH (c:Customer)-[p:PURCHASED]->(b:Book)<-
[:WRITTEN_BY]-(a:Author)
WHERE a.name = "Wilfried Lemahieu"
 AND c.age > 30
 AND p.type = "cash"
RETURN DISTINCT c.name;
```

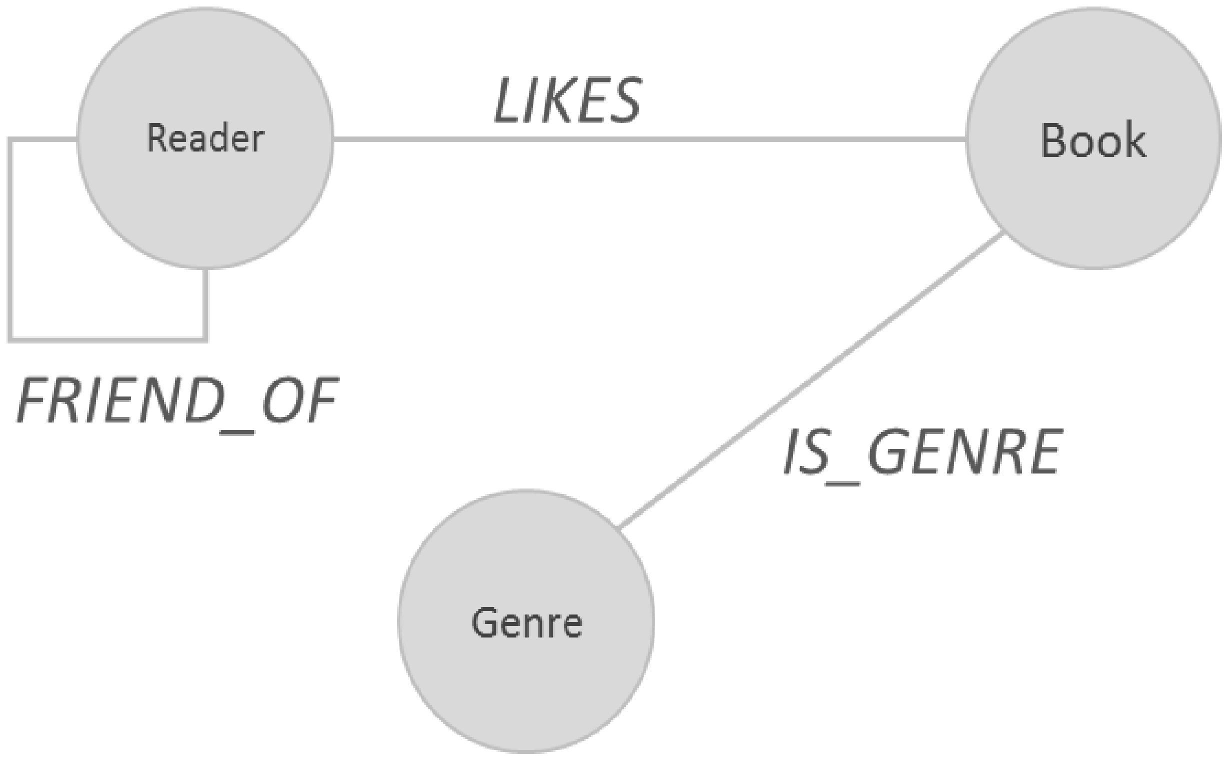
Cypher Overview

- Graph databases are great at managing tree structures
- Example:
  - A tree of book genres and books can be placed under any category level
  - A query to fetch a list of all books in the category "Programming" and all its subcategories
- Cypher can express queries over hierarchies and transitive relationships of any depth simply by appending an asterisk * after the relationship type and providing optional min..max limits

```
MATCH (b:Book)-[:IN_GENRE]->(:Genre)
 -[:PARENT*0..]-(:Genre {name:"Programming"})
RETURN b.title;
```

Exploring a Social Graph

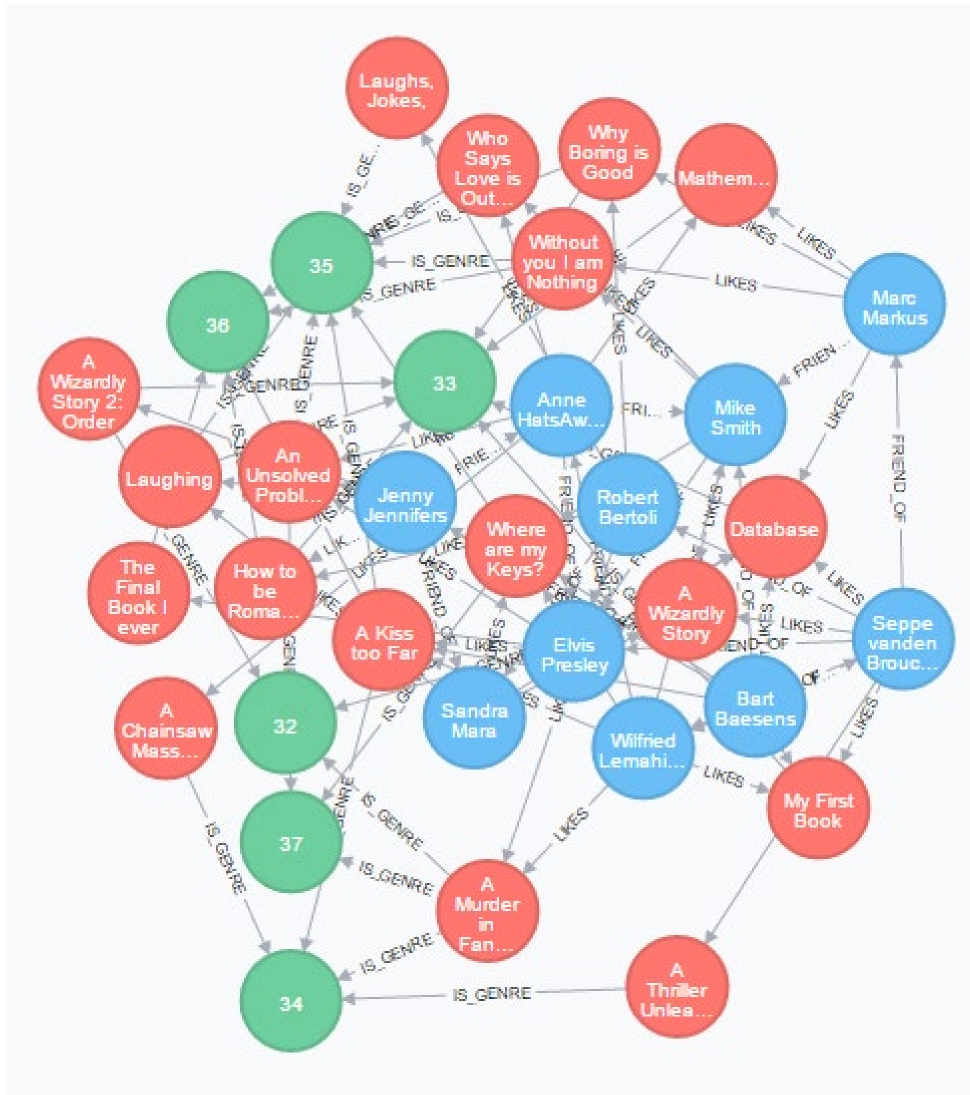
- Example: a social graph for a book-reading club, modeling genres, books, and readers



Exploring a Social Graph

```
CREATE (Bart:Reader {name:'Bart Baesens', age:32})
CREATE (Seppe:Reader {name:'Seppe vanden Broucke', age:30})
…

CREATE (Fantasy:Genre {name:'fantasy'})
CREATE (Education:Genre {name:'education'})
…

CREATE (b01:Book {title:'My First Book'})
CREATE (b02:Book {title:'A Thriller Unleashed'})
…

CREATE
 (b01)-[:IS_GENRE]->(Education),
 (b02)-[:IS_GENRE]->(Thriller),
…

CREATE
 (Bart)-[:FRIEND_OF]->(Seppe),
 (Bart)-[:FRIEND_OF]->(Wilfried),
…
CREATE
 (Bart)-[:LIKES]->(b01), (Bart)-[:LIKES]->(b03),
 (Bart)-[:LIKES]->(b05), (Bart)-[:LIKES]->(b06),
 …
```

Exploring a Social Graph



Exploring a Social Graph

- Who likes romance books?

```
MATCH (r:Reader)--(:Book)--(:Genre
{name:'romance'})
RETURN r.name
```

Returns:

Elvis Presley

Mike Smith

Anne HatsAway

Robert Bertoli

…

Exploring a Social Graph

- Who are Bart's friends that liked Humor books?

```
MATCH (me:Reader)--(friend:Reader)--(b:Book)--(g:Genre)
WHERE g.name = 'humor' AND me.name = 'Bart Baesens'
RETURN DISTINCT friend.name
```

- Can you recommend some humor books that Seppe's friends liked and Seppe has not liked yet?

```
MATCH (me:Reader)--(friend:Reader),
  (friend)--(b:Book),
  (b)--(genre:Genre)
 WHERE NOT (me)--(b)
  AND me.name = 'Seppe vanden Broucke' AND genre.name = 'humor'
 RETURN DISTINCT b.title
```

Exploring a Social Graph

- Get a list of people who have liked books Bart liked, sorted by most liked books in common

```
MATCH (me:Reader)--(b:Book),
 (me)--(friend:Reader)--(b)
WHERE me.name = 'Bart Baesens'
RETURN friend.name, count(*) AS common_likes
ORDER BY common_likes DESC
```

friend.name	common_likes
Wilfried Lemahieu	3
Seppe vanden Broucke	2
Mike Smith	1

Graph Databases

- Location-based services
- Recommender systems
- Social media (e.g., Twitter and FlockDB)
- Knowledge-based systems

Other NoSQL Categories

- XML databases
- OO databases
- Database systems to deal with time series and streaming events
- Database systems to store and query geospatial data
- Database systems such as BayesDB which let users query the probable implication of their data

Evaluating NoSQL DBMSs

- Most NoSQL implementations have yet to prove their true worth in the field
- Some queries or aggregations are particularly difficult; map–reduce interfaces are harder to learn and use
- Some early adopters of NoSQL were confronted with some sour lessons
  – e.g., Twitter and HealthCare.gov

Evaluating NoSQL DBMSs

- NoSQL vendors start focusing again on robustness and durability, whereas RDBMS vendors start implementing features to build schema-free, scalable data stores
- NewSQL: blend the scalable performance and flexibility of NoSQL systems with the robustness guarantees of a traditional RDBMS

Evaluating NoSQL DBMSs

	RDBMSs	NoSQL Databases	NewSQL
Relational	Yes	No	Yes
SQL	Yes	No	Yes
Column stores	No	Yes	Yes
Scalability	Limited	Yes	Yes
Eventually consistent	Yes	Yes	Yes
BASE	No	Yes	No
Big volumes of data	No	Yes	Yes
Schema-less	No	Yes	No

Conclusion

- The NoSQL movement
- Key–value stores
- Tuple and document stores
- Column-oriented databases
- Graph-based databases
- Other NoSQL categories