INTERNET SYSTEMS

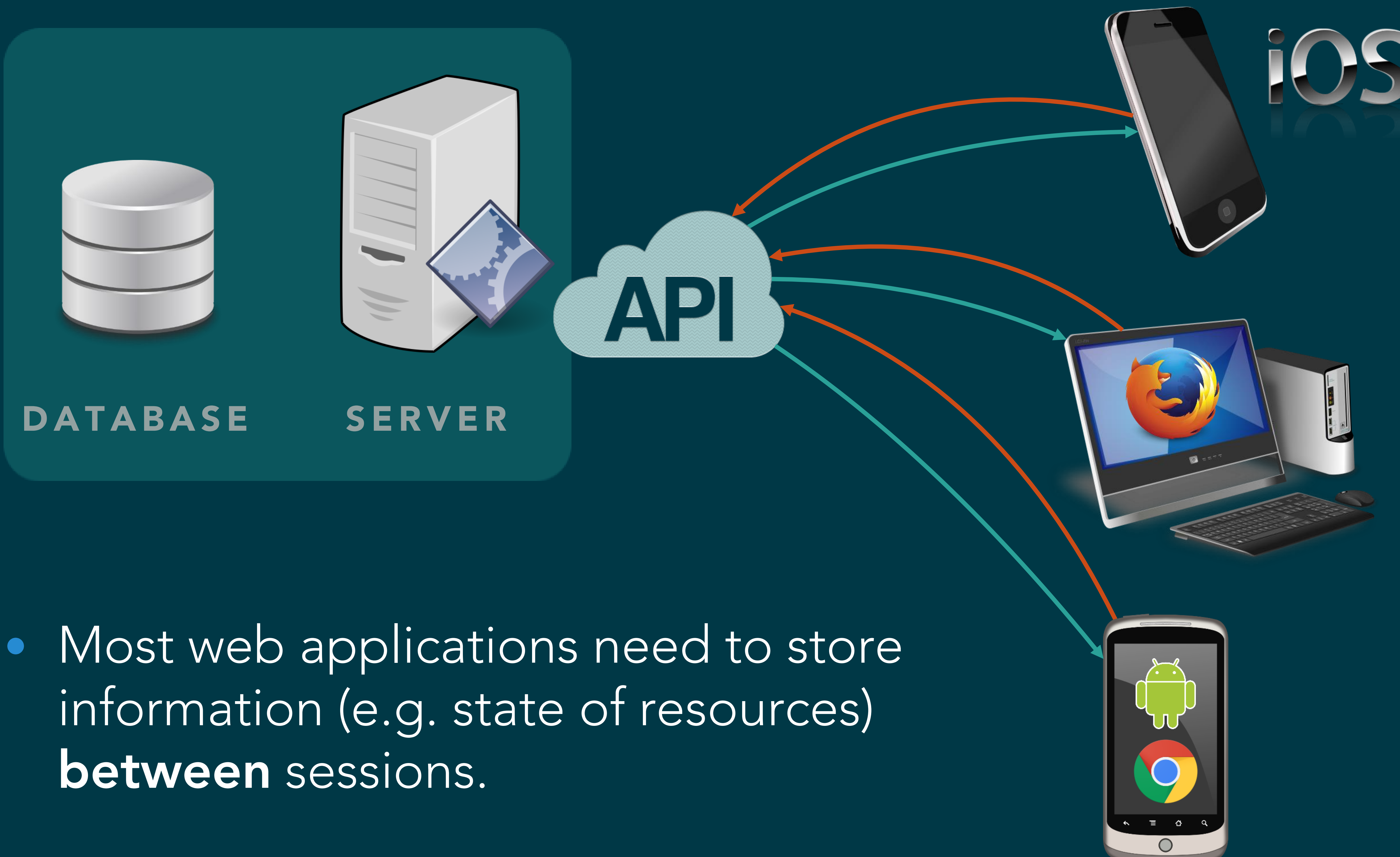
DATA PERSISTENCE

TOMASZ PAWLAK, PHD
MARCIN SZUBERT, PHD
POZNAN UNIVERSITY OF TECHNOLOGY, INSTITUTE OF COMPUTING SCIENCE

PRESENTATION OUTLINE

- Motivation
- Relational Databases
  - JDBC — Java Database Connectivity
  - ORM — Object-Relational Mapping
  - JPA — Java Persistence API
- NoSQL Databases
  - Challenges and common features
  - Key-value stores: Redis, Riak
  - Document-oriented databases: MongoDB, CouchDB
  - Graph databases: Neo4J
  - Column-oriented databases: HBase, Cassandra

MODERN WEB APPLICATION



- Most web applications need to store information (e.g. state of resources) **between** sessions.

MOTIVATION

- **Persistence** is making data last across multiple executions of an application by saving it in **non-volatile storage**.
- Persistence simply means that we would like our application's data to **outlive** the applications process.
- Most web applications achieve persistence by storing data in **databases**, which maintain **data integrity** and potentially can reduce the amount of **data duplication**.
- Database Management Systems (**DBMS**) provides not only persistence but also other services such as **queries**, auditing and **access control**.

PRESENTATION OUTLINE

- Motivation
- **Relational Databases**
  - JDBC — Java Database Connectivity
  - ORM — Object-Relational Mapping
  - JPA — Java Persistence API
- NoSQL Databases
  - Challenges and common features
  - Key-value stores: Redis, Riak
  - Document-oriented databases: MongoDB, CouchDB
  - Graph databases: Neo4J
  - Column-oriented databases: HBase, Cassandra

RELATIONAL DATABASES

- Relational database organizes data into one or more **tables** (**relations**) of rows and columns.
- Each table stores exactly one type of **entity** — the rows (**records**) represent entity instances and the columns represent their attributes (**fields**).

CustNo	Name	Address	City	State	Zip
1	Emma Brown	1565 Rainbow Road	Los Angeles	CA	90014
2	Darren Ryder	4758 Emily Drive	Richmond	VA	23219
3	Earl B. Thurston	862 Gregory Lane	Frankfort	KY	40601
4	David Miller	3647 Cedar Lane	Waltham	MA	02154

RELATIONAL DATABASES

CustNo	Name	Address	City	State	Zip
1	Emma Brown	1565 Rainbow Road	Los Angeles	CA	90014
2	Darren Ryder	4758 Emily Drive	Richmond	VA	23219
3	Earl B. Thurston	862 Gregory Lane	Frankfort	KY	40601
4	David Miller	3647 Cedar Lane	Waltham	MA	02154

ISBN	Title	Price
0596101015	PHP Cookbook	44.99
0596527403	Dynamic HTML	59.99
0596005436	PHP and MySQL	44.95
0596006815	Programming PHP	39.99

CustNo	ISBN	Date
1	0596101015	Mar 03 2009
2	0596527403	Dec 19 2008
2	0596101015	Dec 19 2008
3	0596005436	Jun 22 2009
4	0596006815	Jan 16 2009

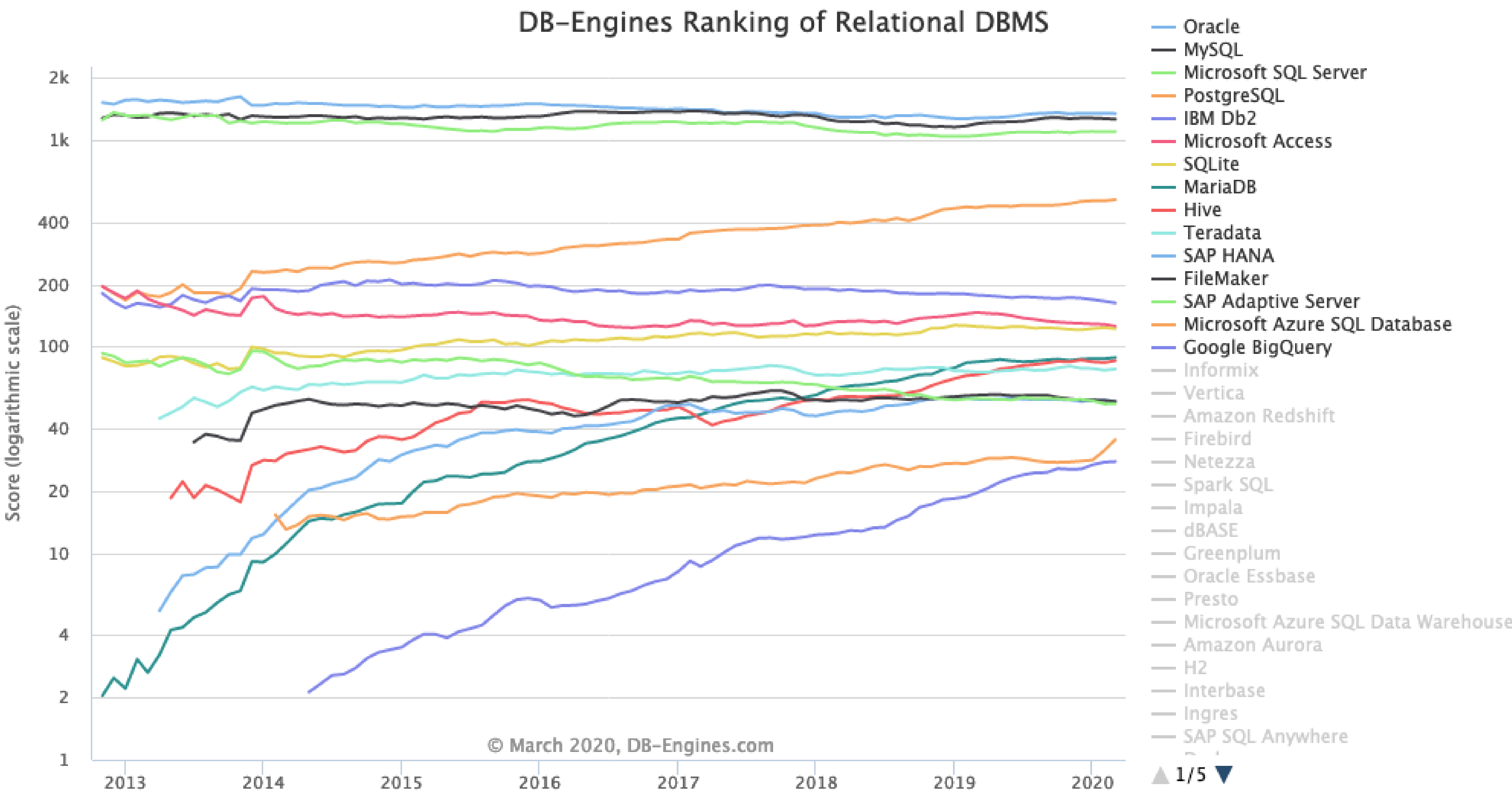
RELATIONAL DATABASES
STRUCTURED QUERY LANGUAGE

- Structured Query Language (**SQL**) — a special-purpose programming language designed for managing data held in a relational database management system. SQL consists of:
  - **data definition language** (**DDL**) — schema creation and modification: **`CREATE, ALTER, TRUNCATE, DROP`**
  - **data manipulation language** (**DML**) — data CRUD operations: **`INSERT, SELECT, UPDATE, DELETE`**
- SQL became a standard of the American National Standards Institute (**ANSI**) in 1986, and of the International Organization for Standardization (**ISO**) in 1987. Current release is SQL:2016.
- Despite standardization, most SQL code is **not completely portable** among different database systems without adjustments.

RELATIONAL DATABASES

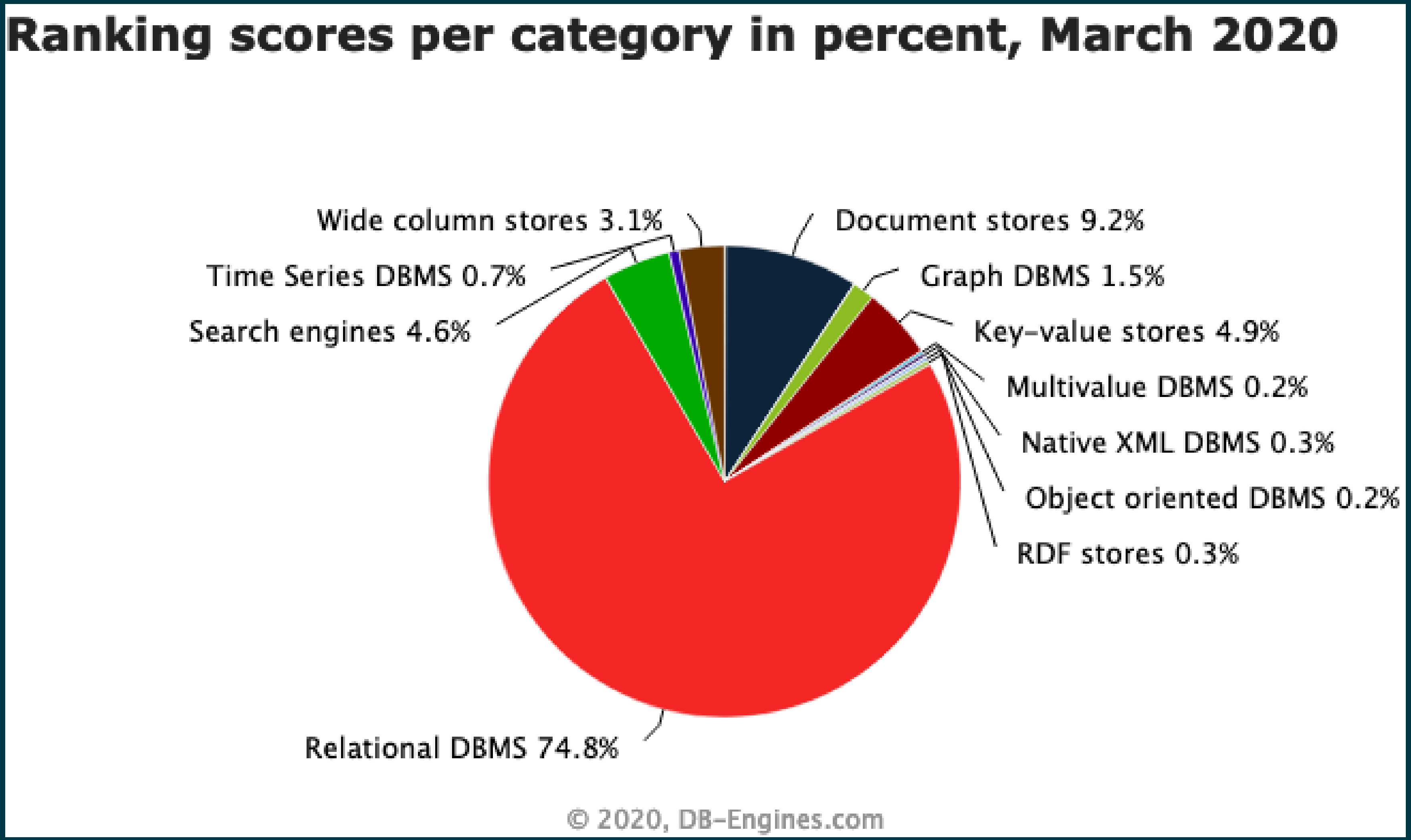
- Oracle RDBMS
- MySQL
- Microsoft SQL Server
- PostgreSQL
- IBM DB2
- SQLite
- Maria DB
- Java DB — Apache Derby

RELATIONAL DATABASES



Source: http://db-engines.com/en/ranking_trend/relational+dbms

RELATIONAL DB VS NOSQL DB



Source: https://db-engines.com/en/ranking_categories

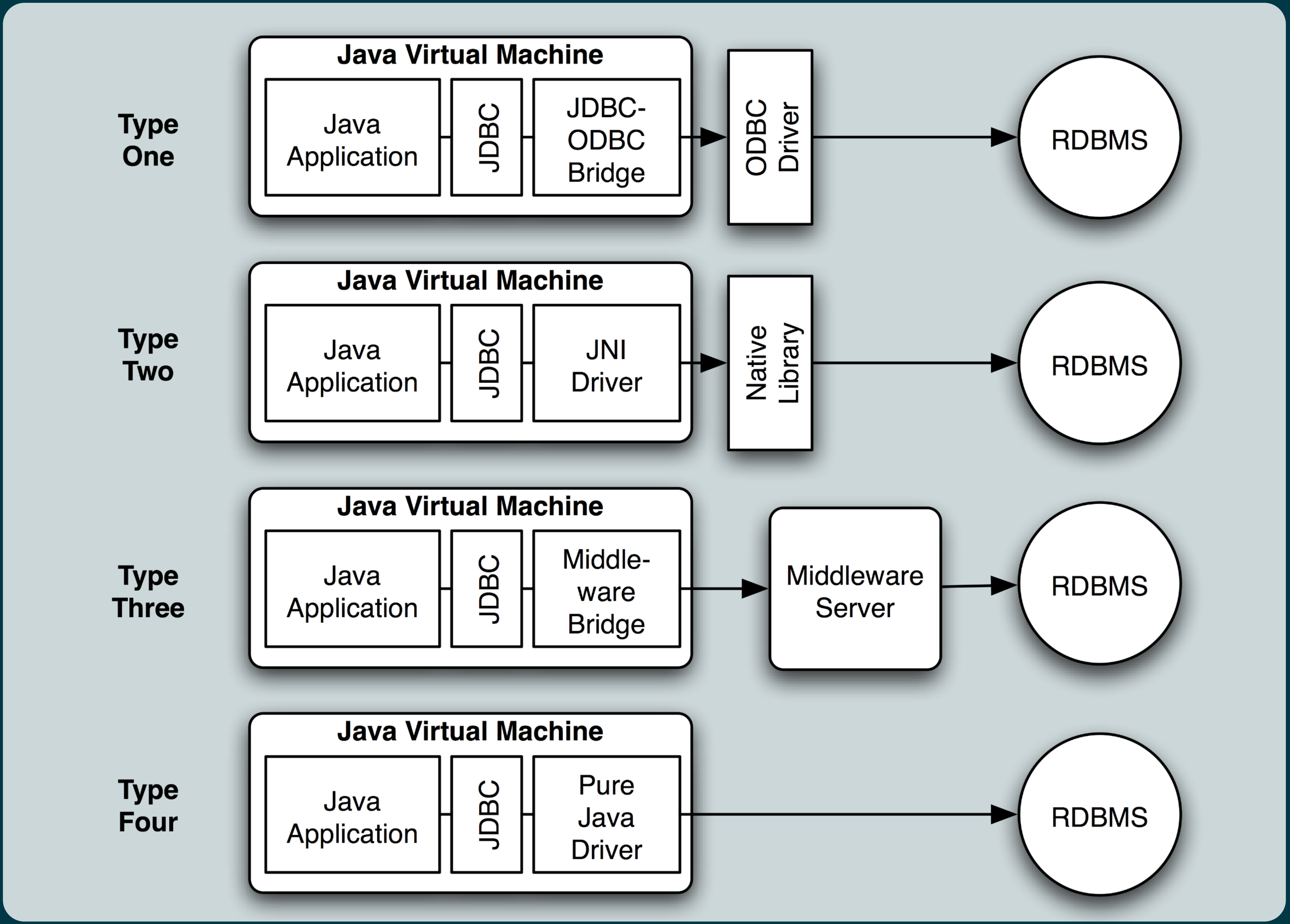
RELATIONAL DATABASES STRENGTHS

- Despite the growing interest in newer database trends, the relational style remains the **most popular** — it has been the focus of intense **academic research** and **industrial improvements** for more than 50 years.
- Many programming models, like **object-relational mapping** (**ORM**), assume an underlying relational database.
- Queries are **flexible** — you needn't know how you plan to actually use the data, since you can always perform some joins, filters, views, and indexes.
- **Normalized** relational databases **minimize redundancy**.

JAVA DATABASE CONNECTIVITY

- The JDBC API provides **universal data access** from the Java programming language to a wide range of **tabular** data, including **relational databases**, spreadsheets, and flat files.
- JDBC allows a Java program to:
  - establish a connection with a data source
  - create SQL statements (e.g. precompiled statements)
  - execute created SQL queries in the database
  - retrieve and modify the resulting records
- To use the JDBC API with a particular DBMS, you need a **JDBC driver** that provides the connection to the database and implements the protocol for transferring the query and result.

JDBC DRIVERS



JDBC EXAMPLE

1. Connect to a data source, like a database.
2. Send queries and update statements to the database.
3. Retrieve and process the results received from the database.

```
public void connectAndQuery(String username, String password) {

    Connection con = DriverManager.getConnection(
                         "jdbc:myDriver:myDatabase",
                         username,
                         password);

    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");

    while (rs.next()) {
        int x = rs.getInt("a");
        String s = rs.getString("b");
        float f = rs.getFloat("c");
    }
}
```

JDBC TRANSACTIONS

- A **transaction** is a set of one or more statements that is executed as a unit, so either all of the statements are executed or none of them is executed.
- When a connection is created, it is in **auto-commit** mode — each individual SQL statement is treated as a transaction and is automatically **committed** right after it is **executed**.

```
try {
  connection.setAutoCommit(false);
  // create and execute statements etc.
  connection.commit();
} catch(Exception e) {
  connection.rollback();
} finally {
  connection.close();
}
```

- Transaction **isolation level** controls the degree of locking that occurs when selecting data, e.g.,

Isolation Level	Transactions	Dirty Reads	Non-Repeatable Reads	Phantom Reads
TRANSACTION_NONE	Not supported	Not applicable	Not applicable	Not applicable
TRANSACTION_READ_UNCOMMITTED	Supported	Allowed	Allowed	Allowed
TRANSACTION_READ_COMMITTED	Supported	Prevented	Allowed	Allowed
TRANSACTION_REPEATABLE_READ	Supported	Prevented	Prevented	Allowed
TRANSACTION_SERIALIZABLE	Supported	Prevented	Prevented	Prevented

JDBC EXAMLE — MOVIE MANAGER

ID	TITLE	DIRECTOR	SYNOPSIS
1	Top Gun	Tony Scott	When Maverick encounters a pair of MiGs…
2	Jaws	Steven Spielberg	A tale of a white shark!

```java
public class MovieManager {

  private Connection connection = null;
  private String url = "jdbc:mysql://localhost:3307/mm";

  private Connection getConnection() {
     if (connection == null) {
        try {
           connection = DriverManager.getConnection(url, "uname", "pass");
        } catch (SQLException e) {
           System.err.println("Exception while creating a connection");
        }
     }
     return connection;
  }
```

JDBC EXAMLE — MOVIE MANAGER PERSISTING A MOVIE

```java
private String insertSql = "INSERT INTO MOVIES VALUES (?,?,?,?)";

private void persistMovie() {
   try {
      PreparedStatement pst = getConnection().
                        prepareStatement(insertSql);

      pst.setInt(1, 1);
      pst.setString(2, "Top Gun");
      pst.setString(3, "Tony Scott");
      pst.setString(4, "Maverick is a pilot. When he encounters a
                  pair of MiGs over the Persian Gulf...");

      pst.execute();
      System.out.println("Movie persisted successfully!");
   } catch (SQLException ex) {
      System.err.println(ex.getMessage());
   }
}
```

JDBC EXAMLE — MOVIE MANAGER PERSISTING A MOVIE

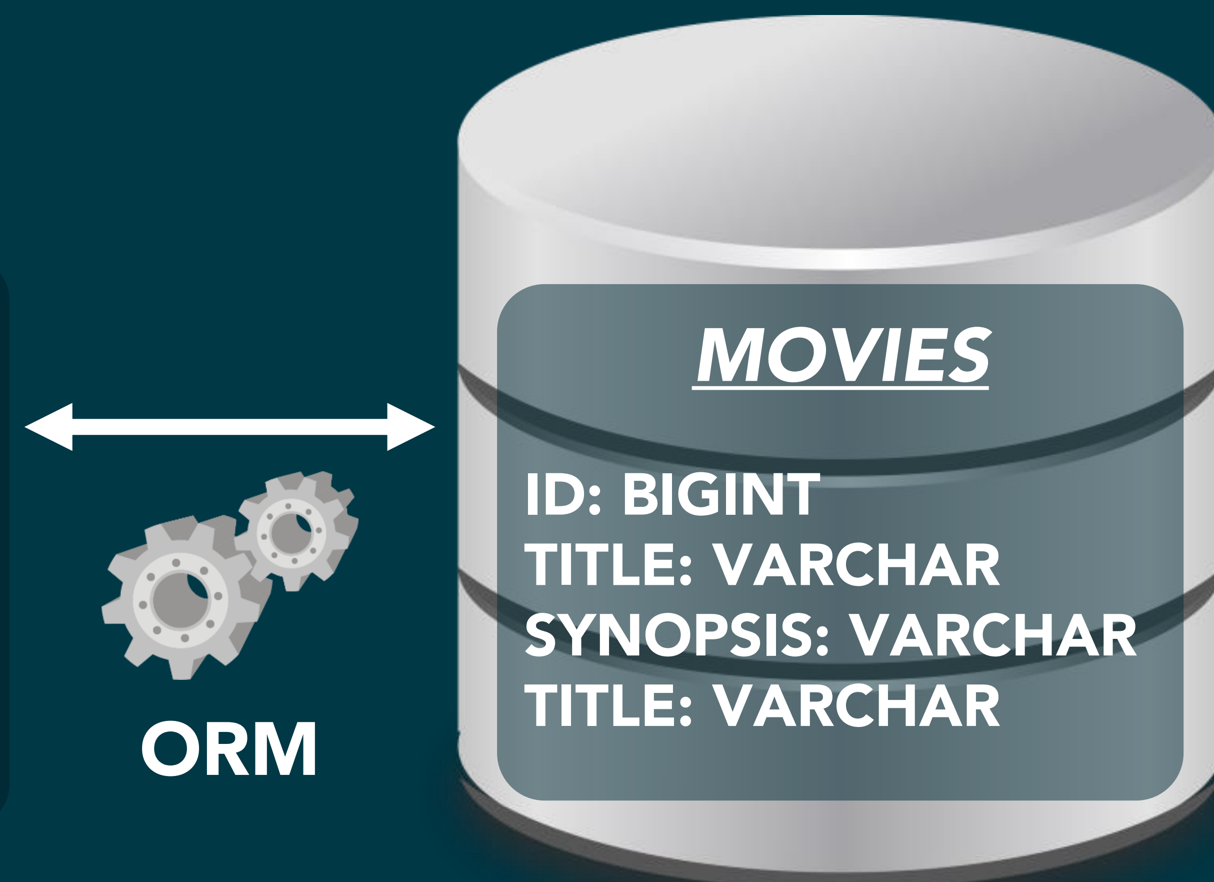
```
public static class Movie {
   private long id;
   private String title;
   private String synopsis;
   private String director;  // Setters and getters omitted
}

private String insertSql = "INSERT INTO MOVIES VALUES (?,?,?,?)";

private void persistMovie(Movie movie) {
   try {
      PreparedStatement pst = getConnection().
                        prepareStatement(insertSql);
      pst.setInt(1, movie.id);
      pst.setString(2, movie.title);
      pst.setString(3, movie.director);
      pst.setString(4, movie.synopsis);
      pst.execute();
   } catch (SQLException ex) {
      System.err.println(ex.getMessage());
   }
}
```

OBJECT-RELATIONAL MAPPING

- Storing **object**-oriented entities in a **relational** database requires a lot of **repetitive**, **error-prone** code and **conversion** between data types.

- **Object-Relational Mapping** (**ORM**) delegates the task of creating a correspondence between objects and tables to external tools — classes, objects, attributes are mapped to tables, rows, columns.

```
public static class Movie {
   private long id;
   private String title;
   private String synopsis;
   private String director;
}
```



JDBC EXAMLE — MOVIE MANAGER PERSISTING A MOVIE

```java
public static class Movie {
    private long id;
    private String title;
    private String synopsis;
    private String director;  // Setters and getters omitted
}

private String insertSql = "INSERT INTO MOVIES VALUES (?,?,?,?)";

private void persistMovie(Movie movie) {
    try {
        PreparedStatement pst = getConnection().
                                prepareStatement(insertSql);
        pst.setInt(1, movie.id);
        pst.setString(2, movie.title);
        pst.setString(3, movie.director);
        pst.setString(4, movie.synopsis);
        pst.execute();
    } catch (SQLException ex) {
        System.err.println(ex.getMessage());
    }
}
```

```java
@PersistenceContext(unitName="MovieManager")
private EntityManager em;

public void persistMovieORM(Movie movie) {
    em.persist(movie);
}
```

ORM BENEFITS

- ORM **reduces** the amount of code that needs to be written — developers transparently use entities instead of tables
- Avoids **low-level** JDBC and SQL code — eliminates the **'hand' mapping** from a SQL `ResultSet` to a POJO.
- Reduces the amount of work required when a domain data model and/or relational data model **change**.
- Provides high end **performance features** such as caching and sophisticated database and query **optimizations**.

ORM DRAWBACKS

- Using an ORM requires creating **formal mapping instructions** telling how to map objects to database records.
- **Switching** between **different ORMs** may require significant work because mapping instructions take different forms.
- The **high level of abstraction** can make it hard to understand what happens behind the scenes — if an ORM generates poor SQL statements, it could result in **bad application performance**.
- Many ORMs can **generate table schema** automatically based on your mapping instructions, but this should **never** be used in a **production** environment.

ORM CHALLENGES

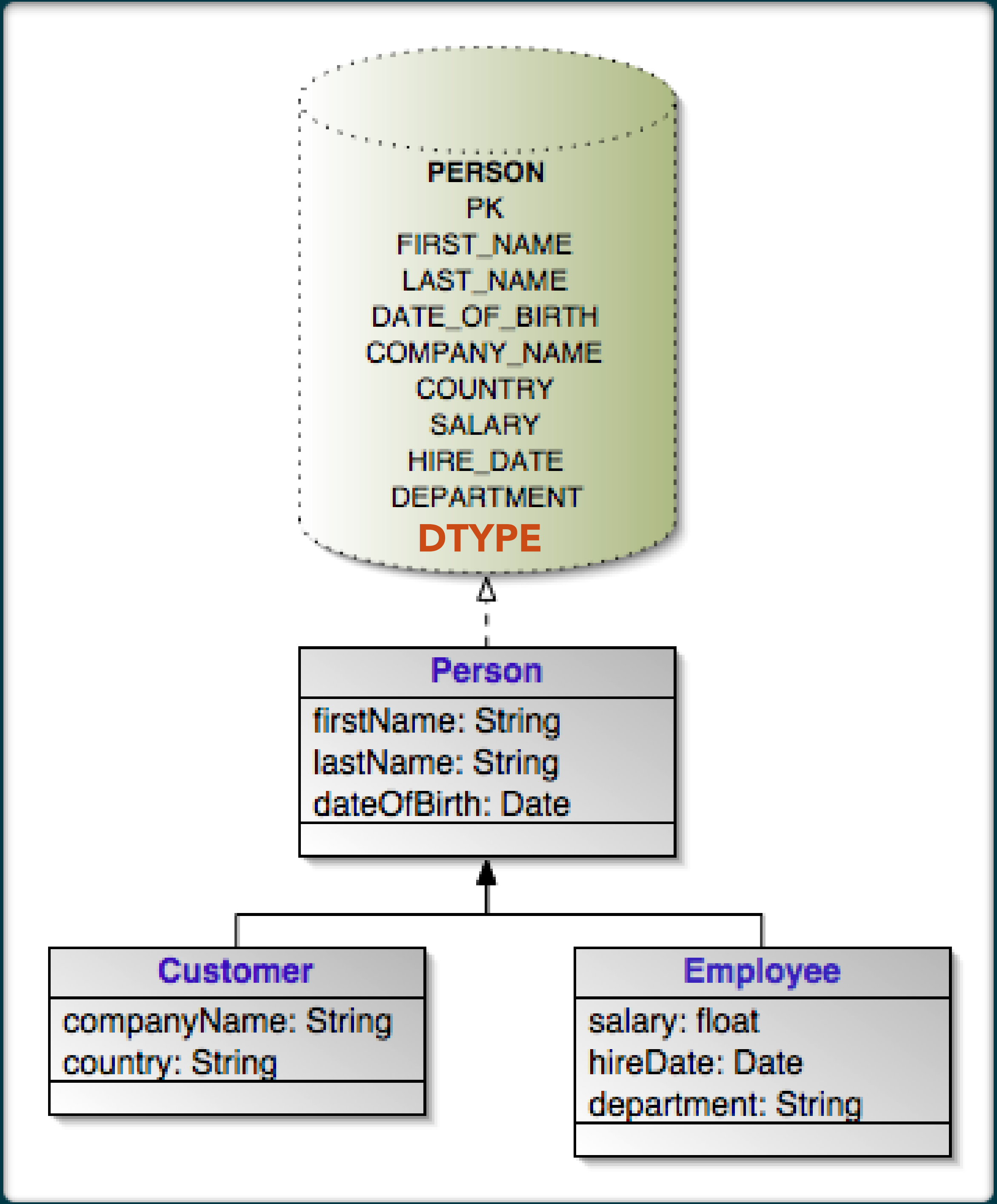
- Type mismatches between programming languages,
- How to find the row from the object, and vice-versa?
- How to keep the object and the row in sync?
- How to represent collections?
- How to represent inheritance?
- How to share sub-objects?

OBJECT-RELATIONAL MISMATCH
INHERITANCE MISMATCH

- Inheritance
  - The fundamental object-oriented programming principle
    - Class B inherits fields and methods of class A
  - No natural way to represent inheritance in a relational database
    - Tables cannot inherit a one from an other

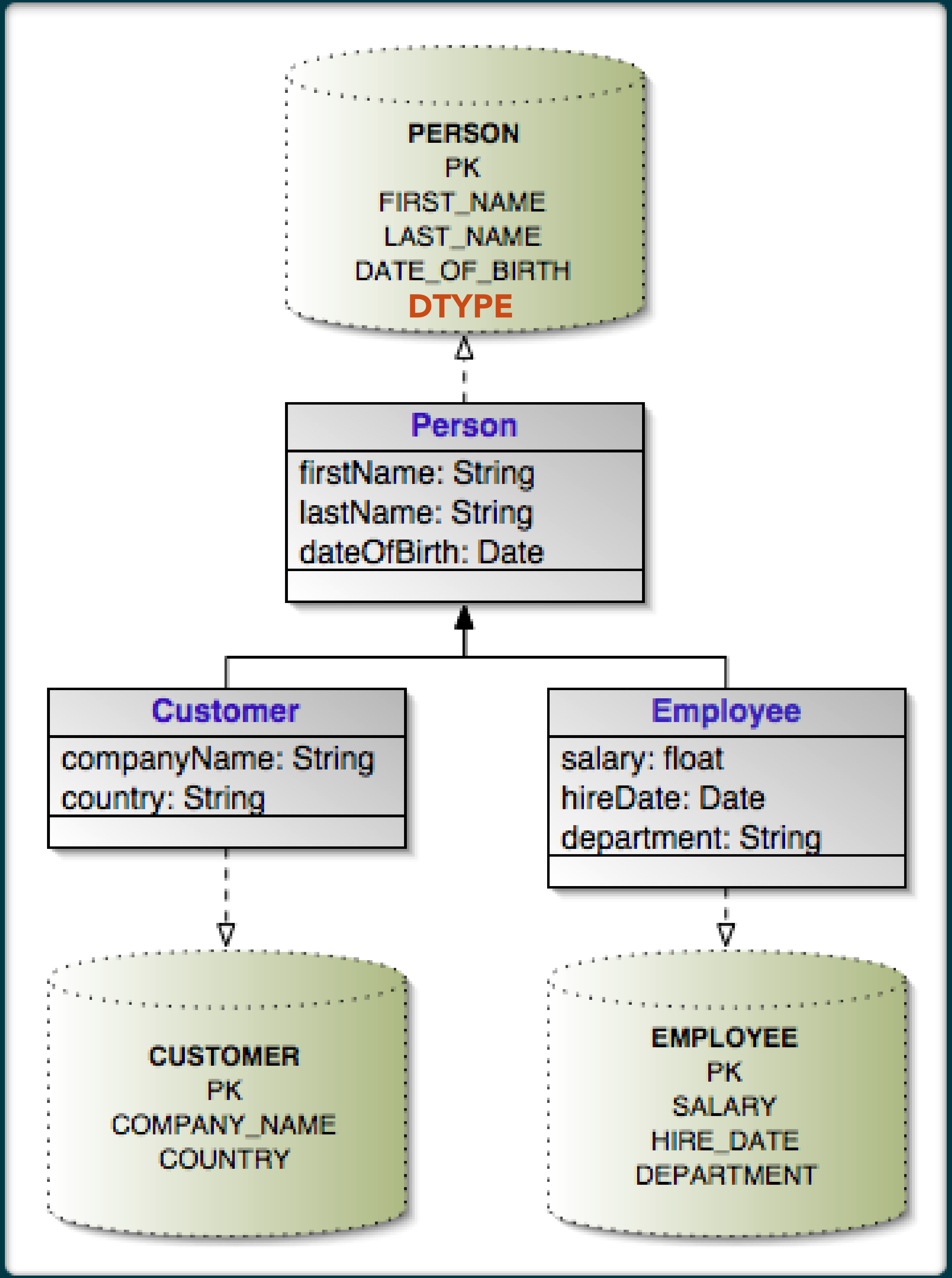
OBJECT-RELATIONAL MISMATCH FLAT INHERITANCE MAPPING

- *Single table strategy:*
  the sum of the attributes of the entire hierarchy is flattened down to a single table (default strategy).
- Advantage — simple and fast:
  - never requires a join to retrieve a single persistent instance from DB;
  - persisting or updating an instance requires only a single statement;
- Disadvantage — wide tables:
  - deep inheritance hierarchy leads to tables with many empty columns.



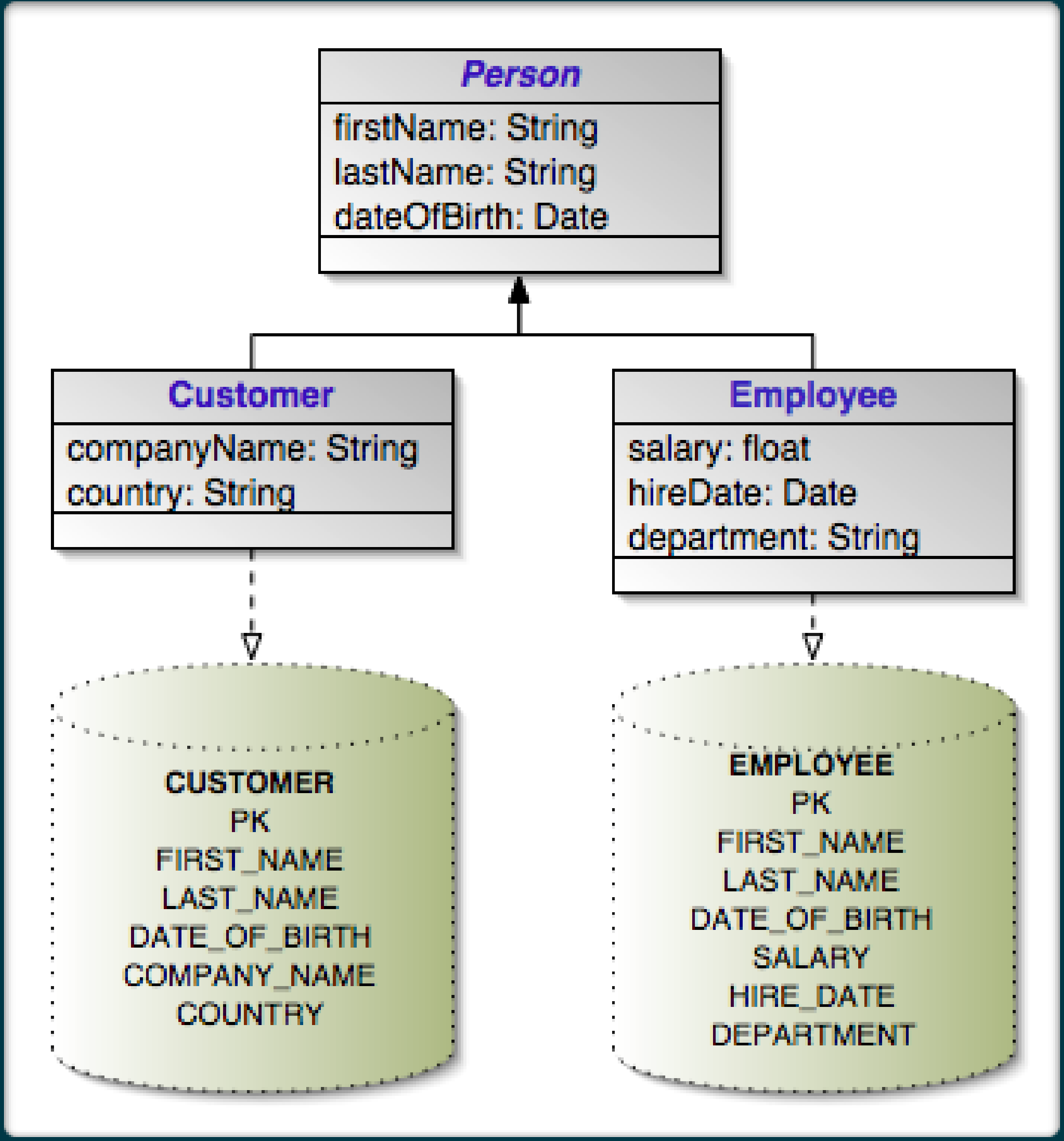
OBJECT-RELATIONAL MISMATCH VERTICAL INHERITANCE MAPPING

- *Joined-subclass strategy:* each entity in the hierarchy, concrete or abstract, is mapped to its own dedicated table.
- Advantage — normalization:
  - redundant data will not exist in any of the tables;
  - adding new subclasses requires minor modifications in database schema;
- Disadvantage — low performance:
  - retrieving or storing subclasses may require multiple join operations;



OBJECT-RELATIONAL MISMATCH
HORIZONTAL INHERITANCE MAPPING

- *Table-per-concrete-class / one-table-per-leaf strategy:* each concrete entity hierarchy is mapped to its own separate table.
- Advantage — efficient (not always):
  - when querying instances of a concrete class — never requires join operations;
  - adding new classes does not require modifying existing tables;
- Disadvantage — restrictions:
  - polymorphic queries across a class hierarchy are expensive (**`UNION`**);



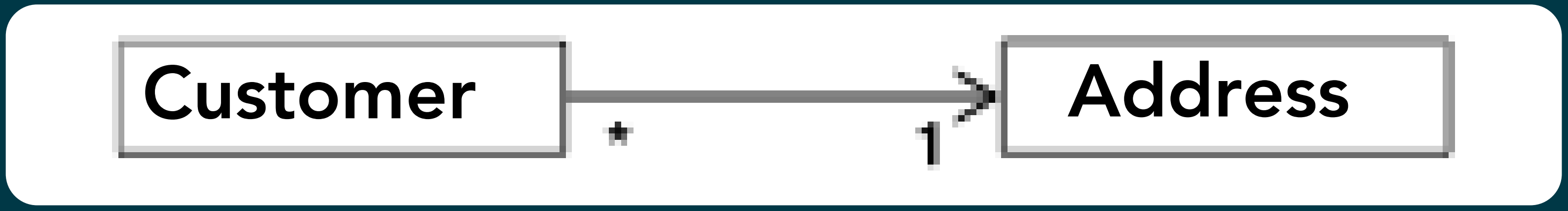
OBJECT-RELATIONAL MISMATCH
CHOOSING INHERITANCE STRATEGIES

- Database maintainability:
  1. **joined-subclass** — changing fields requires modifying only one table; adding new class to the hierarchy only requires a new table (no changes in existing ones).
  2. **table-per-concrete-class** — a change to a column in a parent class requires that the column change be made in all child tables.
  3. **single-table** — many columns that aren't used in every row, as well as a rapidly horizontally growing table.

- Performance:
  1. **single-table** — a select query for any class in the hierarchy will only read from one table, with no joins necessary.
  2. **table-per-concrete-class** — good performance with the leaf nodes in the class hierarchy; any queries related to the parent classes will require unions to get results
  3. **joined-subclass** — requires joins for any select query; the number of joins will be related to the size (depth) of the class hierarchy.

OBJECT-RELATIONAL MISMATCH ASSOCIATIONS AND RELATIONSHIPS

- Representing **associations** in the object world is easy.
- **Multiplicity** refers to how many of the specific objects are related to how many of the other target objects.
  - **One-to-one**: one customer has one address
  - **One-to-many**: one customer has multiple addresses
  - **Many-to-one**: many customers have one address
  - **Many-to-many**: one customer has multiple addresses while one address can be assigned to many customers
- **Directionality** refers to the possibility of navigating from source object to target object: unidirectional or bidirectional.



OBJECT-RELATIONAL MISMATCH MODELING RELATIONSHIPS

Customer

Primary key	Firstname	Lastname	Foreign key
1	James	Rorisson	11
2	Dominic	Johnson	12
3	Maca	Macaron	13

Address

Primary key	Street	City	Country
11	Aligre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

Customer

Primary key	Firstname	Lastname
1	James	Rorisson
2	Dominic	Johnson
3	Maca	Macaron

Address

Primary key	Street	City	Country
11	Aligre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

Join table

Customer PK	Address PK
1	11
2	12
3	13

ELEMENTS OF ORM TECHNOLOGY

- Object-relational **mapping metadata** (in **annotations** or **XML descriptions** — useful when DB configuration changes)
- **API** for managing object persistence and perform database-related operations, such as CRUD.
- **Query language** that allows to retrieve objects without writing SQL queries **specific** to the database.
- **Transactions** and **locking** mechanisms useful when accessing data **concurrently**.
- **Callbacks** and **listeners** to hook business logic into the life cycle of a persistent object.

ORM TECHNOLOGIES IN JAVA

- **1994**: **The Object People** developed **TopLink** for Smalltalk
- **1998**: Java version of TopLink developed.
- **2001**: **Hibernate ORM** started — one of the most popular ORMs, later developed by **JBoss** which is now a division of **Red Hat**.
- **2002**: Oracle Corporation acquired TopLink.
- **2006**: **Java Persistence API** created to provide a standard API for Java persistence in relational databases using ORM technology.
- **2007**: TopLink source code was donated to the Eclipse Foundation and the **EclipseLink** project was born.
- **2009**: Sun Microsystems had selected the EclipseLink project as the **reference implementation** for the JPA 2.0 (later also for the JPA 2.1).

JAVA PERSISTENCE API (JPA)

- JPA 2.1 specification (JSR 338) defines:
  - object-relational mapping metadata (**annotations** and **XML mapping descriptors**)
  - API for the management of persistence (**entity managers** and **persistence contexts**)
  - **Java Persistent Query Language (JPQL)** — a platform-independent query language which resembles SQL in syntax, but operates against entity objects rather than directly with tables.
- JPA implementations use **JDBC** for executing SQL statements, but do not require to deal with JDBC directly.

JPA EXAMPLE

```
EntityManagerFactory factory = Persistence.
                               createEntityManagerFactory("emp");
EntityManager em = factory.createEntityManager();

em.getTransaction().begin();

Query query = em.createQuery("SELECT e " +
                             "  FROM Employee e " +
                             " WHERE e.division = 'Research'" +
                             "   AND e.avgHours > 40");

List results = query.getResultList();
for (Object res : results) {
  Employee emp = (Employee) res;
  emp.setSalary(emp.getSalary() * 1.1);
}

em.getTransaction().commit();

em.close();
```

JPA ENTITIES

- An **entity** is a lightweight persistence domain object that lives shortly in memory and persistently in a database.
- The mapping between entity and DB table is derived following reasonable **defaults** but can be overridden using **annotations** (**convention over configuration**):
  - the entity name is mapped to a relational table name (e.g., the `Book` entity is mapped to a `BOOK` table);
  - attribute names are mapped to a column name (e.g., the `id` attribute, or the `getId()` method, is mapped to an `ID` column);
  - JDBC rules apply for mapping Java primitives to relational data types (e.g. a `String` will be mapped to `VARCHAR`, a `Long` to a `BIGINT`).

JPA ENTITY — ANNOTATIONS

```java
@Entity
@Table(name = "t_contact")
public class Contact implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @NotNull
    protected String firstName;
    @Column(name = "surname", nullable = false, length = 2000)
    protected String lastName;

    @Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",
            message = "{invalid.phonenumber}")
    protected String mobilePhone;

    @Temporal(TemporalType.DATE)
    @Past
    protected Date birthday;

    @Transient
    private Integer age;
}
```

JPA ENTITY — XML MAPPING

- When the metadata are really coupled to the code (e.g., a primary key), it does make sense to use annotations, since the metadata are just another aspect of the program.

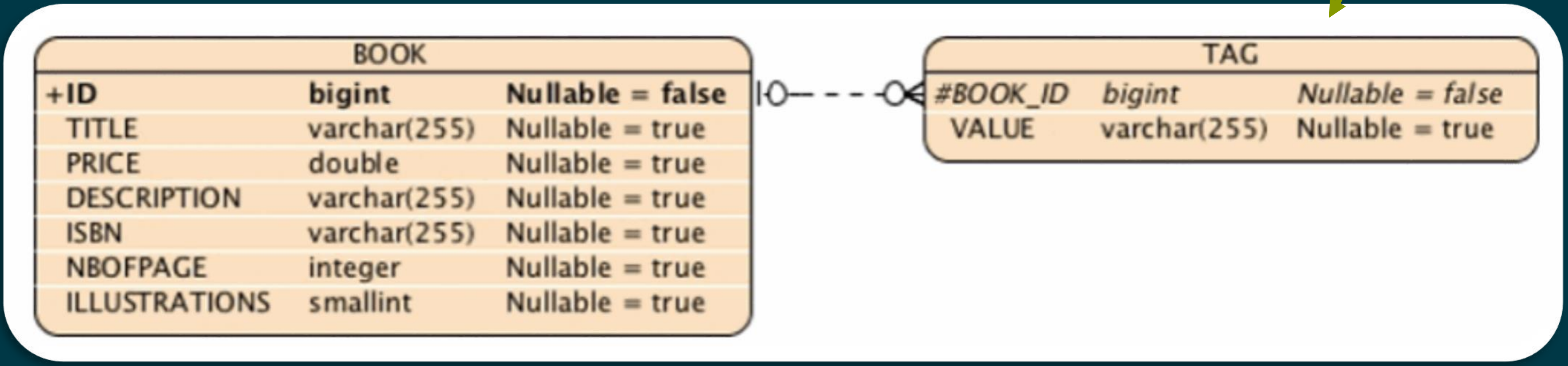
- XML mapping may be used to configure mapping at deployment.
  - XML mapping takes **precedence** over annotations.
  - Certain column options may need to be adjusted depending on the database type in use — this may be better expressed in external **XML deployment descriptors** so the code doesn't have to be modified.

```
<entity class="pl.put.tpsi.Contact">
   <table name="contact_xml_mapping" />
   <attributes>
      <basic name="lastName">
         <column name="familyName" length="500" nullable="false" />
      </basic>
   </attributes>
 </entity>
```

COLLECTIONS IN ENTITIES

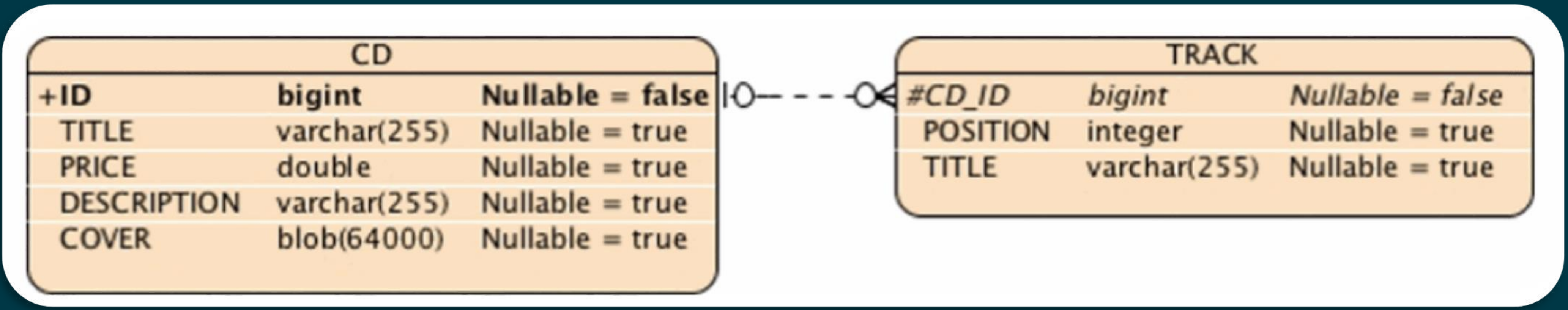
```java
@Entity
public class Book {
  @Id @GeneratedValue
  private Long id;
  private String title;
  private Float price;
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;
  @ElementCollection(fetch = FetchType.LAZY)
  @CollectionTable(name = "Tag")
  @Column(name = "Value")
  private List<String> tags = new ArrayList<>();
}
```

This is not JPA entity



COLLECTIONS IN ENTITIES

```java
@Entity
public class CD {
  @Id @GeneratedValue
  private Long id;
  private String title;
  private Float price;
  private String description;
  @Lob
  private byte[] cover;
  @ElementCollection
  @CollectionTable(name = "track")
  @MapKeyColumn(name = "position")
  @Column(name = "title")
  private Map<Integer, String> tracks = new HashMap<>();
}
```



ENTITY INHERITANCE

- JPA supports three inheritance strategies which are switched by the `strategy` element of `@Inheritance`:
  - `InheritanceType.SINGLE_TABLE` — default strategy; table contains a **discriminator column** to identify the subclass to which the instance represented by the row belongs;
  - `InheritanceType.JOINED` — the root table contains the **discriminator column**; each subclass table contains its own attributes and a **primary key** that refers to the root table's **primary key** (they do not hold a discriminator column).
  - `InheritanceType.TABLE_PER_CLASS` — support for this strategy is **optional**; there is no discriminator column, no shared columns but all tables must share a **common primary key** that matches across all tables in the hierarchy.

ENTITY INHERITANCE

```java
@Entity
@Inheritance(strategy = InheritanceType.
                 SINGLE_TABLE)
@DiscriminatorColumn (name="disc",
                 discriminatorType =
                 DiscriminatorType.CHAR)
@DiscriminatorValue("I")
public class Item {
  @Id @GeneratedValue
  protected Long id;
  protected String title;
  protected Float price;
  protected String description;
}
```

```java
@Entity
@DiscriminatorValue("B")
public class Book extends Item {
  private String isbn;
  private String publisher;
  private Integer nbOfPage;
  private Boolean illustrations;
}

@Entity
@DiscriminatorValue("C")
public class CD extends Item {
  private String musicCompany;
  private Integer numberOfCDs;
  private Float totalDuration;
  private String genre;
}
```

ID	DISC	TITLE	PRICE	DESCRIPTION	MUSIC COMPANY	ISBN	...
1	I	Pen	2.10	Beautiful black pen			...
2	C	Soul Trane	23.50	Fantastic jazz album	Prestige		...
3	C	Zoot Allures	18	One of the best of Zappa	Warner		...
4	B	The robots of dawn	22.30	Robots everywhere		0-554-456	...
5	B	H2G2	17.50	Funny IT book ;o)		1-278-983	...

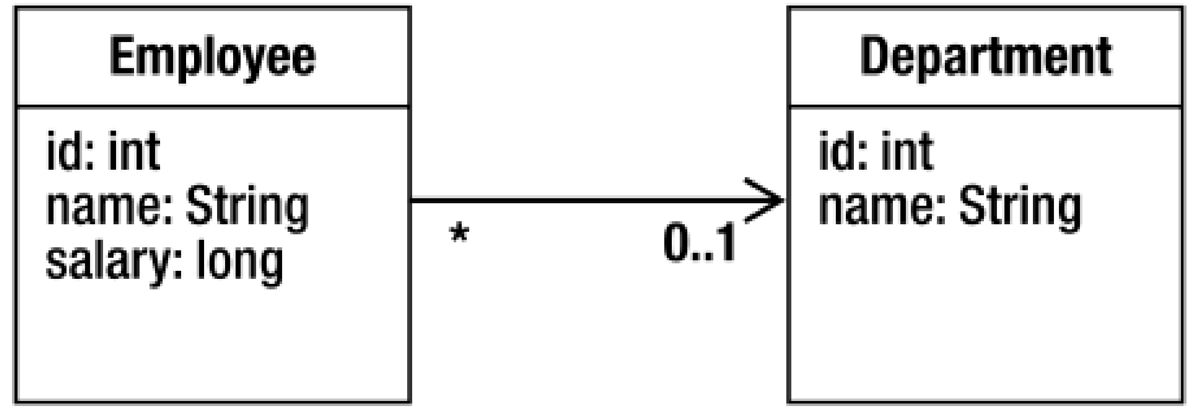
ENTITY RELATIONSHIPS

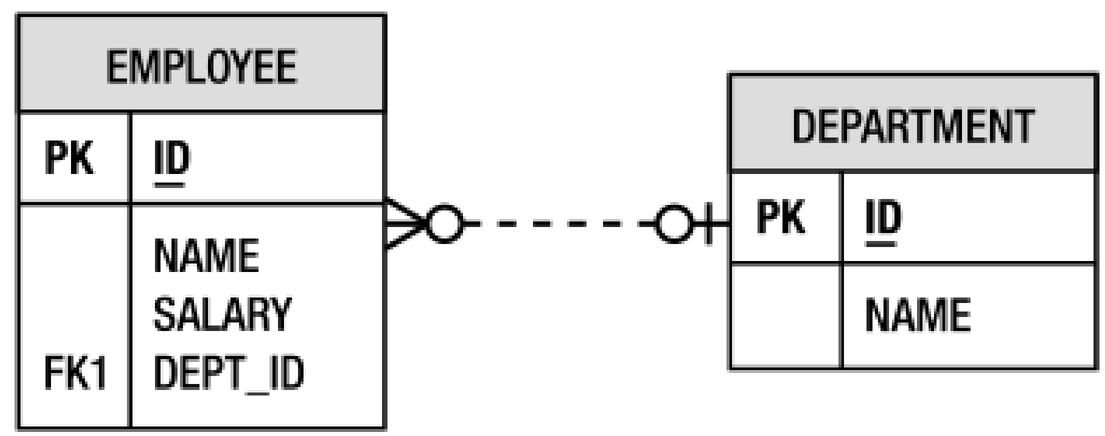
- If entities contained only simple persistent state, the business of object-relational mapping would be a trivial.
- Most entities have **associations** with other entities:
  - **single-valued associations** — an association from an entity instance to another entity instance (where the cardinality of the target is "one"): `@ManyToOne` and `@OneToOne`;
  - **many-valued association** — the source entity references one or more target entity instances, i.e. relationship is to a collection of other objects: `@OneToMany` and `@ManyToMany`;
- All relationships in Java and JPA are **unidirectional**, while in a relational database relationships are defined through **foreign keys** and **querying** such that the inverse query always exists.

SINGLE-VALUED ASSOCIATIONS
MANY-TO-ONE MAPPING

```java
@Entity
public class Employee {
  @Id
  private int id;
  private String name;

  @ManyToOne
  @JoinColumn(name="DEPT_ID")
  private Department department;
  // ...
}
```

```java
@Entity
public class Department {
  @Id
  private int id;
  private String name;
  // ...
}
```

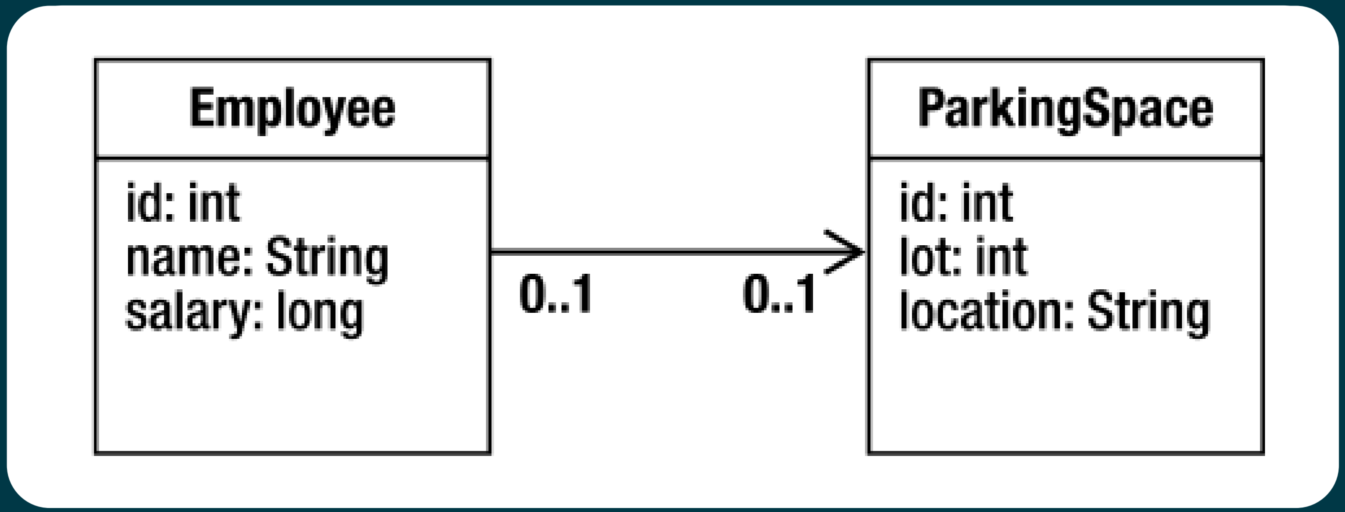


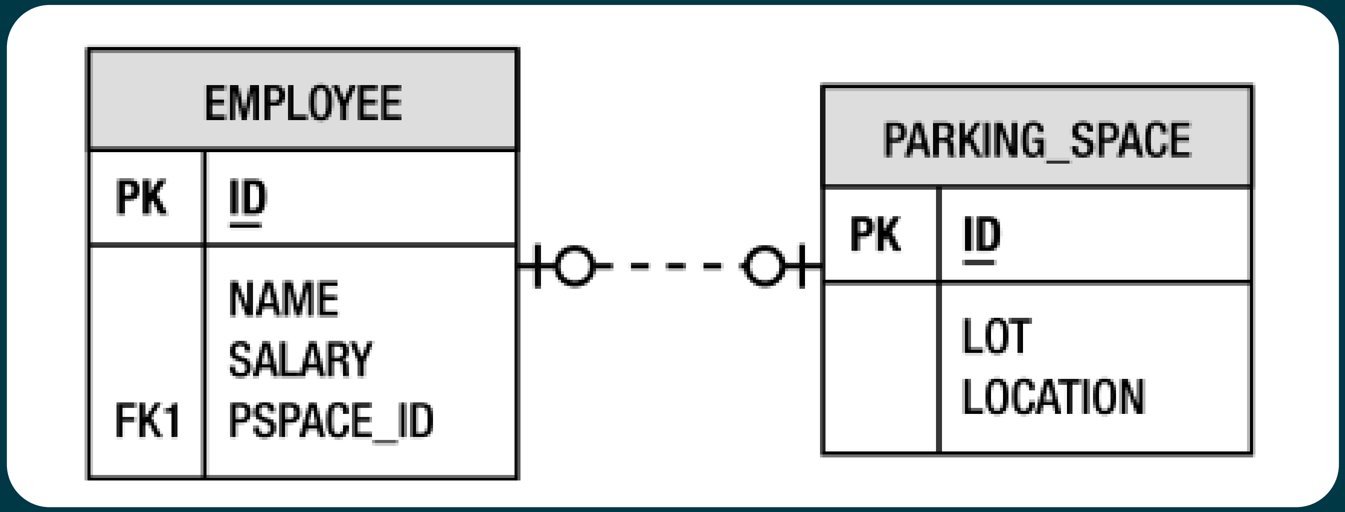


SINGLE-VALUED ASSOCIATIONS
ONE-TO-ONE MAPPING

```java
@Entity
public class Employee {
  @Id
  private int id;
  private String name;

  @OneToOne
  @JoinColumn(name="PSPACE_ID")
  private ParkingSpace parkingSpace;
  // ...
}
```

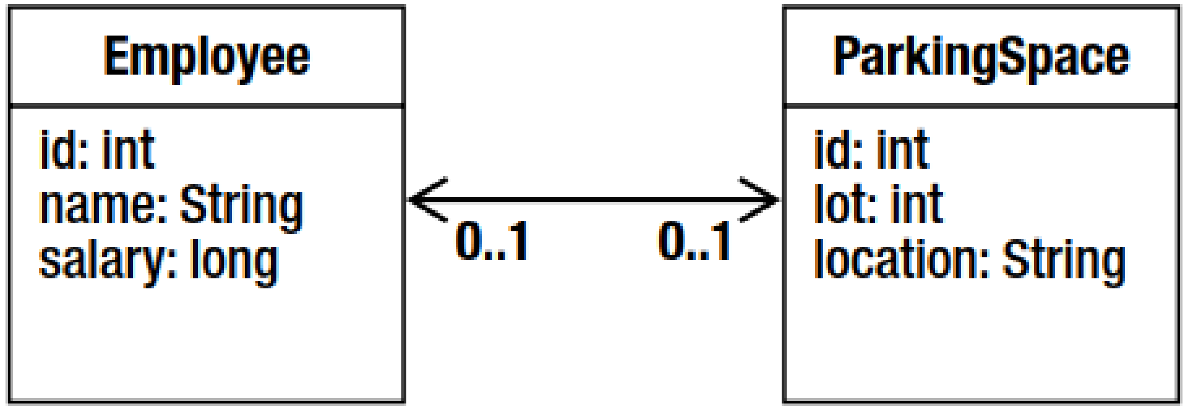
```java
@Entity
public class ParkingSpace {
  @Id
  private int id;
  private int lot;
  private String location;
  // ...
}
```

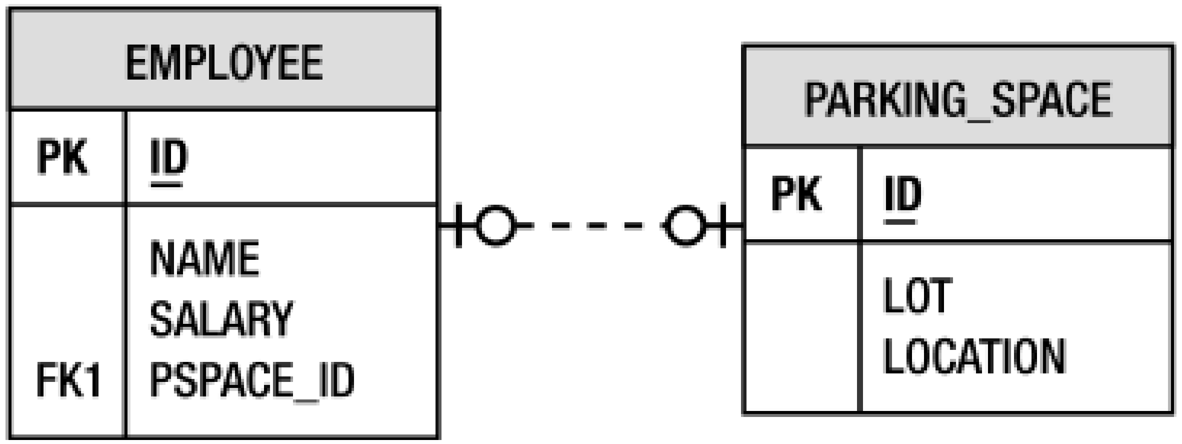




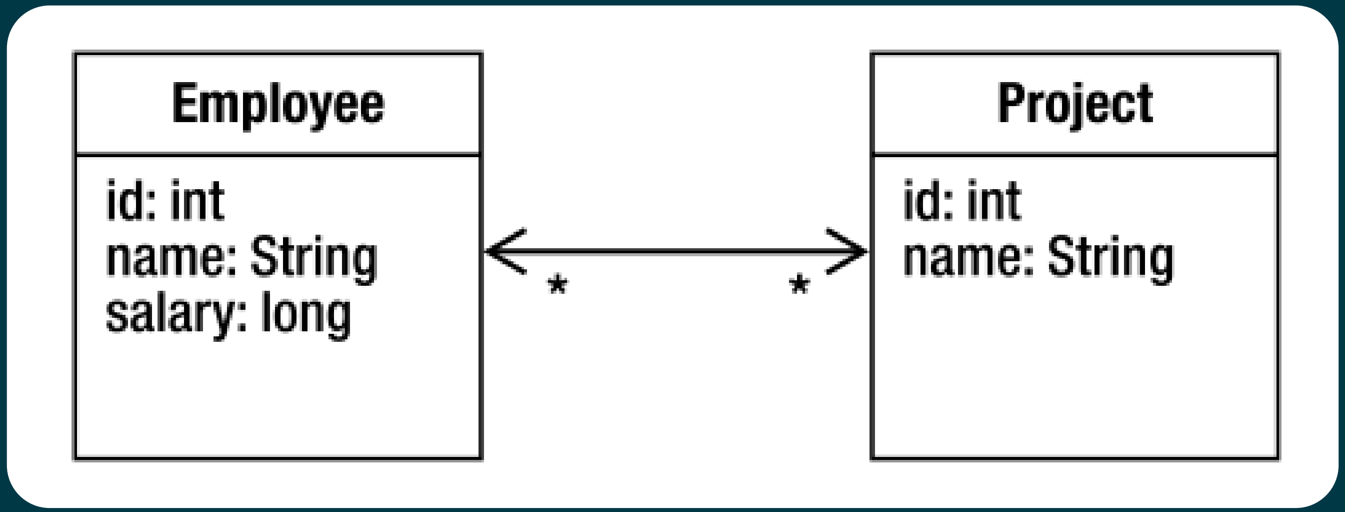
SINGLE-VALUED ASSOCIATIONS BIDIRECTIONAL ONE-TO-ONE MAPPING

```java
@Entity
public class Employee {
  @Id
  private int id;
  private String name;

  @OneToOne
  @JoinColumn(name="PSPACE_ID")
  private ParkingSpace parkingSpace;
  // ...
}
@Entity
public class ParkingSpace {
  @Id
  private int id;
  private int lot;
  private String location;

  @OneToOne(mappedBy="parkingSpace")
  private Employee employee;
  // ...
}
```





COLLECTION-VALUED ASSOCIATIONS
BIDIRECTIONAL MANY-TO-MANY MAPPING

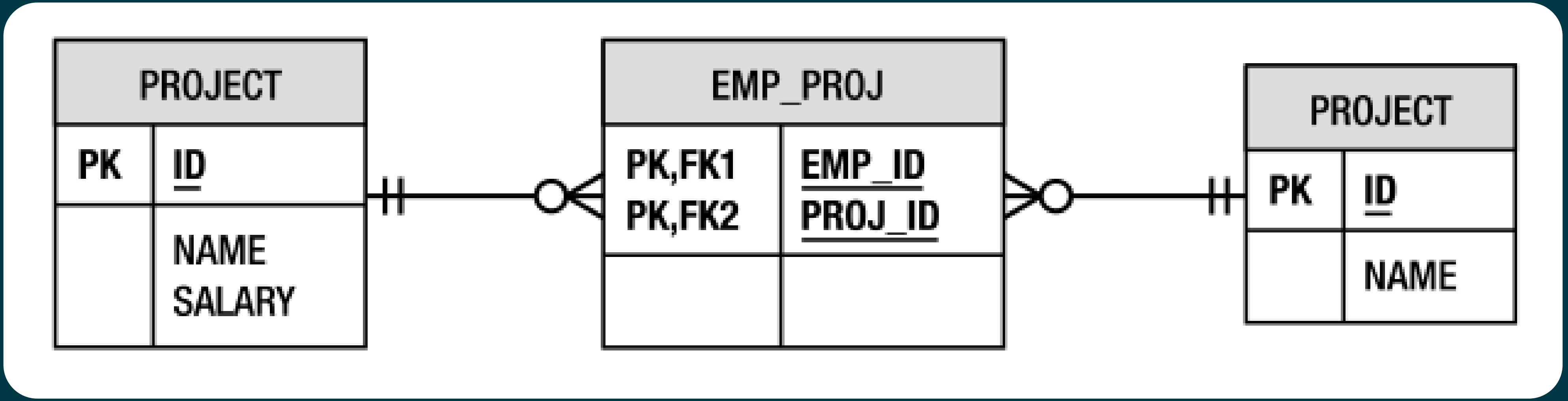



```java
@Entity
public class Employee {
  @Id
  private int id;
  private String name;

  @ManyToMany
  private Collection<Project> projects;
  // ...
}
```

```java
@Entity
public class Project {
  @Id
  private int id;
  private String name;

  @ManyToMany(mappedBy="projects")
  private Collection<Employee> employees;
  // ...
}
```

COLLECTION-VALUED ASSOCIATIONS
BIDIRECTIONAL MANY-TO-MANY MAPPING

```java
@Entity
public class Employee {
  @Id
  private int id;
  private String name;

  @ManyToMany
  @JoinTable(name="EMP_PROJ",
             joinColumns=@JoinColumn(name="EMP_ID"),
             inverseJoinColumns=@JoinColumn(name="PROJ_ID"))
  private Collection<Project> projects;
  // ...
}
```



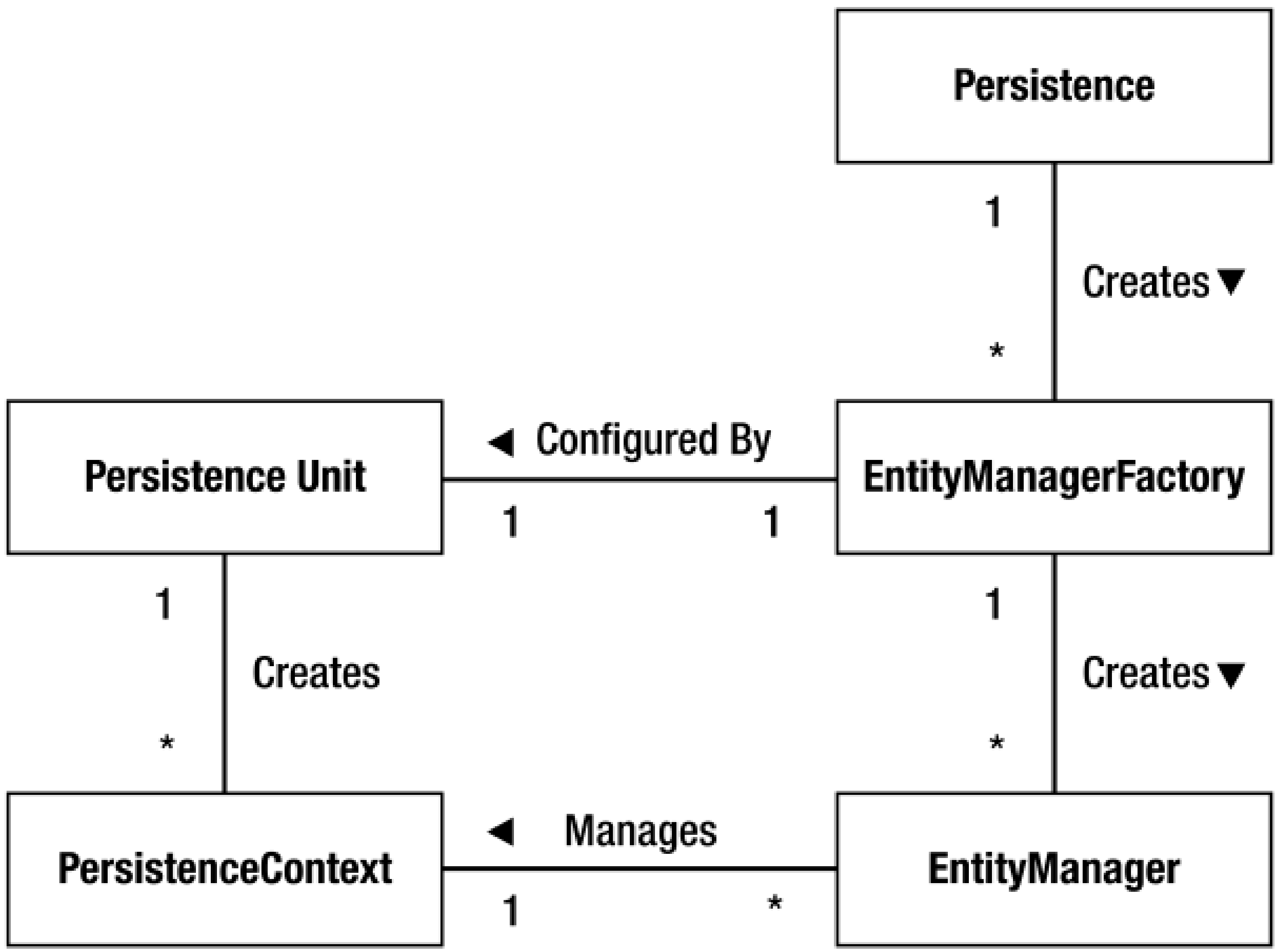
ENTITY MANAGER

- The **entity manager** is a central piece of JPA:
  - **manages** the state and **life cycle** of entities in a **persistence context**
  - **creates** and **removes** persistent entity instances,
  - **finds** entities by their **primary** key,
  - **locks** entities for protecting against **concurrent** access,
  - **executes JPQL queries** to retrieve entities following certain **criteria**.

- `EntityManager` is an interface implemented by a JPA provider that generates and executes SQL statements.

- **Persistence context** — a set of managed entity instances in which only one entity instance with the same persistent **identity** can exist (can be seen as a **first-level cache** where the entity manager stores entities before flushing the content to the database).

ENTITY MANAGER ARCHITECTURE




```
EntityManagerFactory emf = Persistence.
                           createEntityManagerFactory("unit");
EntityManager em = emf.createEntityManager();
```

ENTITY MANAGER IN ACTION

```java
protected EntityManager em;

public Employee createEmployee(int id, String name, long salary) {
   Employee emp = new Employee(id, name, salary);
   em.getTransaction().begin();
   em.persist(emp);
   em.getTransaction().commit();
   return emp;
}

public void removeEmployee(Employee emp) {
   if (emp != null) {
      em.getTransaction().begin();
      em.remove(emp);
      em.getTransaction().end();
   }
}

public Employee findEmployee(int id) {
   return em.find(Employee.class, id);
}
```

PERSISTENCE UNIT

- The **persistence unit** indicates to the **entity manager**:
  - type of database to use,
  - connection parameters,
  - list of entities that can be managed in a persistence context.
- Additionally, persistence unit:
  - specifies a persistence provider
  - declares the type of transactions
    - JTA – external transaction manager
    - resource-local – use DBMS
  - references external XML mapping files
- Persistence units are defined in an XML file called **persistence.xml**, which can contain one or more **named** persistence unit configurations, but each persistence unit is separate and distinct from the others.

PERSISTENCE UNIT
DATABASE SCHEMA GENERATION

- JPA 2.1 introduces an API and properties of persistence.xml that allow **generation** of database artifacts like tables, indexes, and constraints in a **standard** and **portable** way.
- The persistence provider can be configured to:
  - create the database tables,
  - load data into the tables,
  - remove the tables.
- These tasks are typically used during the **development** phase of a release, not against a **production** database.

PERSISTENCE UNIT EXAMPLE

```
<persistence-unit name="tpsi" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <class>pl.put.tpsi.Book</class>
  <mapping-file>META-INF/book_mapping.xml</mapping-file>
  <properties>
     <property name="javax.persistence.schema-generation.database.action"
        value="drop-and-create" />
     <property name="javax.persistence.schema-generation.scripts.action"
        value="drop-and-create" />
     <property name="javax.persistence.schema-generation.scripts.create-target"
        value="create.sql" />
     <property name="javax.persistence.schema-generation.scripts.drop-target"
        value="drop.sql" />

     <property name="javax.persistence.jdbc.driver"
        value="org.apache.derby.jdbc.EmbeddedDriver" />

     <property name="javax.persistence.jdbc.url"
        value="jdbc:derby:lab05;create=true" />

     <property name="eclipselink.logging.level" value="INFO" />
  </properties>
</persistence-unit>
```

QUERYING DATABASE

- `EntityManager` API allows to find a **single** entity using its unique identifier.
- To retrieve a **set** of entities based of different **criteria**, five different types of **queries** that can be used in code:
  - **Dynamic JPQL queries** — the simplest form of query, consisting of a JPQL query string dynamically specified at runtime.
  - **Named JPQL queries** — static and unchangeable.
  - **Criteria API** — object-oriented query API (introduced in JPA 2.0)
  - **Native queries** — a native SQL statement instead of a JPQL
  - **Stored procedure queries** — JPA 2.1 brings a new API to call stored procedures.

QUERYING DATABASE
DYNAMIC QUERIES

- Dynamic queries are defined on the fly directly within an application's business logic:

```
String jpqlQuery = "SELECT c FROM Customer c";
if (someCriteria) {
    jpqlQuery += " WHERE c.firstName = 'Betty'";
}
query = em.createQuery(jpqlQuery, Customer.class);
List<Customer> customers = query.getResultList();
```

- Queries can be parameterized by using named parameters prefixed with a colon:

```
query = em.createQuery("SELECT c FROM Customer c
                         WHERE c.firstName = :fname");
query.setParameter("fname", "Betty");
```

QUERYING DATABASE
NAMED JPQL QUERIES

- Named queries are static queries expressed in metadata inside either a `@NamedQuery` annotation or the XML equivalent.

```
@Entity
@NamedQueries({
 @NamedQuery(name = "findAll", query="SELECT c FROM Customer c"),
 @NamedQuery(name = "findWithParam", query="SELECT c FROM Customer c
              WHERE c.firstName = :fname")
})
public class Customer { }
```

- Executing named queries:

```
Query query = em.createNamedQuery("findWithParam", Customer.class);
query.setParameter("fname", "Vincent");
List<Customer> customers = query.getResultList();
```

QUERYING DATABASE
CRITERIA API

- JPA 2.0 introduced Criteria API which allows to write any query in an **object-oriented** and **syntactically** correct way.
- Most of the mistakes that a developer could make writing a statement are found at **compile time**, not at **runtime** (in contrast to writing JPQL query strings).

```java
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.equal(c.get("firstName"), "Vincent"));
Query query = em.createQuery(criteriaQuery);
List<Customer> customers = query.getResultList();
```

JPA PROVIDERS

- **EclipseLink**
  - Reference implementation of JPA 2.0 and JPA 2.1
  - Included in GlassFish and Oracle WebLogic application servers
- **Hibernate ORM**
  - Provides its own native API, in addition to full JPA support
  - Included in JBoss / WildFly application server
- **Apache OpenJPA**
  - Included in IBM WebSphere application server

OTHER ORM FRAMEWORKS

- Java
  - Apache Cayenne
  - JOOQ
- .NET
  - Entity Framework
  - NHibernate
- Python
  - SQL Alchemy
- Ruby on Rails

PRESENTATION OUTLINE

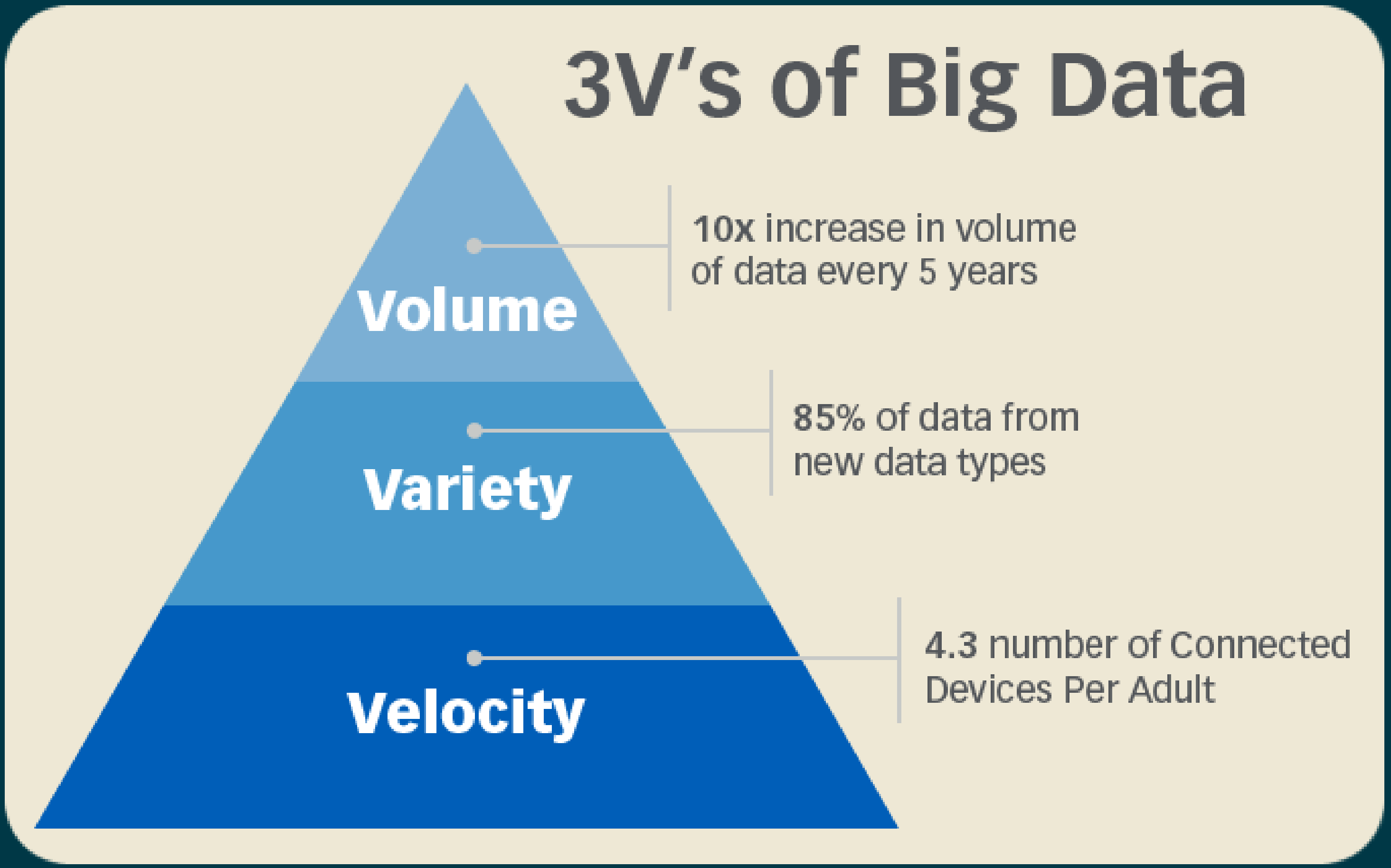
- Motivation
- Relational Databases
  - JDBC — Java Database Connectivity
  - ORM — Object-Relational Mapping
  - JPA — Java Persistence API
- **NoSQL Databases**
  - Challenges and common features
  - Key-value stores: Redis, Riak
  - Document-oriented databases: MongoDB, CouchDB
  - Graph databases: Neo4J
  - Column-oriented databases: HBase, Cassandra

NOSQL DATABASES

- **NoSQL** databases come in a **variety** of shapes — the only feature that unifies them is that they are **not relational**.
- There are **no standard APIs** to interact with a NoSQL database — each NoSQL database offers its own **library**.
- Web-friendly, language-agnostic interactions — many NoSQL databases ship with **RESTful APIs** based on HTTP.
- NoSQL databases find growing use in industry of **big data** and **web applications**.

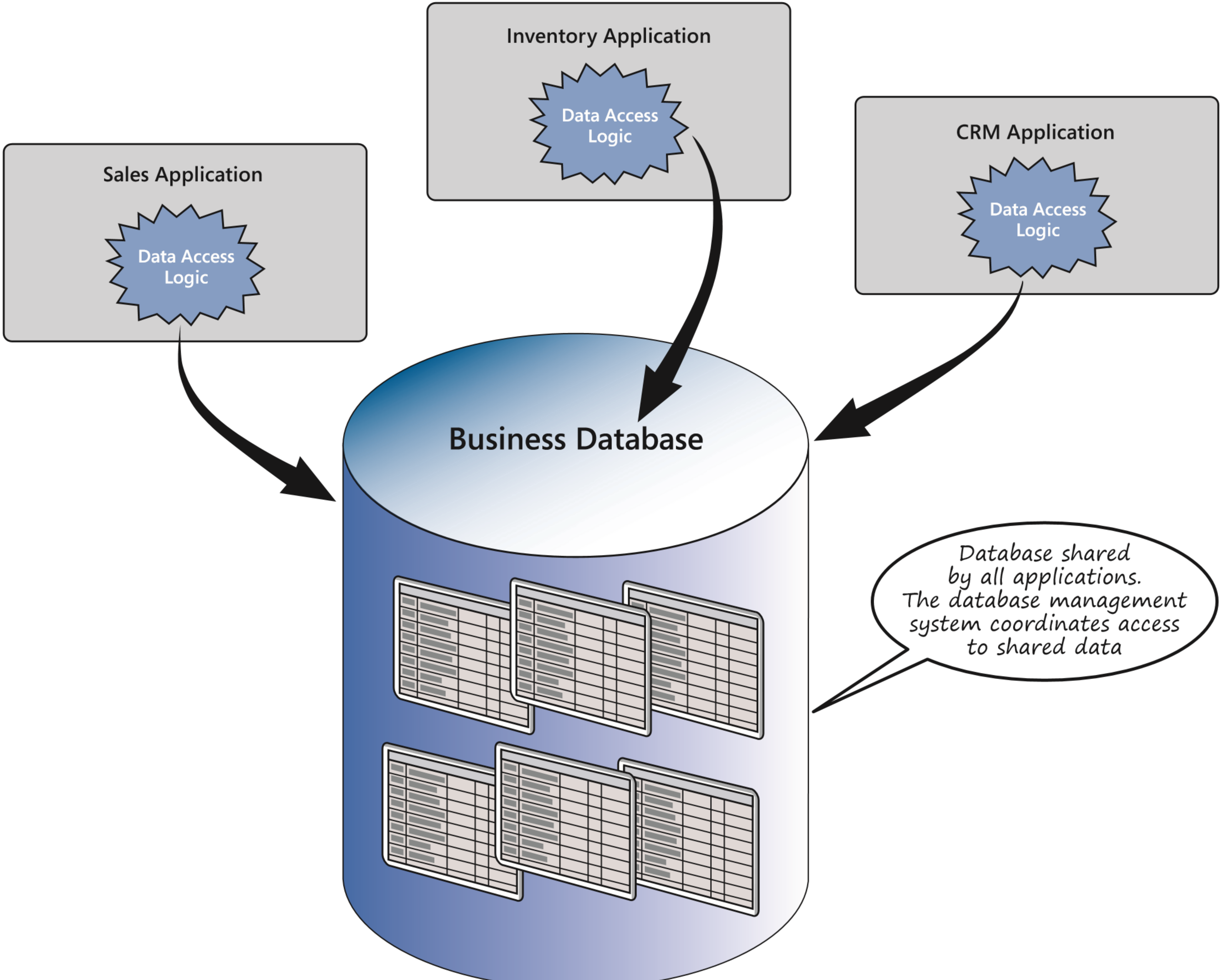
NOSQL BUSINESS DRIVERS

- Driving forces:
  - **v**ariety / flexibility
  - **v**olume / scalability
  - **v**elocity / performance
  - availability
- NoSQL answers:
  - schemaless
  - partitioning
  - replication
  - eventual consistency



VARIETY / FLEXIBILITY – RELATIONAL DBMS

- In theory, the relational model is extremely flexible and can model almost any type of data and relationships
- In practice, it can lead to solutions that **overemphasize** the **tabular way** in which the data is stored and queried using SQL
- The **impedance mismatch** between the structure of data in the database and the models used by most applications has an adverse effect on the productivity of developers
- Applications are **tightly coupled** to a database schema — once a relational schema has been fixed it can become difficult to change, especially as many applications depend on it


Inventory Application
Data Access Logic
CRM Application
Data Access Logic
Sales Application
Data Access Logic
Business Database
Database shared by all applications. The database management system coordinates access to shared data

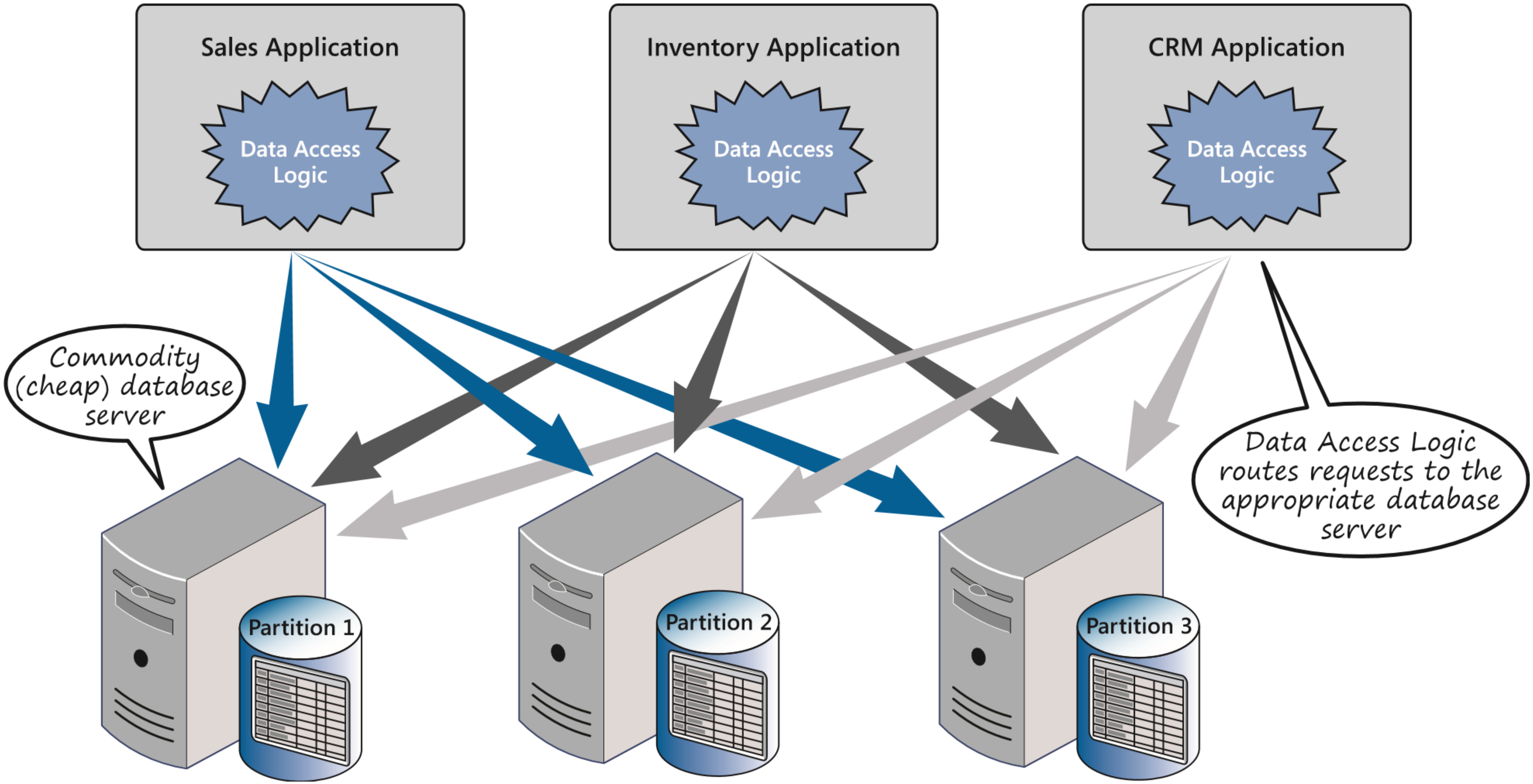
VARIETY / FLEXIBILITY – NOSQL DBMS

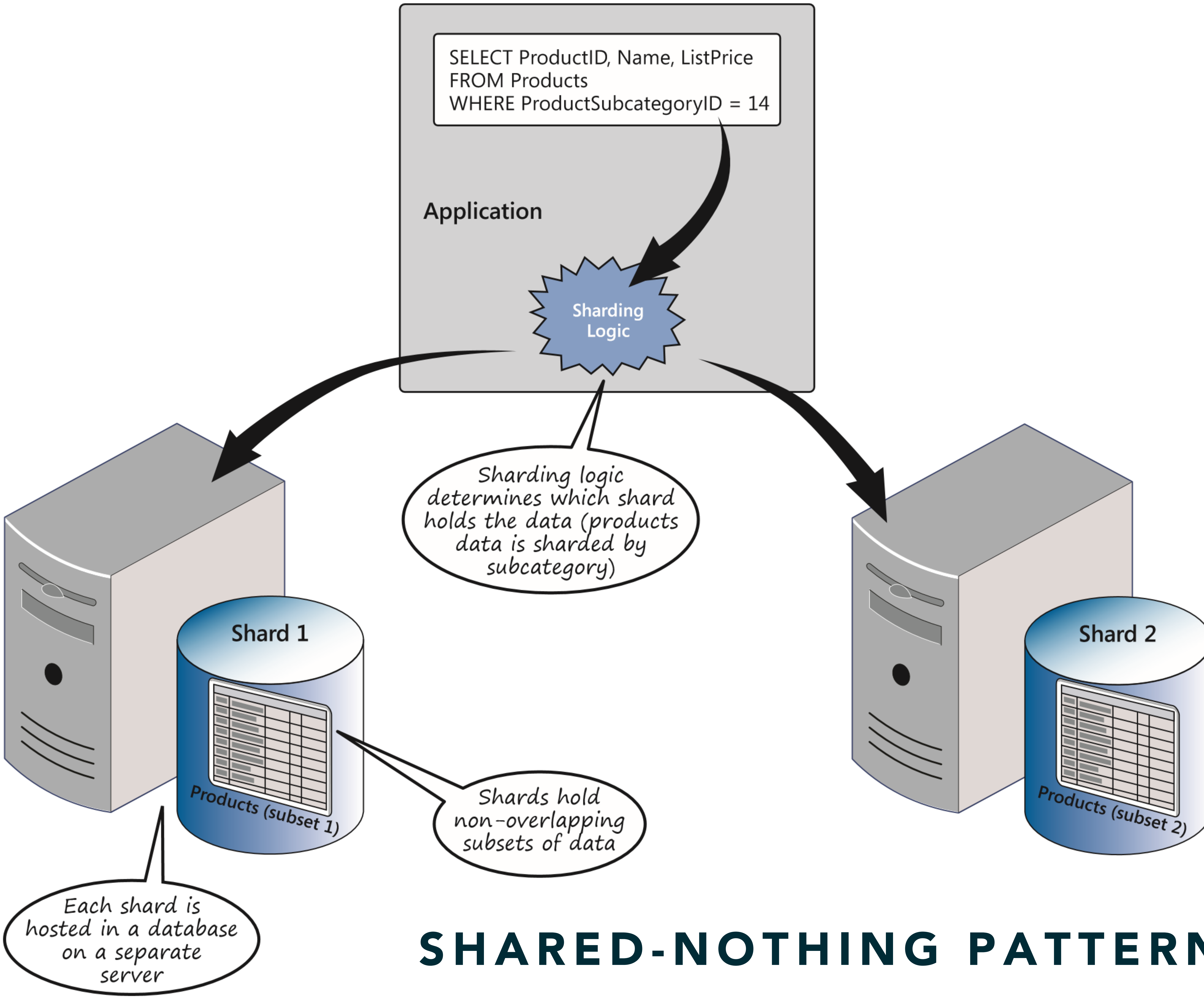
- NoSQL databases are **schemaless** — the responsibility for managing the structure of data has moved from the databases to the applications that use them
- The simplified APIs exposed by most NoSQL databases enable an application to store and retrieve data, but rarely impose any **restrictions** on data structure
- When business requirements change, the applications can **freely modify** the **structure** of the data that they store
- The database may end up holding a **non-uniform** set of data — requires a great deal of discipline from applications

VOLUME / SCALABILITY

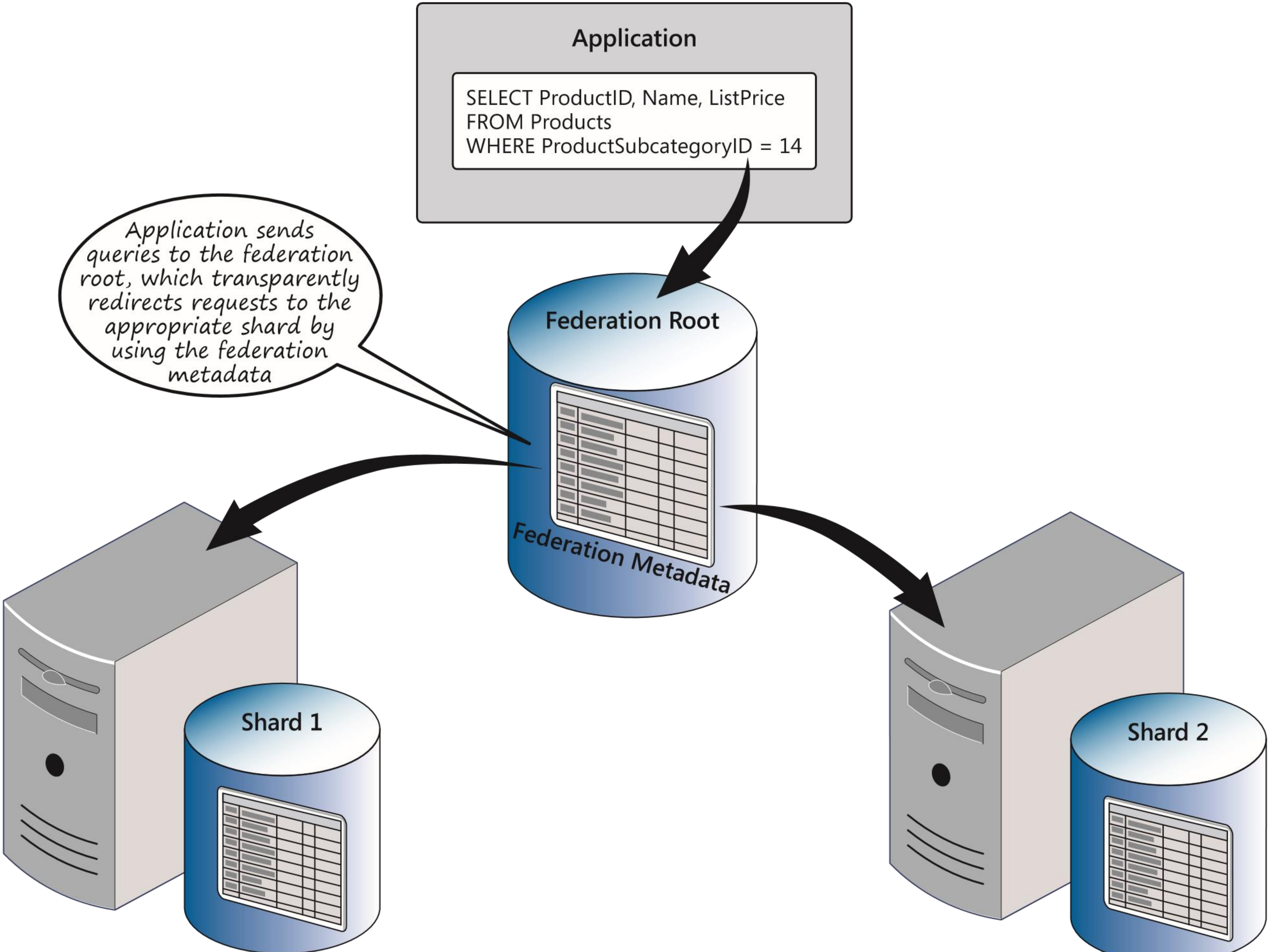
- Many systems make assumptions about the number of requests — what if the **volume** of traffic increases?
- **Vertical scaling** (scaling up) — adding resources to a single node in a system, e.g., purchasing a faster CPU or more memory to a single database server.
- **Horizontal scaling** (scaling out) — adding more nodes to a system, e.g., **partitioning** the data into a set (cluster) of smaller databases and running each database on a **separate commodity server**.

SCALING OUT USING **SHARDING** (HORIZONTAL PARTITIONING)





SHARED-NOTHING PATTERN

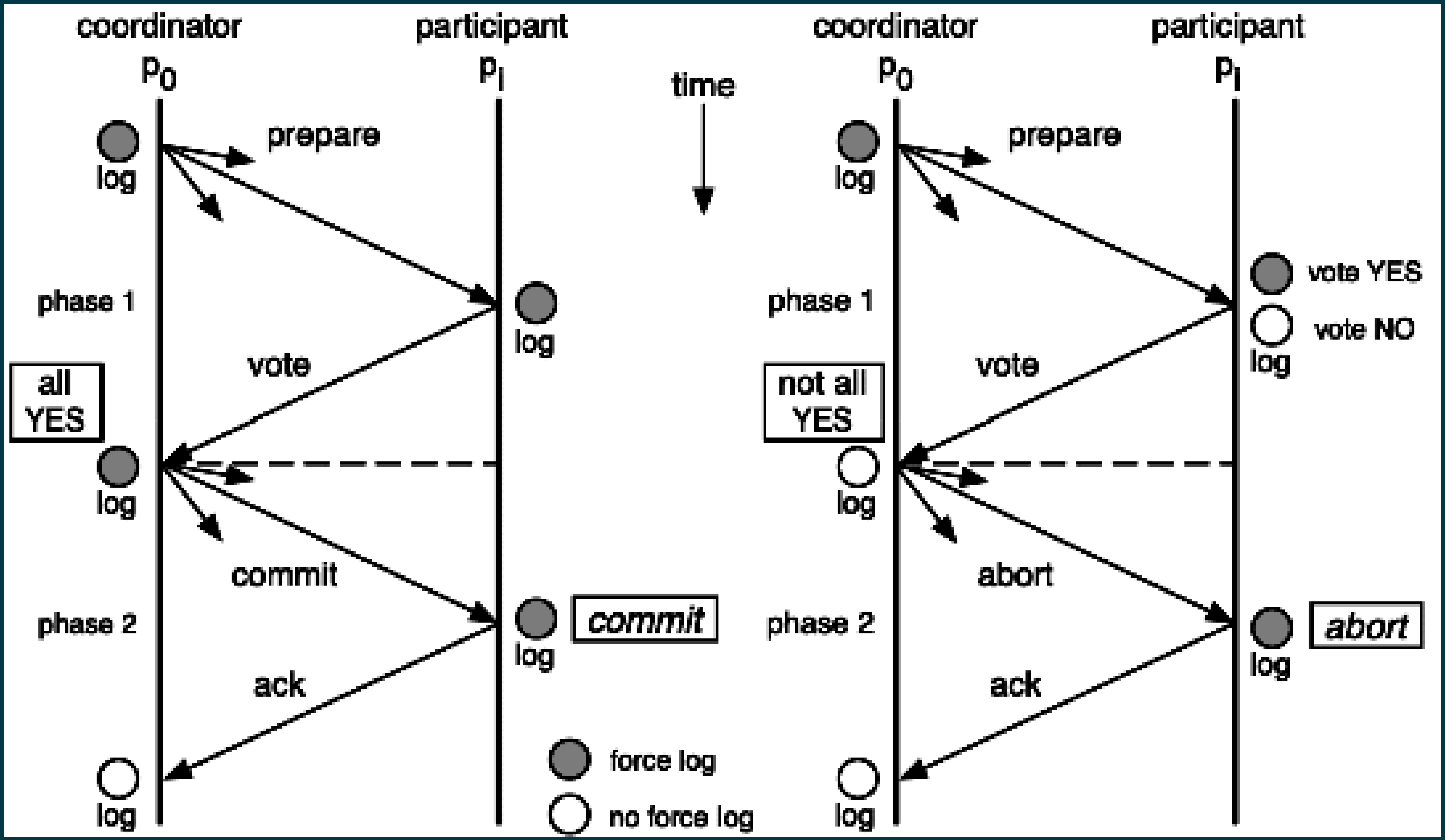


FEDERATION APPROACH

SCALABILITY — RDBMS

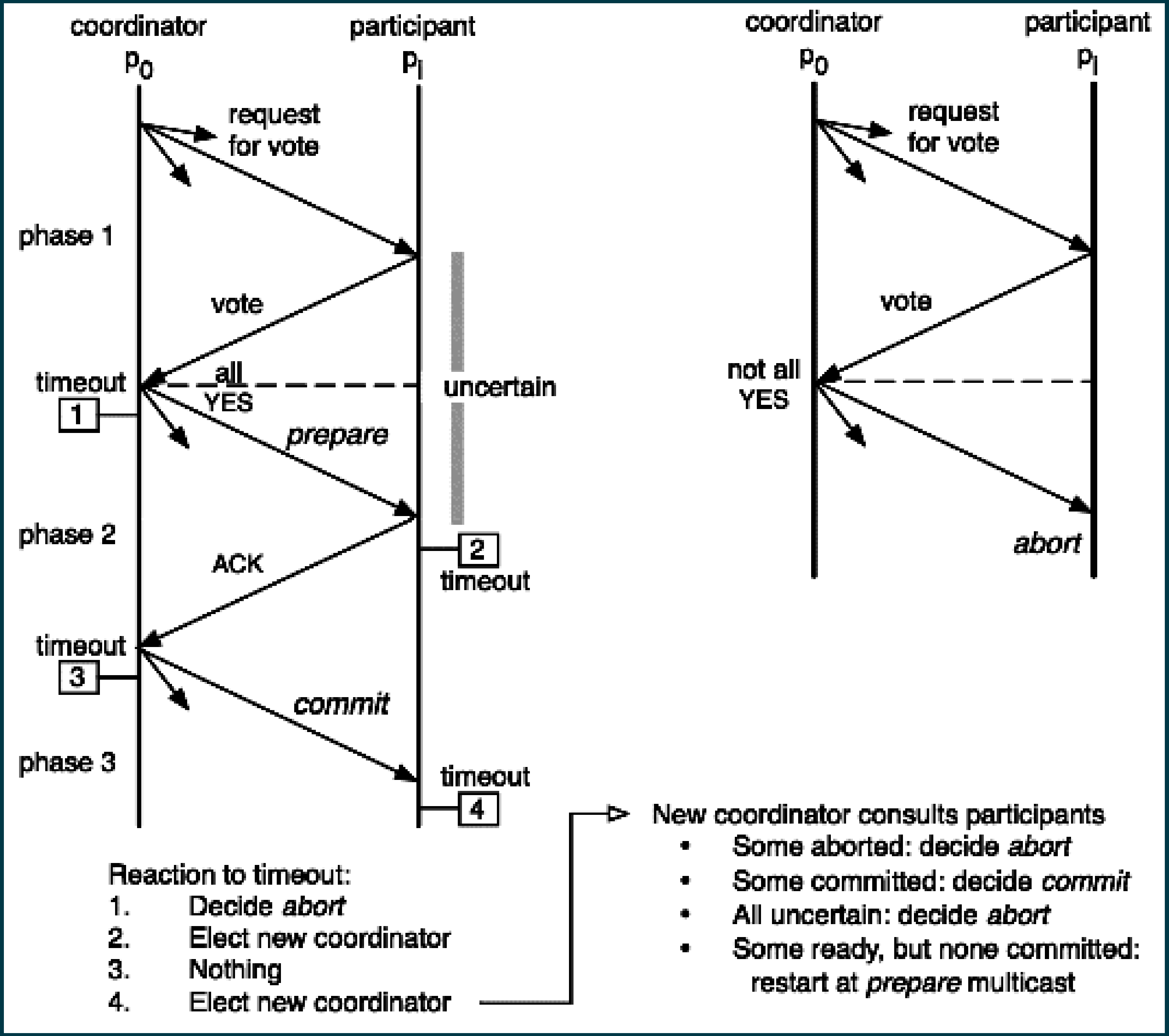
- Partitioning enables to handle **more users or data**, but can also have a detrimental effect on the **performance**:
  - **queries** that need to **join** data held in different shards,
  - **transactions** that update data spread across multiple shards.
- Relational databases place great emphasis on ensuring that data is **consistent — ACID transactions**.
- Local ACID transactions use **two-phase commit (2PC) algorithm** to achieve consistency.
- Implementing **distributed** ACID transactions across multiple databases requires careful coordination of **locking — two-phase commit** (2PC) **or three-phase commit (3PC) protocols**.
- Consistency comes at a price of **decreased performance**.

2PC PROTOCOL



Source: S. Krakowiak, Middleware Architecture with Patterns and Frameworks, 2009

3PC PROTOCOL



Source: S. Krakowiak, Middleware Architecture with Patterns and Frameworks, 2009

SCALABILITY — NOSQL

In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability.

BASE: AN ACID ALTERNATIVE — DAN PRITCHETT, EBAY

- NoSQL databases provide **eventual** rather than **immediate consistency** — **BASE** transactions:
  - **Basic Availability** — a guarantee that every request quickly receives some copy of data or an error,
  - **Soft state** — the state of the system may change over time, at times without any input (for eventual consistency).
  - **Eventual consistency** — the database may be momentarily inconsistent but will be consistent eventually.

SCALABILITY — NOSQL

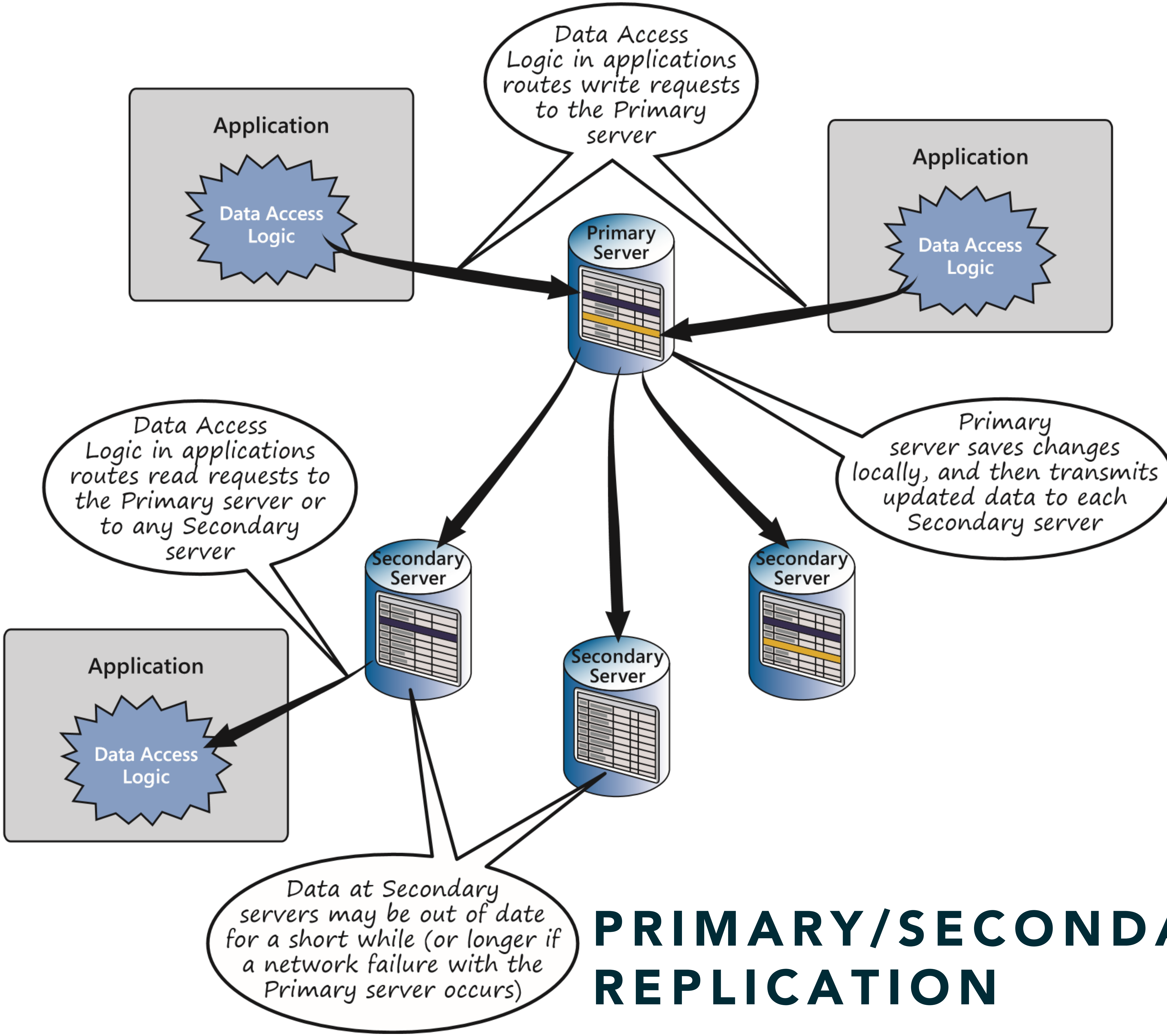
- BASE transactions philosophy:
  - focus on **throughput** and **availability**, **not consistency**
  - **never block** a write, even at the risk of being out of sync
  - be **optimistic** — assume that **eventually** all nodes will catch up and **become consistent**.
  - keep things **simple** and **fast** by **avoiding locking**
- **ACID** and **BASE** are not strict oppositions — they lie on a **continuum** and you can decide how close you want to be to one end of the continuum or the other according to your priorities.

AVAILABILITY — RDBMS

- The relational model does not address the issue of **availability**, although many relational database management systems offer high availability (for a price).
- Availability can be implemented by maintaining a **copy** of the **database** hosted by a separate **failover server** that is brought online if the primary server fails.
- Increased **availability** complicates **consistency** — maintaining multiple copies of data requires ensuring that each copy is modified consistently.

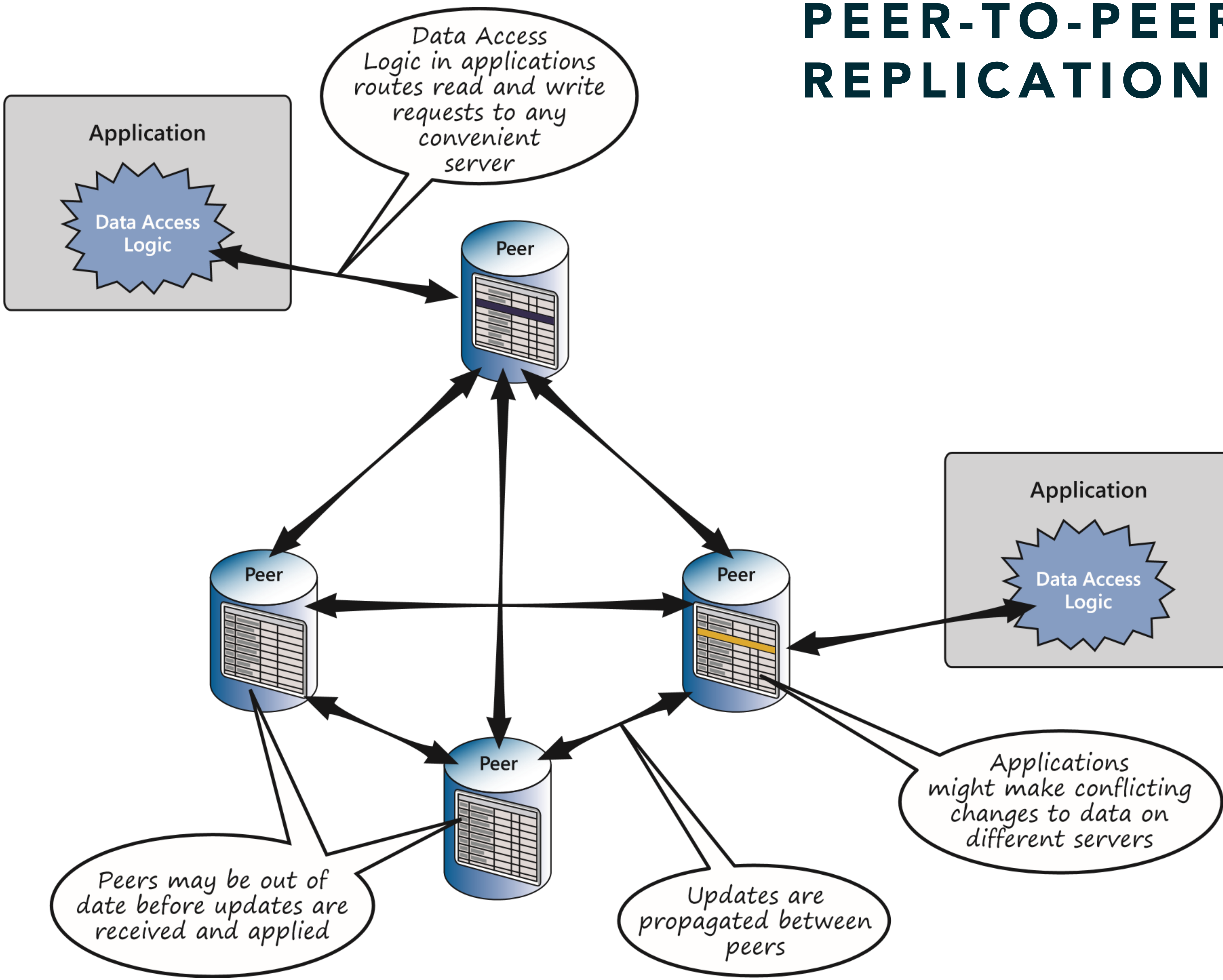
AVAILABILITY — NOSQL DBMS

- Most NoSQL databases are **explicitly designed around high-availability**, and their architecture is geared towards maintaining high performance and availability.
- Absolute **consistency** is a **lower priority**, as long as data is not lost and **eventually** becomes **consistent**.
- NoSQL databases commonly implement one of **two replication schemes** for ensuring that data is available:
  - **primary/secondary** replication (master/subordinate replication),
  - **peer-to-peer** replication.



PRIMARY/SECONDARY REPLICATION

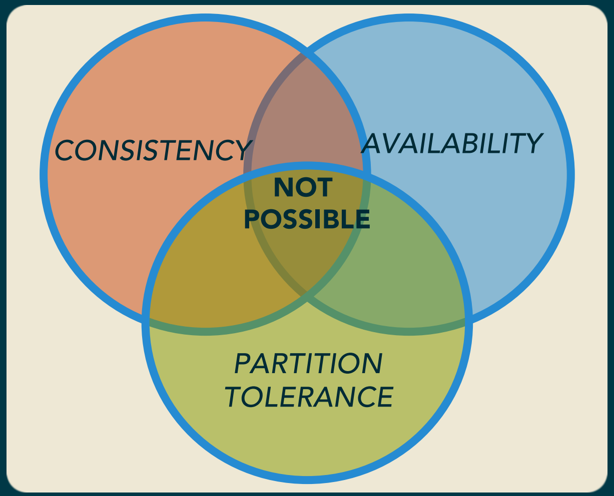
PEER-TO-PEER REPLICATION



AVAILABILITY — NOSQL DBMS

- **Replication** improves **performance** and **availability**, but at the increased risk of **inconsistency** between replicated data.
- Many NoSQL databases include **mechanisms** that help to **reduce the likelihood of inconsistent** data:
  - read and write **quorums** — subset of the servers in a replication cluster must agree on the value of a data item.
  - data **versioning** — **optimistic locking scheme**:
    1. **reading a data item together with version information,**
    2. **attempting to update the data item — re-reading the version information,**
    3. **if the version information is unchanged — saving modified data back to the database together with new version information,**
    4. **if the version information is different — retrieving the latest value of the data from the database and going back to step 2.**
    - Version information is stored using a version vector (a kind of vector clock)
    - Versioning conflicts are resolved using configurable rules (e.g., siblings or last-write-wins)

CAP THEOREM



Though its desirable to have Consistency, High-availability and Partition-tolerance in every system, unfortunately no system can achieve all three at the same time.

- **Consistency** — all database clients see the same data, even with concurrent updates.
- **Availability** — all database clients are able to immediately access some version of the data.
- **Partition tolerance** — operations will complete, even if individual components are unavailable.

PRESENTATION OUTLINE

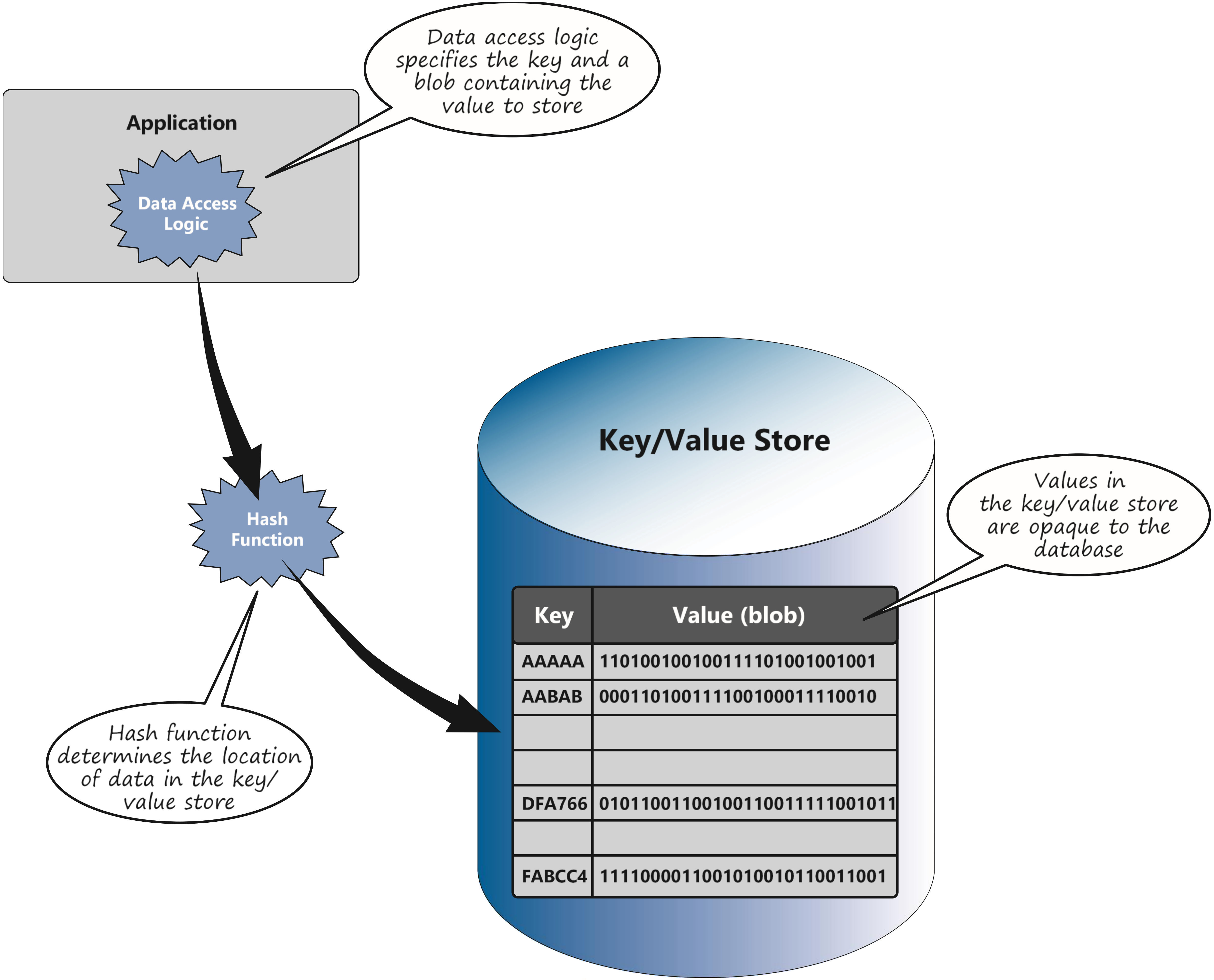
- Motivation
- Relational Databases
  - JDBC — Java Database Connectivity
  - ORM — Object-Relational Mapping
  - JPA — Java Persistence API
- NoSQL Databases
  - Challenges and common features
  - **Key-value stores: Redis, Riak**
  - **Document-oriented databases: MongoDB, CouchDB**
  - **Graph databases: Neo4J**
  - **Column-oriented databases: HBase, Cassandra**

NOSQL DATABASES

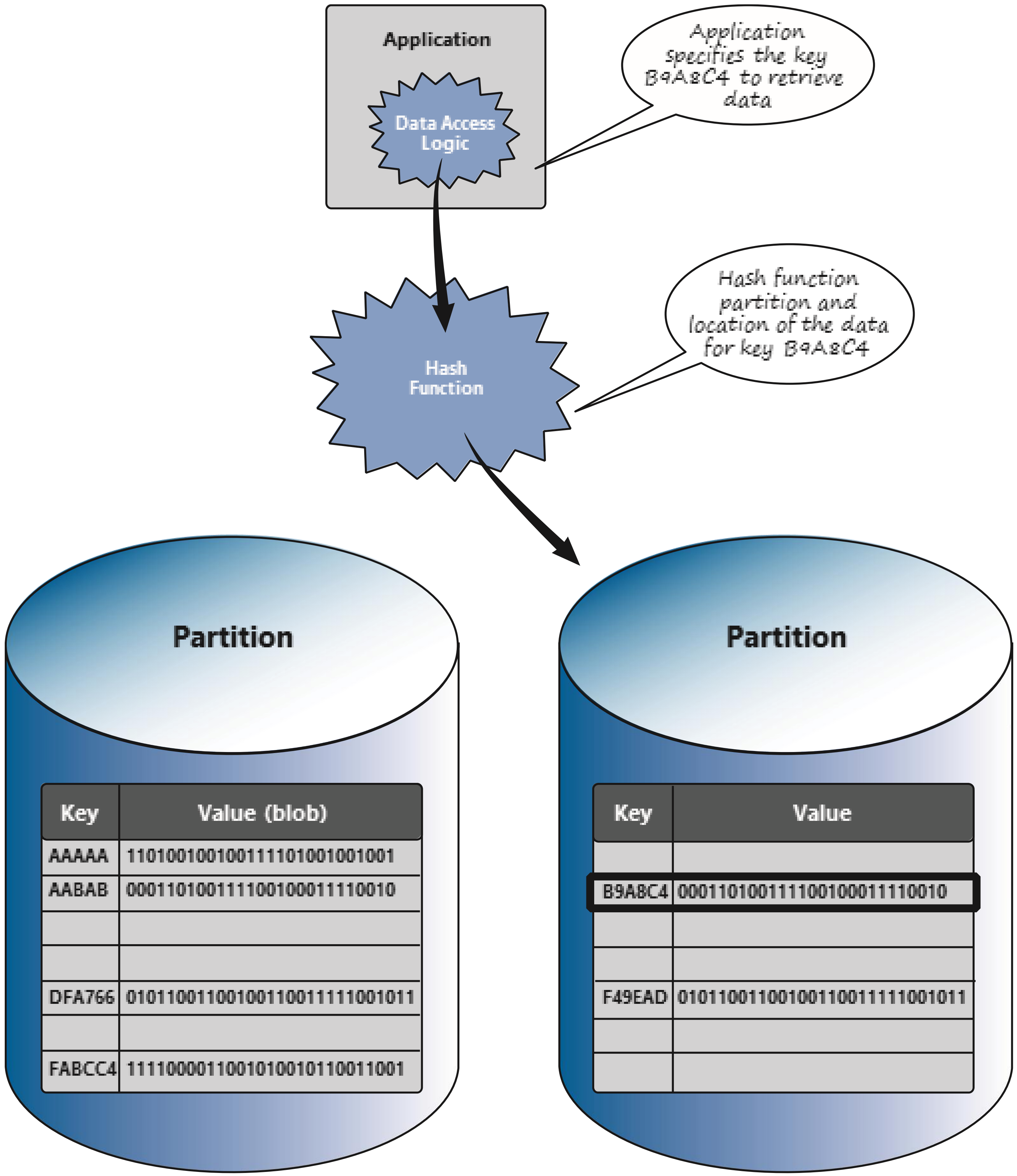
- The software industry has attempted to **categorize** NoSQL databases into a small set of functional areas:
  - key-value stores
  - document databases
  - column-family databases
  - graph databases.
- Some NoSQL databases fit naturally into a single category, while others include functionality that spans few categories.

KEY-VALUE STORES

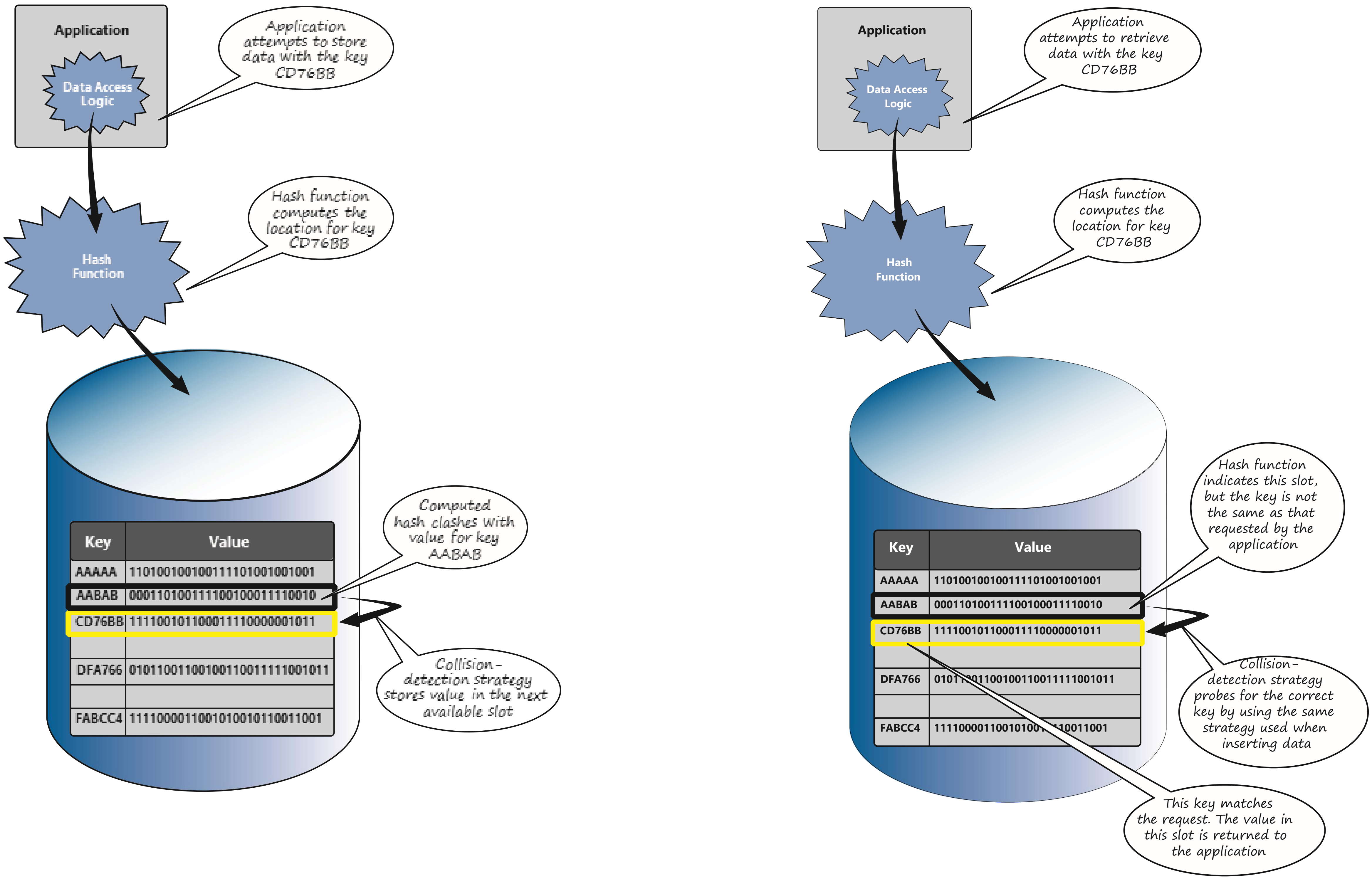
- A **key-value store** implements the **simplest** of the NoSQL storage mechanisms — a large **hash table**.
- The values deposited in a key-value store are **BLOBs**, they are **opaque** to the database management system.
- Key-value stores have no query language, they support only simple **query**, **insert**, and **delete** operations — the keys provide the only means of access to the data values.
- In most implementations, reading or writing a single value is an **atomic** operation.



Data access logic specifies the key and a blob containing the value to store
Application
Data Access Logic
Hash Function
Hash function determines the location of data in the key/ value store
Key/Value Store
Values in the key/value store are opaque to the database
Key
Value (blob)
AAAAA
1101001001001111010010010001
AABAB
0001101001111001000111110010
DFA766
0101100110010011001111001011
FABCC4
1111000011001010010110011001



Application
Data Access Logic
Application specifies the key B9A8C4 to retrieve data
Hash Function
Hash function partition and location of the data for key B9A8C4
Partition
Key
Value (blob)
AAAAA
110100100100111101001001001
AABAB
000110100111100100011110010
DFA766
01011001100100110011111001011
FABCC4
1111000011001010010110011001
Partition
Key
Value
B9A8C4
000110100111100100011110010
F49EAD
01011001100100110011111001011



Application
Data Access Logic
Application attempts to store data with the key CD76BB
Hash Function
Hash function computes the location for key CD76BB
Key
Value
AAAAA
110100100100111101001001001
AABAB
000110100111100100011110010
CD76BB
111100101100011110000001011
DFA766
010110011001001100111110010 11
FABCC4
111100001100101001011001 1001
Computed hash clashes with value for key AABAB
Collision-detection strategy stores value in the next available slot
Application
Data Access Logic
Application attempts to retrieve data with the key CD76BB
Hash Function
Hash function computes the location for key CD76BB
Key
Value
AAAAA
110100100100111101001001001
AABAB
000110100111100100011110010
CD76BB
111100101100011110000001011
DFA766
FABCC4
Hash function indicates this slot, but the key is not the same as that requested by the application
Collision-detection strategy probes for the correct key by using the same strategy used when inserting data
This key matches the request. The value in this slot is returned to the application

KEY-VALUE STORES

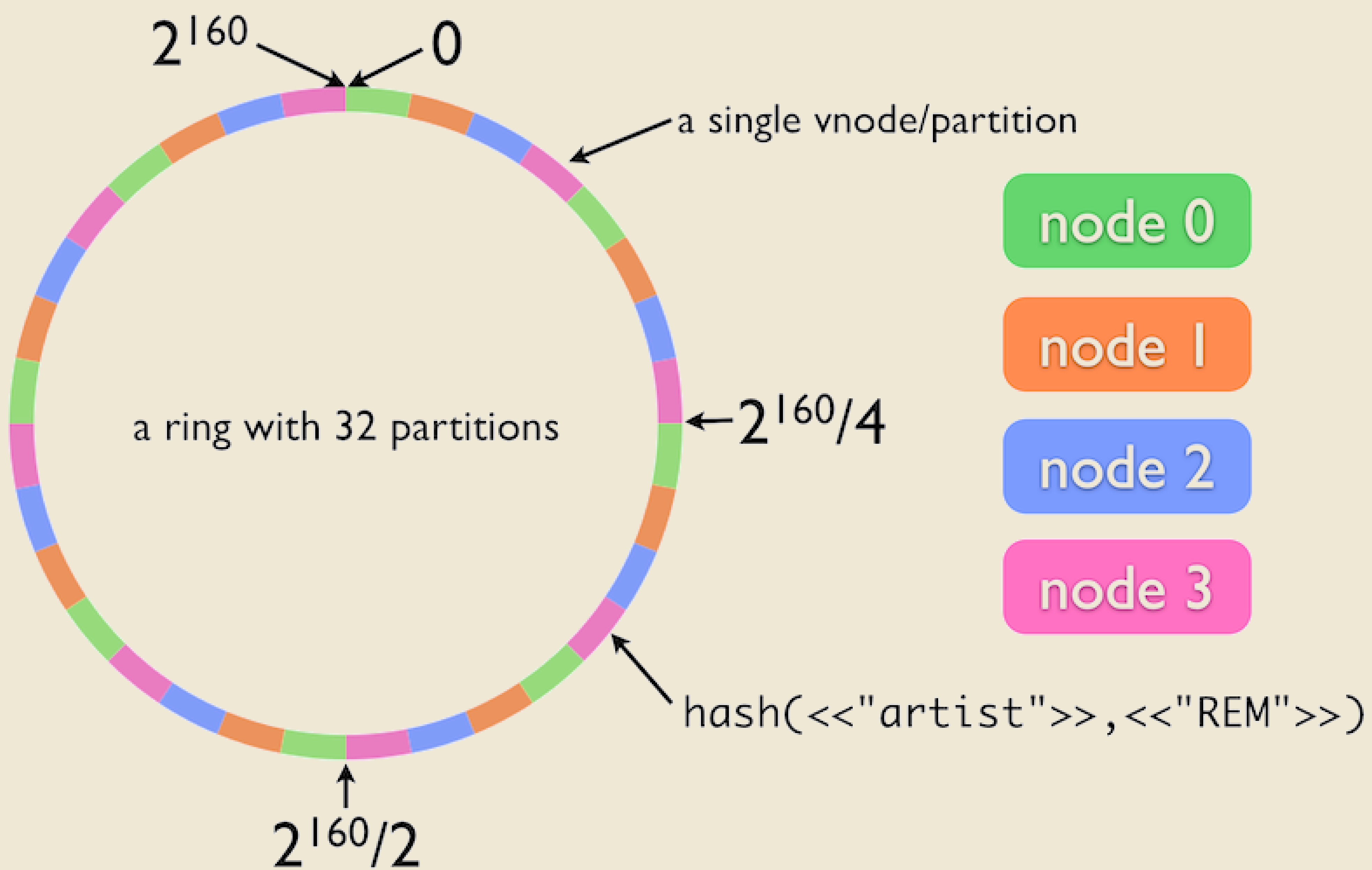
- Key-value stores focus on the ability to store and retrieve data rather than the **structure** of that data.
- Most implementations are very **quick** and **efficient**, lending themselves to **fast scalable** applications that need to read and write **large amounts of data**.
- Useful when the data are **not highly related** — e.g. in a web application for storing users' session data.
- With little or **no indexes** and **scanning capabilities**, key-value stores do not allow to perform complex queries on your data (other than basic CRUD operations).

KEY-VALUE STORES
RIAK

riak

- **Availability** — Riak writes to and reads from **multiple servers** to offer data availability even when hardware or the network itself are experiencing **failure** conditions.
- **Masterless** — Requests are not held hostage to a specific server in the cluster that may or may not be available.
- **Scalability** — Riak automatically distributes data around the cluster (**consistent hashing**) and yields a near-linear performance increase as you add capacity.
- **Tunable CAP properties** — eventual to strong consistency.
- Values are accessible through simple **RESTful API** through HTTP (GET, PUT, DELETE) — flexibility for a web system.

KEY-VALUE STORES
RIAK RING



KEY-VALUE STORES
REDIS

- **Unique data model** — values can contain more complex data types: strings, lists, sets, hashes; Redis is not a plain key-value store, actually it is a **data structures server**.

- **In-memory** — high **performance** is achieved with the limitation of data sets that can't be larger than memory.

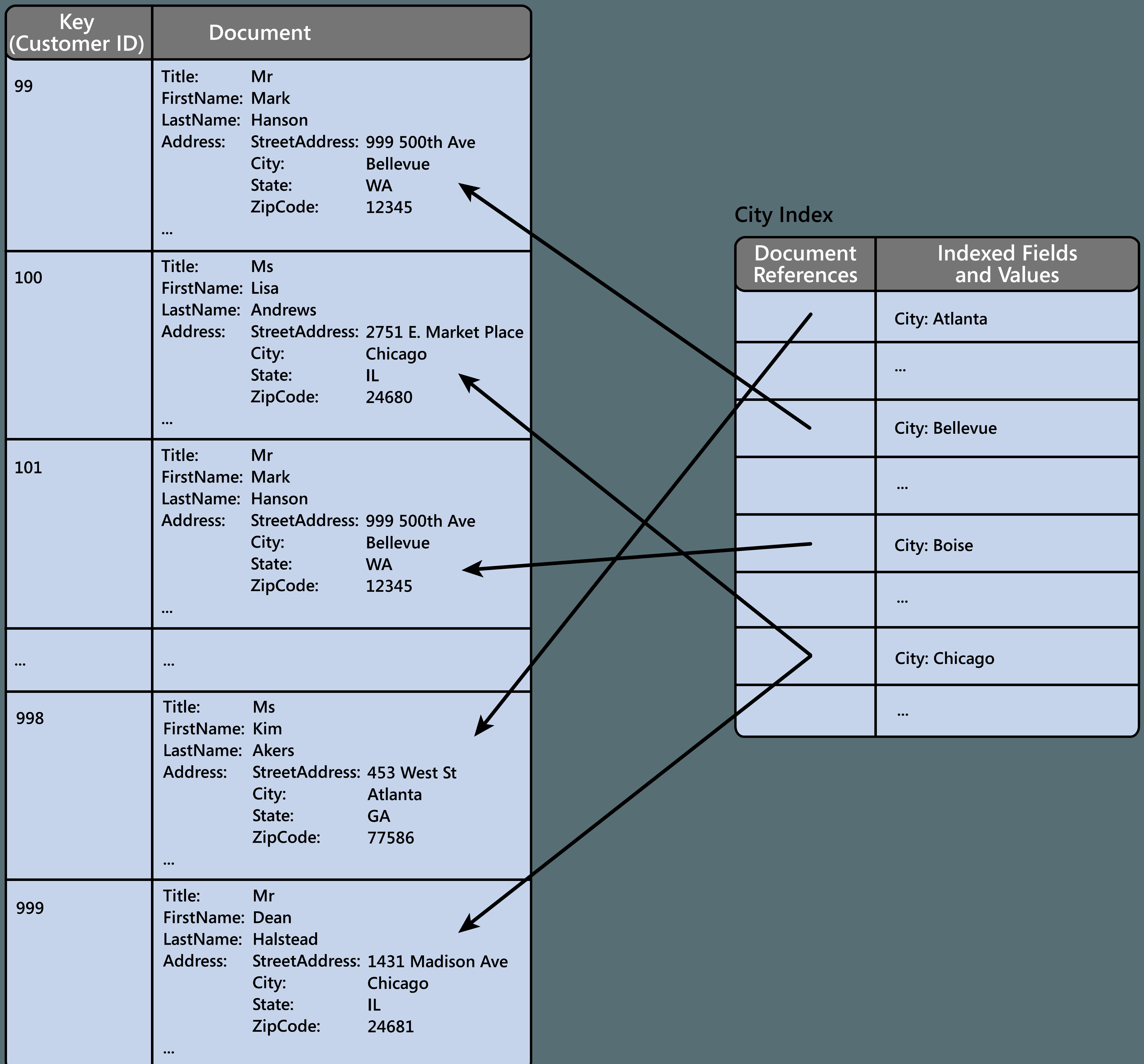
- **Optional durability** — the dataset can be asynchronously transferred from memory to disk from time to time.

- **Master-slave asynchronous replication** — slave servers are exact copies of master server and can serve read queries.

DOCUMENT DATABASES

- A document database is similar in concept to a key-value store except that the values are **documents** (collections of named **fields** and **values**).
- The data in the **fields** in a document can be encoded in a variety of ways, including **XML**, **YAML**, **JSON**, **BSON** etc.
- The fields in the documents are **exposed** to the database management system, enabling an application to **query** and **filter** data by using the values in these fields.
- **In-place updates** enable an application to **modify** the values of **specific fields** without rewriting the entire document.

Row Key	Document
1001	OrderDate: 06/06/2013 OrderItems: ProductID: 2010 Quantity: 2 Cost: 520 ProductID: 4365 Quantity: 1 Cost: 18 OrderTotal: 1058 Customer ID: 99 ShippingAddress: StreetAddress: 999 500th Ave City: Bellevue State: WA ZipCode: 12345
1002	OrderDate: 07/07/2013 OrderItems: ProductID: 1285 Quantity: 1 Cost: 120 OrderTotal: 120 Customer ID: 220 ShippingAddress: StreetAddress: 888 W. Front St City: Boise State: ID ZipCode: 54321


Key (Customer ID)
Document
99
Title: Mr
FirstName: Mark
LastName: Hanson
Address: StreetAddress: 999 500th Ave
City: Bellevue
State: WA
ZipCode: 12345
...
100
Title: Ms
FirstName: Lisa
LastName: Andrews
Address: StreetAddress: 2751 E. Market Place
City: Chicago
State: IL
ZipCode: 24680
...
101
Title: Mr
FirstName: Mark
LastName: Hanson
Address: StreetAddress: 999 500th Ave
City: Bellevue
State: WA
ZipCode: 12345
...
...
...
998
Title: Ms
FirstName: Kim
LastName: Akers
Address: StreetAddress: 453 West St
City: Atlanta
State: GA
ZipCode: 77586
...
999
Title: Mr
FirstName: Dean
LastName: Halstead
Address: StreetAddress: 1431 Madison Ave
City: Chicago
State: IL
ZipCode: 24681
...
City Index
Document References
Indexed Fields and Values
City: Atlanta
...
City: Bellevue
...
City: Boise
...
City: Chicago
...

MONGO DB

- **Agility** — document data model makes it easy for you to store data of any structure and dynamically modify the schema.
- **High availability** — implemented using primary/secondary replication.
- **Scalability** — automatic sharding distributes data across a cluster of machines.
- **High performance** — secondary indexes provide fast, fine-grained access to data, including fully consistent indexes on any field (also geospatial and text).

MONGO DB

- Documents are organized into collections
- JSON
  - Document format
  - Query format

MONGO DB: INSERT DOCUMENT

```
db.restaurants.insert(
   {
      "address": {
         "street": "2 Avenue",
         "zipcode": "10075",
         "building": "1480",
         "coord": [-73.9557413, 40.7720266]
      },
      "borough": "Manhattan",
      "cuisine": "Italian",
      "grades": [
         {
            "date": ISODate("2014-10-01T00:00:00Z"),
            "grade": "A",
            "score": 11
         },
         {
            "date": ISODate("2014-01-16T00:00:00Z"),
            "grade": "B",
            "score": 17
         }
      ],
      "name": "Vella",
      "restaurant_id": "41704620"
   }
)
```

MONGO DB: FIND DOCUMENT

```
// get all documents in collection
db.restaurants.find()

// get documents by field value
db.restaurants.find({ "borough": "Manhattan" })

// get documents by value of field of nested object
db.restaurants.find({ "address.zipcode": "10075" })

// use operator (greater than)
db.restaurants.find({ "grades.score": { $gt: 30 } })

// combine conditions (AND)
db.restaurants.find({ "cuisine": "Italian", "address.zipcode": "10075" })

// combine conditions (OR)
db.restaurants.find(
   { $or: [{ "cuisine": "Italian" }, { "address.zipcode": "10075" }] }
)

// sort documents (1 and -1 denote ascending and descending orders, respectively)
db.restaurants.find().sort({ "borough": 1, "address.zipcode": 1 })
```

MONGO DB: UPDATE DOCUMENT

```
// update the first found document
db.restaurants.update(
    // condition
    { "name": "Juni" },
    // values to set
    {
        $set: { "cuisine": "American (New)" },
        $currentDate: { "lastModified": true }
    }
)

// update all matching documents
db.restaurants.update(
  { "address.zipcode": "10016", cuisine: "Other" },
  {
      $set: { cuisine: "Category To Be Determined" },
      $currentDate: { "lastModified": true }
  },
  { multi: true }
)
```

MONGO DB: REMOVE DOCUMENT

```
// remove all matching documents
db.restaurants.remove({ "borough": "Manhattan" })

// remove just one matching document
db.restaurants.remove({ "borough": "Queens" }, { justOne: true })

// remove all documents
db.restaurants.remove({})

// drop collection
db.restaurants.drop()
```

COLUMN-FAMILY DATABASES

- Key-value stores and document databases are very **row focused** — optimized towards retrieving **complete entities** that match one or more criteria.
- Column-family databases allow to retrieve data from a subset of fields across a collection of documents.
- **Column-family** — a logically related collection of columns that hold the data for a set of entities.

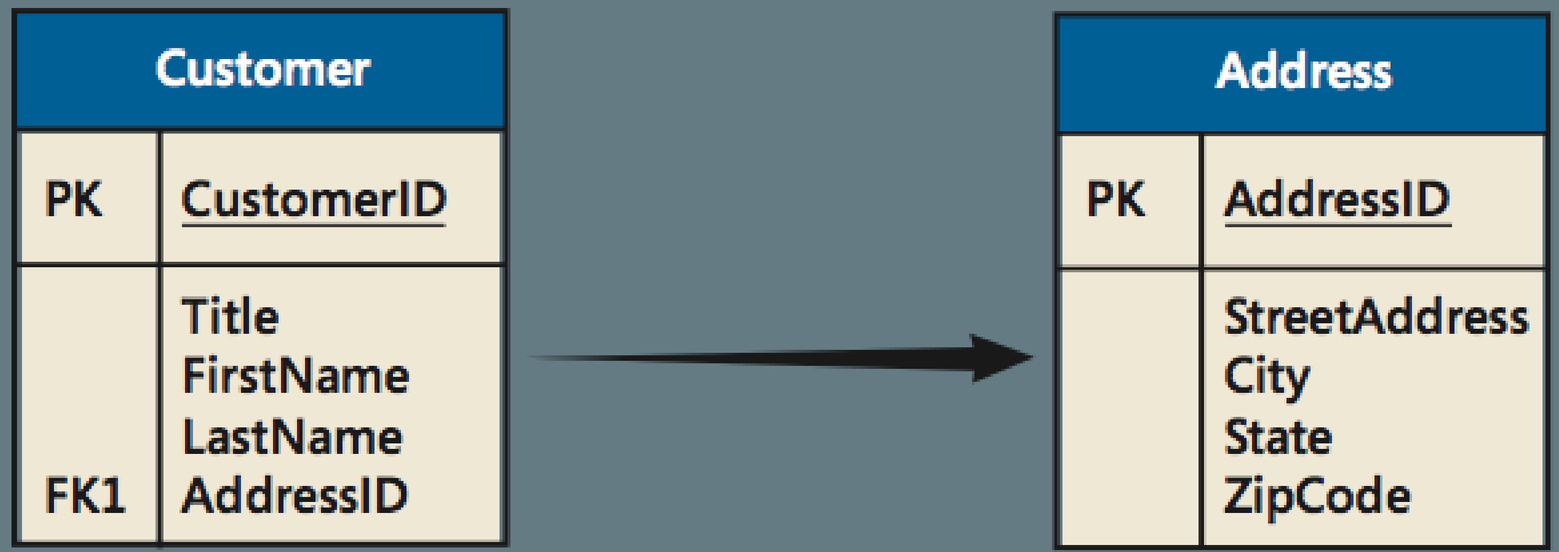
COLUMN-FAMILY DATABASES VERSUS RELATIONAL DATABASES

- **Like** relational databases:
  - a column-family DB organizes its data into rows and columns
  - column families give flexibility to perform complex queries
- **Unlike** a relational database:
  - the names of columns do not have to be **static**,
  - **structure** of the information can **vary** from row to row — columns do not have to conform to a rigidly defined schema
- The real power of a column-family database lies in its **denormalized** approach to **structuring sparse data**.

COLUMN-FAMILY DATABASES PERFORMANCE AND GENERALITY

- Column-family databases are designed to hold **vast amounts** of data (hundreds of millions, or billions of rows containing hundreds of columns).
- A well-designed column-family database is inherently **faster** and **more scalable** than a relational database that holds an equivalent volume of data.
- Performance comes at a price of **decreased generality** — column-families are designed to **optimize** the most common queries.

A RELATIONAL MODEL



Customer Table

CustomerID	Title	FirstName	LastName	AddressID
1	Mr	Mark	Hanson	500
2	Ms	Lisa	Andrews	501
3	Mr	Walter	Harp	500

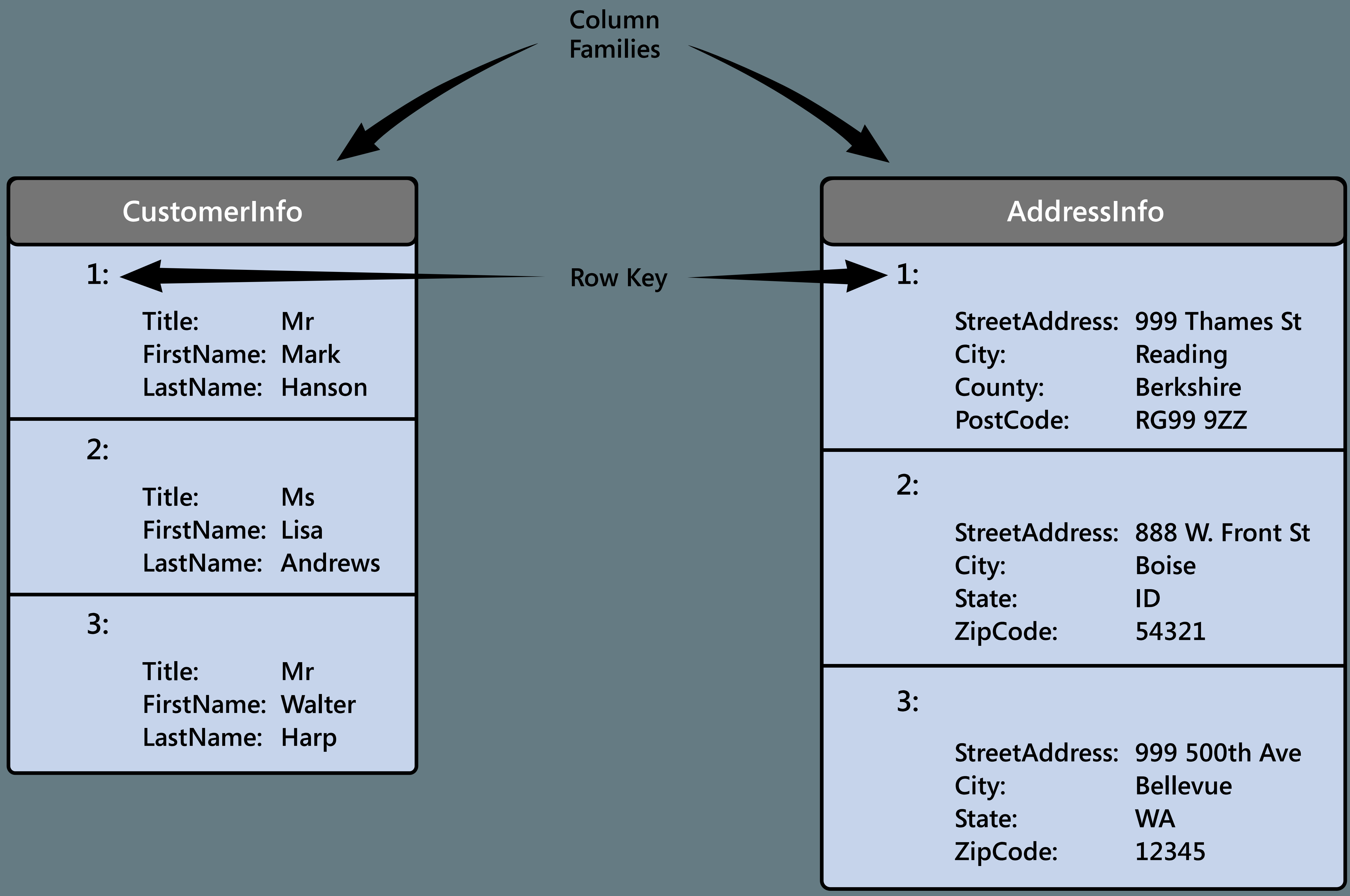
Address Table

AddressID	StreetAddress	City	State	ZipCode
500	999 500th Ave	Bellevue	WA	12345
501	888 W. Front St	Boise	ID	54321

A COLUMN MODEL

Row Key	Column Families			
CustomerID	CustomerInfo		AddressInfo	
1	CustomerInfo:Title CustomerInfo:FirstName CustomerInfo:LastName	Mr Mark Hanson	AddressInfo:StreetAddress AddressInfo:City AddressInfo:County AddressInfo:PostCode	999 Thames St Reading Berkshire RG99 922
2	CustomerInfo:Title CustomerInfo:FirstName CustomerInfo:LastName	Ms Lisa Andrews	AddressInfo:StreetAddress AddressInfo:City AddressInfo:State AddressInfo:ZipCode	888 W. Front St Boise ID 54321
3	CustomerInfo:Title CustomerInfo:FirstName CustomerInfo:LastName	Mr Walter Harp	AddressInfo:StreetAddress AddressInfo:City AddressInfo:State AddressInfo:ZipCode	999 500th Ave Bellevue WA 12345

PHYSICAL STORAGE MAY VARY



COLUMN-FAMILY DATABASES
CASSANDRA

- **Decentralized** — no single points of failure, no network bottlenecks, every node in the cluster is identical.
- **Fault Tolerant** — data is replicated to multiple nodes; failed nodes can be replaced with no downtime.
- **Scaling** — read and write throughput both increase linearly as new machines are added
- **Hadoop** integration, with **MapReduce** support.

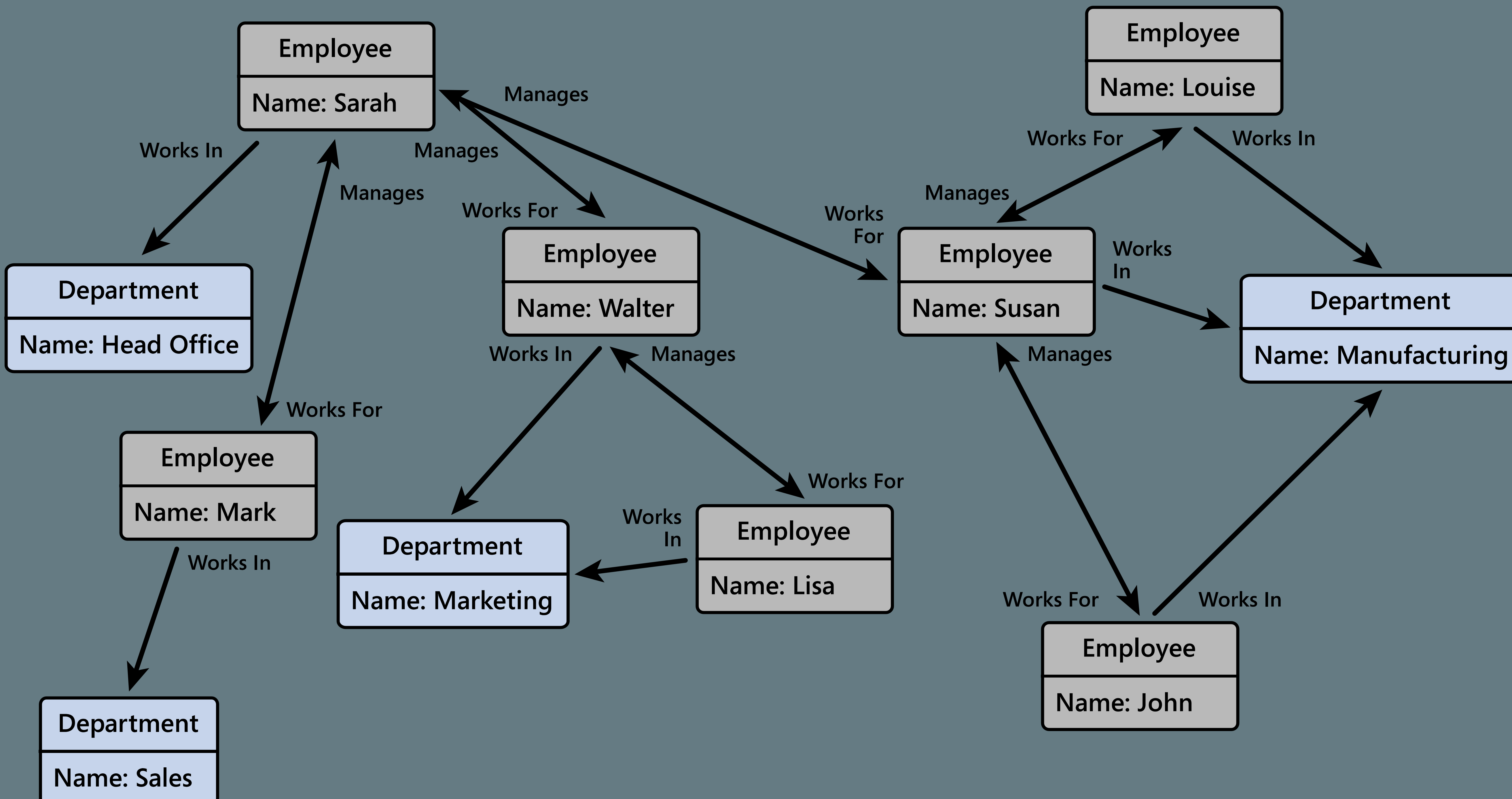
QUERYING CASSANDRA

- Cassandra Query Language (CQL) is SQL-inspired query language for Cassandra DB

```
SELECT name, occupation FROM users WHERE userid IN (199, 200, 207);
SELECT JSON name, occupation FROM users WHERE userid = 199;
SELECT name AS user_name, occupation AS user_occupation FROM users;

SELECT time, value
FROM events
WHERE event_type = 'myEvent'
  AND time > '2011-02-03'
  AND time <= '2012-01-01'

SELECT COUNT (*) AS user_count FROM users;
```

GRAPH DATABASES

- The main focus in **graph** databases is on the **relationships** between the stored entities.
- A graph database stores two types of information:
  - **nodes** that you can think of as instances of **entities**,
  - **edges** which specify the **relationships** between nodes.
- Nodes and edges can both have **properties** that provide information; additionally, edges can have a **direction** indicating the nature of the relationship.


Employee
Name: Sarah
Manages
Manages
Works In
Manages
Works For
Department
Name: Head Office
Works For
Employee
Name: Walter
Works For
Employee
Name: Susan
Employee
Name: Louise
Works For
Works In
Manages
Works In
Department
Name: Manufacturing
Works In
Manages
Manages
Works For
Employee
Name: Mark
Works In
Department
Name: Marketing
Works In
Employee
Name: Lisa
Works For
Works For
Works In
Employee
Name: John
Department
Name: Sales

GRAPH DATABASES
COMMON APPLICATIONS

- **Social Networking** — a typical social networking database contains a significant number of contacts together with the connections that these contacts have with each other.
- **Calculating Routes** — a graph database helps in solving complex routing problems that would require considerable resources to resolve by using an algorithmic approach.
- **Generating Recommendations** — a graph database as a recommendations engine by storing information about which products are frequently purchased together.

GRAPH DATABASES
RETRIEVING DATA

- All graph databases provide a means to walk through a set of connected nodes based on their relationships: **imperative** or **declarative approach** to querying.

- All queries against a graph database require a **node** (or a collection of nodes) as a **starting point**.

- Most graph databases allow to define **indexes** over **properties** to enable the database server to quickly locate a **node** or **relationship.**

GRAPH DATABASES
NEO4J

- Neo4j is an open-source, **ACID transaction** compliant graph database implemented in Java and Scala.

- **Cypher Query Language** — a declarative, SQL-inspired language for describing patterns in graphs — allows us to describe **what** we want from a graph database without requiring us to describe exactly **how** to do it:

```
MATCH (you {name:"You"})-[:FRIEND]->(yourFriends)
RETURN you, yourFriends
```

```
MATCH (ch:Person { name:'Charlie Sheen' })-[:ACTED_IN]-(m:Movie)
RETURN m
```

CONCLUSIONS

- Relational databases provide an excellent mechanism for storing and accessing data in a very **generalized** manner but this generalization can also be a weakness.
- Relational model might not always provide the optimal way to meet **specific data access requirements**.
- NoSQL databases might not be as **comprehensive** as a relational ones, but their focus on a **well-defined tasks** enables them to be highly optimized for those tasks.
- Understanding the **strength** and **weaknesses** of different NoSQL databases allows to match the application requirements.

REFERENCES

- *Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence* — Douglas McMurtry, Andrew Oakley, John Sharp, Mani Subramanian, Hanz Zhang, Microsoft, Inc., 2013 https://www.microsoft.com/en-us/download/details.aspx?id=40327

- *Seven Databases in Seven Weeks: A Guide to Modern Databases and NoSQL Movement* — Eric Redmond and Jim R. Wilson, Pragmatic Programmers, LLC, 2012

- *Pro JPA 2: Mastering the Java Persistence API, Mike Keith and Merrick Schnicariol, Apress, 2009*

- *Java Platform, Enterprise Edition: The Java EE Tutorial* https://docs.oracle.com/javaee/7/tutorial

- *Java Persistence* — http://en.wikibooks.org/wiki/Java_Persistence

- *Getting Started with MongoDB - https://docs.mongodb.org/getting-started/shell/*