Brief Introduction to Database Systems

Technical Report · March 2023

DOI: 10.13140/RG.2.2.35951.30884

CITATIONS
5

READS
3,391

1 author:

Manuel José Fernández Iglesias
University of Vigo
265 PUBLICATIONS **1,768** CITATIONS

SEE PROFILE

Brief Introduction to Database Systems

Version 1.1/EN - April 2024

Manuel José Fernández Iglesias
atlanTTic - Universidade de Vigo

atlanTTic
Universidade de Vigo

Summary

Databases are structured systems that allow storing, managing and accessing large volumes of data efficiently. Their study is fundamental in the field of computer science and technology, since databases are a key part of most modern computer applications and systems. In software engineering, database management systems are used to store and manage data in a structured way, which facilitates access to and manipulation of information. Currently, two main models are used for data management using databases, relational databases and non-relational or NoSQL databases. The latter in turn include several approaches based on different data models, such as document databases, object-oriented databases, hierarchical databases, graph stores or key-value pair stores. Each approach has its particular strengths and weaknesses and the choice of a particular model will depend on each application. In general, database software provides functionality for data storage and replication, data redundancy prevention, data security and recovery, and performance optimization. Through these notes, we aim to explore the basic concepts behind databases, including the different types of database management systems, the levels of abstraction in data management, as well as the fundamentals of relational and non-relational databases. By understanding these concepts, we will acquire a solid foundation for working with databases in software development, systems administration and other areas related to computer science and technology.

DOI: 10.13140/RG.2.2.35951.30884

Contents

1 Introduction

Databases are organized collections of information that are stored and managed in a computer system. They are designed to efficiently manage large volumes of structured or unstructured data, and provide a way to store and retrieve data for processing.

Databases are usually the central back-end element of a software system. When we talk about the back-end, we refer to the part of the system responsible for managing data and processing requests from the front-end or user interface. It therefore includes the server logic, the database and the application logic, elements that together provide the intended functionality of the system.

The back-end is often referred to as the *server side* of the system, since it is responsible for executing code and performing operations that are not visible to the user. As we have just indicated, from a software engineering perspective it is organized into three layers, namely the server layer, the database layer and the application layer, each responsible for a specific set of tasks. The back-end is key to providing the functionality and ensuring the performance of a software system. It must be designed and implemented to be scalable, secure and reliable, to ensure that the system can handle the expected number of users and requests, providing the required functionality or services while maintaining data consistency and integrity. To achieve these properties, the selection of the database management system is of fundamental importance.

A database can be thought of as a large electronic filing cabinet that stores information in an organized manner. Each database usually consists of one or more tables, collections of documents, graphs or objects, depending on the type of database. Each of these elements represents a specific entity or concept, such as customers, orders or products. In turn, these elements are composed of records that represent specific entity instances. For example, relational database tables are composed of rows and columns, where each row represents a single record or instance, and each column represents a specific entity's attribute or property. Documents in document-oriented databases correspond to records in the relational model. These are composed, for example, of fields to define specific properties. In object-oriented databases, each object models a record, and the attributes of that object model the properties of instances.

Databases are used in a wide range of applications in business, finance, healthcare, education, entertainment, science, engineering and many others. They are essential for managing large volumes of data, enabling efficient data processing, providing security, ensuring data integrity and consistency, and facilitating data analysis and reporting.

In these introductory notes to databases we will explore the essential concepts that enable data management in modern computer systems. We will begin with an overview of database management systems (DBMSs) and the different levels of abstraction involved in data management. Next, we will focus on data definition and manipulation, exploring the key concepts behind these processes. Then, we will delve into the various types of database management systems, describing both relational and non-relational or NoSQL databases. Finally, we will recap the key concepts covered in this manual and establish a solid foundation for further study of databases.

2 Database Management Systems

A database management system (DBMS) is a software application used to interact with a database efficiently and securely. The DBMS provides an interface for creating, modifying, storing, and retrieving data from the database, and also manages aspects such as data integrity, security, concurrency, and failover. In this document we will describe the most common types of DBMSs, and refer to them interchangeably as database management systems or DBMSs, database systems, or simply as databases.

DBMSs are responsible for handling the internal details of how data is stored on the hard drive or cloud, how data is efficiently accessed, and how data integrity and consistency is ensured. This allows developers and users to focus on the logic of their applications rather than worrying about the technical aspects of data management. Thus, a DBMS is responsible for performing a series of basic tasks to efficiently manage the data in the database, among which are its storage; the creation and modification of schemas and data models; the insertion, update, recovery and deletion of data; the maintenance of information integrity; the control of access and transactions, or the backup and recovery procedures in case of failures.

2.1 Data Management Abstraction Levels

One of the main objectives of a DBMS is to provide a view of data at different levels of abstraction. In the context of DBMS, several levels of abstraction can be identified that provide different views of the data stored in the database. These abstraction levels are known as physical level, logical level and view level.

- At the physical level, it is described how the data is physically stored on the chosen storage devices. Details such as file structure, data access methods and the physical organization of records on disk are defined at this level.
- The logical level describes the logical structure of the database in terms of records, graphs, documents, tables, objects, relationships, attributes or constraints, depending on the DBMS type. This level thus provides a conceptual view of the data, independent of the underlying physical implementation.
- The view level represents specific higher-level views of the data for specific use cases or applications. Views allow certain data to be hidden or presented differently according to user needs, without affecting the underlying database structure (i.e., logical level).

An example of the aspects described at the DBMS's physical level would be for example the organization of data on a solid state disk (SSD). At the physical level, data is stored on disk using specific storage structures that determine how it is accessed and managed. For example, suppose we have a database containing a table of employees. At the physical level, the data in this table could be stored in a file on disk. Within this file, the employee records could be organized into data blocks, which are the basic read/write units on disk. Each data block may contain multiple employee records, and these records would be stored contiguously on disk. In addition, storage techniques such as partitioning and indexing may be used to improve the performance and efficiency of data access. This level of abstraction ultimately focuses on the technical details of how data is stored and managed on disk.

An example of the logical level in a DBMS would be the structure of tables and relationships within a relational database (cf. Sect. 3). At the logical level, in this type of DBMS,

data is conceptually represented in the form of tables, where each table contains rows and columns representing entities and attributes respectively. For example, let's imagine a company's database that stores information about employees. At a logical level, we could have a table called "Employees" with columns called "Identifier", "First Name", "Last Name", "Department", "Salary", etc. Each row in this table represents an individual employee, while each column represents a specific attribute of that particular employee, such as their ID, first name, last name, etc. In addition, at the logical level, relationships between tables are defined to represent how entities are related to each other. For example, we could have an additional table called "Departments" that stores information about the company's departments, with a "Department ID" column that relates to the "Department" column in the "Employees" table. We can say that this level provides a conceptual view of the data, independent of the underlying physical implementation.

An example of the view level in a DBMS would be the creation of specific views for different users or applications. Views allow users to access a specific part of the data in the database, according to their needs, without having to access the underlying database directly. For example, in an online store database that contains information about products, customers and orders, the store manager might have a view that shows all available products, while a salesperson might have a view that shows only the best-selling products, or a warehouse clerk might have a view that shows products that need to be replenished. Each view provides a different perspective of the data in the database, filtering and presenting only the information relevant to the specific use case or application. Views can include additional calculated columns, documents or records, or filters and security restrictions to suit the specific needs of particular users. This level ultimately provides an additional layer of abstraction and security for end users.

2.2 Data Definition and Manipulation

Data definition and manipulation in a DBMS is performed using two main types of languages: data definition language (DDL) and data manipulation language (DML). With DDL, users can define the database structure, that is, create, modify or delete database objects, such as tables, indexes, views, stored procedures, etc. while with DML, data manipulation operations are performed, such as inserting, updating, deleting and retrieving data in a database. Users can interact with the database using the commands or expressions of these languages through user interfaces (e.g., database management applications or tools) or through application programs that connect to the database using application programmers' interfaces (APIs) and programming languages.

The Data Definition Language (DDL) consists of a set of commands used in DBMSs to define and manage the structure of a database. DDL is used to create, modify and delete database objects such as tables, indexes, views, stored procedures and constraints. It is therefore essential for defining the database structure according to system requirements and ensuring its integrity and consistency. DDL commands are used by database administrators and application developers to make changes to the database structure in a controlled and secure manner. Some typical DDL commands are:

- Those that support the creation of new database objects, such as tables, views, indexes, etc., such as **`CREATE TABLE`** in the case of SQL or Cassandra (cf. Sect. 4.3), or **`CreateCollection()`** in MongoDB (cf. Sect. 4.1).

- Those that allow modifications to the structure of existing database objects, such as adding, modifying or deleting columns in a table, such as **`ALTER TABLE`** in the case of SQL or Cassandra.
- Those used to delete database objects, such as tables, views, indexes, etc., such as **`DROP TABLE`** in SQL or Cassandra, or **`Drop()`** in MongoDB.
- Those that allow to reset the database content without altering its structure, such as **`TRUNCATE`** in SQL, **`DELETE`** without conditions in Cassandra, or **`deleteMany({})`** in MongoDB.
- Those that allow adding metadata, comments or descriptions to database objects, such as **`COMMENT`** in SQL, comentaries in Cassandra's schema files (CQL), or the inclusion of comment fields in MongoDB documents.

The Data Manipulation Language (DML) is a set of commands used to perform data manipulation operations on a database. DML is used to insert, update, delete and retrieve data from a database, making it essential for interacting with and performing operations on data stored in a database. Application developers and end users use DML to perform operations on data and obtain the information needed for their tasks and processes. Some of the most common commands in DML would be:

- Those used to retrieve data from one or more tables or documents in the database, such as **`SELECT`** to perform queries in SQL or Cassandra, or the **`find()`** command in MongoDB to locate specific documents.
- Those that support the addition of new records to a database table or document, such as **`INSERT`** in SQL and Cassandra or **`insertOne()`** in MongoDB.
- Those used to modify the values of existing records in a table or document, such as **`UPDATE`** in SQL and Cassandra or **`updateOne()`** in MongoDB.
- Those that allow you to delete records from a database table or document, such as **`DELETE`** in SQL and Cassandra or **`deleteOne()`** in MongoDB.

Defining and manipulating data in a DBMS using APIs involves using the programming interfaces provided by the DBMS to interact with the database from external applications or systems. These APIs allow developers to access DBMS functionality and perform data definition and manipulation operations programmatically, i.e., through code. In other words, using the DBMS API, developers can create, modify or delete database objects, such as tables, indexes, views, stored procedures, etc., as well as perform data manipulation operations, such as inserting, updating, deleting and retrieving data in the database.

The DBMS APIs provide a programmatic way to interact with the database, allowing developers to integrate DBMS functionality into custom applications or existing systems. This can be done through the use of programming languages and programming environments that are compatible with the DBMS API, providing flexibility and control over data definition and manipulation operations.

2.3 Types of Database Management Systems

Currently, we can identify two main types of database systems: relational and non-relational or NoSQL:

Relational databases These are the most common database system solution. They are based on the relational model, which organizes data in tables with rows and columns.

The relationships among tables are defined by fields shared by these tables, which are used to combine them. Relational databases generally use SQL (Structured Query Language) to manipulate and query data.

Non-relational or NoSQL databases These are databases characterized by their non-adherence to the relational model. In other words, NoSQL databases are not built primarily on tables and, as a result, do not typically use SQL for data manipulation. They are designed to handle large volumes of unstructured or semi-structured data and are based on non-relational data models such as key-value pairs, documents, graphs or objects. NoSQL databases are often used for big data, real-time web applications and other application fields where high scalability and performance are required.

Relational and non-relational databases differ in several aspects, such as their data model, the query language, their degree of scalability, their flexibility in managing information, the way consistency is handled, etc. The main differences are listed below.

1. Data model:
   - Relational databases rely on a tabular schema in which data is organized in tables with predefined columns and relationships among them defined by foreign keys. Data is stored in the rows of these tables.
   - NoSQL databases use other data structures such as document collections, graphs, objects or simple key-value pairs. This provides greater flexibility in storing and managing information.
2. Languages (DDL, DML):
   - Relational databases use SQL (Structured Query Language) as the standard query language.
   - NoSQL databases may or may not have a query language, but they usually relay on application programmer's interfaces (APIs) or specific languages to access data, which are also strongly influenced by the underlying data model.
3. Scalability:
   - Relational databases often require vertical scaling, that is, adding more resources to a single server to improve performance when the amount of information to be managed increases. A greater volume of information requires more disk, more memory or greater processing capabilities in the server.
   - NoSQL databases are horizontally scalable and can be distributed across multiple servers. Data is usually distributed in clusters, and the increase in information to be managed is compensated by adding more servers in parallel, which take care of the new data.
4. Flexibility:
   - Relational databases have strict data integrity rules and require predefined schemata, making them less flexible to adapt to changing data needs.
   - NoSQL databases are more flexible and can handle unstructured or semi-structured data, making them better suited to handle complex data types and evolving data schemas.
5. Degree of compliance with ACID properties (atomicity, consistency, isolation, durability):
   - Relational databases are typically strict in their adherence to ACID properties, ensuring data consistency and transaction integrity.
   - NoSQL databases sacrifice ACID properties, or at least some of them to some extent, for the sake of performance and scalability. For example, it is common for them to provide eventual consistency rather than consistency at all times, which

means that changes to data may take some time to propagate throughout the system.

In short, relational systems are more suitable for applications with well-defined and structured data, with lower scalability requirements and whose transactions comply with ACID properties:

Atomicity A transaction is atomic if it is treated as an indivisible unit of work. This means that either all operations in the transaction complete successfully, or none of them will. If a transaction fails or is interrupted, all of its changes are reverted to their original state and the database remains unchanged.

Consistency A transaction maintains consistency if it transforms the database from one valid state to another valid state. This means that the database must be in a consistent state both before and after each transaction is executed.

Isolation A transaction is considered to satisfy the isolation property if it is executed in isolation from other transactions. This means that changes made by a transaction should not be visible to other transactions until they are completed.

Durability Once a transaction is completed, its changes must be permanent and survive any subsequent system failure. This means that changes are stored on a non-volatile storage medium, such as a hard disk, to ensure that they can be recovered in the event of system failure.

On the other hand, NoSQL databases are more suitable for handling large volumes of unstructured data, which require high scalability and flexibility, with lower requirements for consistency, isolation or atomicity. The properties associated with this model are known as BASE properties:

Basic Availability A response is guaranteed to be generated for each request to the database, although the response may be a notification that the execution of the request failed.

Soft state The state of the system may change over time, even if no request is made, in order to guarantee eventual consistency.

Eventual consistency The database could be in an inconsistent state at some point in time, but it is guaranteed that sooner or later the database will move to a consistent state.

We can find different approaches to NoSQL databases relaying on different data models:

Document-oriented databases They organize data in collections of documents rather than in tables. A document can store both structured and unstructured data, and can even contain nested structures. In fact, few restrictions are imposed on the data stored, as long as they meet the basic requirement of being expressible as documents. Two different documents in a collection need not have the same structure or the same properties.

Object-oriented databases They are designed to store and manage objects, which are instances of classes in an object-oriented programming language such as Java or C#. They can be used to store complex data structures and relationships among objects.

Column-oriented databases They store data in columns rather than rows, which can make querying large data sets faster and more efficient. They are often used for big data, business intelligence and data analytics.

Hierarchical databases They arrange data in tree-like structures, with parent-child relationships between data elements. This type of database is often used in large enterprise applications.

Graph stores Relations among data elements are represented as graphs. When data elements (i.e., graph nodes) share multiple relations, they are indicated by edges connect-

ing those nodes. Graph databases can be considered especially optimized for data with many relations (e.g., in social networking applications, content recommenders, fraud detection, logistics management, computer network management, etc.).

Key-value stores Closely related to document-oriented databases, they store values associated with a key (i.e., as key-value pairs). Values can consist of complex data structures (e.g., JSON records). As in document stores, it is not necessary to apply a schema to the values, but when a new value is inserted, the corresponding key must be specified. Besides, its key must be known to access a specific value.

Each type of database system has its own strengths and weaknesses, and the choice of a database system for a particular application depends on the specific needs of that application. Throughout the following sections we discuss these two types of database systems in more detail and provide simple examples of the most commonly used database systems in use today.

3 Relational Databases

Relational database management systems have a strong theoretical foundation developed over the second half of the last century. Specifically, they are based on relational algebra, a theory that uses algebraic structures with well-founded semantics to model data and define queries on it. Data is is organized and stored into two-dimensional tables with rows and columns, and data values are strongly typed, that is, they must belong to one of the data types defined in the system (e.g., integers, real numbers, characters, strings, dates, etc.). Besides, they use to support uninterpreted blobs to store large unstructured data elements whose interpretation depends on the application (e.g., images, audio files, archive files). Tables can be joined and transformed into new tables according to the rules of relational algebra. Today they are still the most widely used database systems, with a multitude of proposals on the market. Among the most popular are mySQL/MariaDB , PostgreSQL, and Oracle Database.

A simple example of a relational database could be a database used to store information about employees in a company. The database might have a table called **`Employees`** with columns such as **`EmployeeID`**, **`Name`**, **`FamName`**, **`Email`**, **`Department`**, and **`Salary`**. The **`EmployeeID`** column might be used as the primary key for the table, which would ensure that each employee has a unique identifier.

The database might also have a second table called **`Departments`**, with columns such as **`DepartmentID`** and **`DeptName`** to represent a department's name. The **`DepartmentID`** column might be used as the primary key for this table.

To create a relationship between the two tables, the **`Department`** column in the **`Employees`** table could be a foreign key that references the **`DepartmentID`** column in the **`Departments`** table. This would allow for a one-to-many relationship between departments and employees, where each department could have many employees, but each employee can belong to only one department.

We can use the following SQL statements to create the two tables described:

```
1 -- create Employees table
2 CREATE TABLE Employees (
3   EmployeeID INT PRIMARY KEY,
```

```
  Name VARCHAR(50),
  FamName VARCHAR(50),
  Email VARCHAR(100),
  Department INT,
  Salary DECIMAL(10, 2)
);

-- create Departments table
CREATE TABLE Departments (
  DepartmentID INT PRIMARY KEY,
  DeptName VARCHAR(100)
);
```

Note that in the **Employees** table, the **Department** column is a foreign key that references the **DepartmentID** column in the **Departments** table. Below is the SQL statement to add this foreign key constraint (i.e., the **Department** column of an employee has to be one of the values of **DepartmentID**, specifically the one that matches the ID of the department in which the employee is enrolled):

```
-- add foreign key constraint to Employees table
ALTER TABLE Employees
ADD FOREIGN KEY (Department)
REFERENCES Departments(DepartmentID);
```

We can insert rows into the newly created databases using the **INSERT** statement:

```
-- inserting departments into the Departments table
INSERT INTO Departments VALUES (1, 'Sales');
INSERT INTO Departments VALUES (2, 'Marketing');
INSERT INTO Departments VALUES (3, 'HR');

-- inserting employees into the Employees table
INSERT INTO Employees VALUES
      (1, 'John', 'Doe', 'johndoe@example.com', 1, 50000.00);
INSERT INTO Employees VALUES
      (2, 'Jane', 'Doe', 'janedoe@example.com', 2, 60000.00);
INSERT INTO Employees VALUES
      (3, 'Bob', 'Smith', 'bobsmith@example.com', 1, 45000.00);
INSERT INTO Employees VALUES
      (4, 'Alice', 'Johnson', 'alicejohnson@example.com', 3, 55000.00);
```

With this, we will have a database with the following **Departments** and **Employees** tables:

DepartmentID	DeptName
1	Sales
2	Marketing
3	HR

IDE	Name	FamName	Email	Department	Salary
1	John	Doe	johndoe@example.com	1	50000.00
2	Jane	Doe	janedoe@example.com	2	60000.00
3	Bob	Smith	bobsmith@example.com	1	45000.00
4	Alice	Johnson	alicejohnson@example.com	3	55000.00

With this structure in place, data could be easily queried and manipulated using SQL. For example, a query could be used to retrieve all employees in a particular department, or to calculate the average salary for each department. To retrieve all employees in a particular department (in this case, the department with ID=3):

```
SELECT *
FROM Employees
WHERE Department = 3;
```

This SQL statement uses the **SELECT** statement to retrieve all columns from the **Employees** table where the **Department** column equals 3.

IDE	Name	FamName	Email	Department	Salary
4	Alice	Johnson	alicejohnson@example.com	3	55000.00

You could modify the **WHERE** clause to filter by a different department ID. We can also use the (foreign key) relationship between **Department** and **DepartmentID** to obtain the names and surnames of the employees belonging to a certain department, specified by name:

```
SELECT Employees.Name, Employees.FamName
FROM Employees
JOIN Departments
ON Employees.Department = Departments.DepartmentID
WHERE Departments.DeptName = 'Sales';
```

This SQL sentence uses the **JOIN** operator to combine data from tables **Employees** and **Departments** according to the values of columns **Department** from table **Employees** and column **DepartmentID** from **Departments**. From the resulting merged table, we select the rows with **DeptName** equal to "Sales" and from them columns **Name** and **FamName**. The result of this query is a table with the names and surnames of the employees working in the "Sales" department:

Name	FamName
John	Doe
Bob	Smith

To calculate the average salary for each department we use the SQL sentence below:

```
SELECT Departments.DeptName, AVG(Employees.Salary) AS AvgSalary
FROM Employees
JOIN Departments ON Employees.Department = Departments.DepartmentID
GROUP BY Departments.DeptName;
```

Again, this SQL statement uses the **JOIN** clause to combine data from the **Employees** and **Departments** tables, according to the values on the **Department** column in the **Employees** table and the **DepartmentID** column in the **Departments** table. It then uses the **AVG** function to calculate the average salary for each department, and the **GROUP BY** clause to group the results by department name. As a result, the following table is obtained:

DeptName	PromedioSalary
HR	55000.0
Marketing	60000.0
Sales	47500.0

4 Not Only SQL (NoSQL) Databases

As mentioned above, there are several approaches to data definition and management in NoSQL systems: they may or may not have a query language, they provide different APIs to access the information managed, they are based on different models for data organization, etc. The next paragraphs discuss some examples of the different NoSQL approaches listed in Sect. 2.3.

4.1 Document-Oriented Databases

In this introduction, we utilize MongoDB to discuss the basic elements of these databases. MongoDB is a popular open-source document-oriented database. It is designed to store and manage unstructured, semi-structured, and structured data in a flexible and scalable way. MongoDB stores data in documents, which are similar to JavaScript Object Notation (JSON) objects, and allows for the storage of complex data structures with nested arrays and sub-documents. JSON is an open standardized file format for data interchange that uses human-readable text to store and transmit data objects consisting of attribute-value pairs, arrays or other serializable values.

MongoDB is known for its scalability, performance, and high availability features. It can be used for a variety of use cases, including web and mobile applications, content management systems, real-time analytics, and Internet of Things (IoT) applications. MongoDB supports various programming languages and provides a flexible query language that enables complex querying and data aggregation.

Some of the key features of MongoDB include:

- Flexible data model.
- High availability and automatic failover.
- Horizontal scaling with sharding.
- Rich query language with support for complex queries and aggregations.
- Geospatial queries.
- Secondary indexes and full-text search.

Overall, MongoDB is a powerful and popular NoSQL document database that provides developers with a flexible and scalable platform for storing and manipulating data.

To create a collection called **`Employees`** similar to that in the previous section, including sample data, we may use the sample code snippets below:

```
db.createCollection("Employees");

db.Employees.insertMany([
  {
    "EmployeeID": 1,
    "FirstName": "John",
    "LastName": "Doe",
    "Email": "johndoe@example.com",
    "Department": "Sales",
    "Salary": 50000.00
  },
```

```
  {
    "EmployeeID": 2,
    "FirstName": "Jane",
    "LastName": "Doe",
    "Email": "janedoe@example.com",
    "Department": "Marketing",
    "Salary": 60000.00
  },
  {
    "EmployeeID": 3,
    "FirstName": "Bob",
    "LastName": "Smith",
    "Email": "bobsmith@example.com",
    "Department": "Sales",
    "Salary": 45000.00
  },
  {
    "EmployeeID": 4,
    "FirstName": "Alice",
    "LastName": "Johnson",
    "Email": "alicejohnson@example.com",
    "Department": "HR",
    "Salary": 55000.00
  }
]);
```

This MongoDB code creates a **Employees** collection with four sample documents (the four employees from the relational database example at Sect. 3). Note that MongoDB uses a flexible schema, so each document can have different fields and data types.

To create a collection called **Departments** with the departments in Sect. 3 we may use the code snippet below:

```
db.createCollection("Departments");

db.Departments.insertMany([
  {
    "DepartmentID": 1,
    "DeptName": "Sales"
  },
  {
    "DepartmentID": 2,
    "DeptName": "Marketing"
  },
  {
    "DepartmentID": 3,
    "DeptName": "HR"
  }
]);
```

This MongoDB code creates a **Departments** collection with three sample documents. Note that unlike a relational database, MongoDB does not enforce strict relationships between collections, so we do not need to create a foreign key constraint. Instead, we can use queries to link data across collections if needed.

To retrieve all employees in the “Sales” departmentwe can use the sentence below:

```
db.Employees.find({"Department": "Sales"})
```

This MongoDB code uses the **`find()`** method to retrieve all documents in the **`Employees`** collection where the **`Department`** field equals "Sales". The value passed to the **`find()`** method can be modified to retrieve all employees in a different department, or other query operators can be used to filter the results according to several criteria. Additionally, the fields to include or exclude from the results can be specified using the **`projection`** parameter. In a similar way, **`sort`** can be used to sort the results, among other options.

We can use the sentence below to retrieve the emails of all employees in the "Sales" department:

```
db.Employees.find({"Department": "Sales"}, {"Email": 1})
```

As pointed out above, the second argument passed to the **`find()`** method is an optional **`projection`** parameter, which specifies which fields to include in the results. In this case, only the **`Email`** field is included, which has a value of 1, and the rest of the fields are excluded. This second parameter can be modified to include or exclude other fields. Additional query operators can be utilized to filter the results based on different criteria.

If we wanted to retrieve the emails from all employees in the Sales department, and sort them according to the employee's name, we may use the code snippet below:

```
db.employees.find(
   { DeptName: "Sales" },
   { \_id: 0, Email: 1, Name: 1 }
).sort({ Name: 1 })
```

The **`DeptName`** field is used to specify the target department, and the **`projection`** parameter is used to select only the **`Email`** and **`Name`** fields from the documents. Field **`_id`** is included by default in all returned documents. In case it is not needed, this can be indicated with the **`projection`** parameter by assigning to it value 0 or `false`.

The **`sort`** method is used to sort the results by the **`Name`** field in ascending order. By default, MongoDB sorts results in ascending order, so we do not need to specify the sort order explicitly as 1.

This code would return a cursor object containing the emails and names of all employees in the "Sales" department, sorted by name. We can iterate over the cursor to access the results or manipulate them as needed.

```
const cursor = db.Employees.find({"Department": "Sales"}, {"Email": 1});

while (cursor.hasNext()) {
  const employee = cursor.next();
  console.log(employee.Email);
}
```

This JavaScript code assumes that you are using the MongoDB Node.js driver to execute the query and iterate over the cursor. The **`find()`** method returns a cursor to the results, which can be iterated over using the **`hasNext()`** and **`next()`** methods. In each loop iteration,

the **`next()`** method returns the next document in the cursor as a JavaScript object, which can be accessed using dot notation to retrieve the **`Email`** field. This example simply logs the email to the console, but data can be processed in any other way, such as being stored in an array or passed to a function.

Other popular document-oriented database systems include Amazon DynamoDB, the fully managed NoSQL database service from Amazon Web Services (AWS), or Apache CouchDB, a document-oriented database system that provides a RESTful HTTP API for data access.

4.2 Object-Oriented Databases

A popular object-oriented database is db4o, an open-source object-oriented database management system that provides native object persistence and retrieval for Java and .NET applications. It stores objects directly into the database, rather than using a relational model, and provides support for object-oriented concepts such as inheritance, polymorphism, and encapsulation.

The following example illustrates how a simple Java class can be created and persist its objects in db4o:

```
import com.db4o.*;
import com.db4o.query.*;

public class Person {
      // Class attributes
  private String name;
  private int age;
  private String address;

      // Class constructor
  public Person(String name, int age, String address) {
    this.name = name;
    this.age = age;
    this.address = address;
  }

      // Class accessors (getters)
  public String getName() { return name; }
  public int getAge() { return age; }
  public String getAddress() { return address; }

      // Class mutators (setters)
  public void setName(String name) { this.name = name; }
  public void setAge(int age) { this.age = age; }
  public void setAddress(String address) { this.address = address; }

      // Main program
  public static void main(String[] args) {
    // Open the db4o database file
    ObjectContainer db =
      Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), "persons.db");

    // Create a new person object and store it in the database
    Person person1 = new Person("John Doe", 35, "123 Main St.");
```

```
    db.store(person1);

    // Query the database for all persons and print their names
    Query query = db.query();
    query.constrain(Person.class);
    ObjectSet<Person> persons = query.execute();
    while (persons.hasNext()) {
      Person person = persons.next();
      System.out.println(person.getName());
    }

    // Close the database
    db.close();
  }
}
```

This Java code creates a simple **`Person`** class with three attributes (**`name`**, **`age`**, and **`address`**) and methods to get and set these fields (i.e., getters and setters). It then creates a db4o database file, stores a **`Person`** object in the database, queries the database for all persons and prints their names, and finally closes the database.

4.3 Columnar Databases

In a columnar database, data is stored vertically in columns, as opposed to horizontally in rows as in a relational row-oriented database. This allows for more efficient querying and processing of large datasets, especially for analytical workloads.

An example of a columnar database is Apache Cassandra. Cassandra is a highly scalable and distributed NoSQL database that is optimized for write-heavy workloads. It is used by many large-scale organizations, such as Facebook, Netflix, and eBay.

In Cassandra, data is organized into keyspaces, which are similar to databases in a traditional relational database. Each keyspace contains one or more tables, which are defined by a set of columns and their associated data types. The data in a table is stored in columns, which are grouped together into column families.

Here is an example of how a table might be defined in Cassandra using its query language (i.e., CQL, Cassandra Query Language):

```
CREATE TABLE user (
   user\_id uuid PRIMARY KEY,
   first\_name text,
   last\_name text,
   email text,
   age int,
   city text
) WITH compression =
      {'sstable_compression': 'LZ4Compressor', 'chunk_length_kb': '64'};
```

In this example, the user table has six columns: **`user_id`**, **`first_name`**, **`last_name`**, **`email`**, **`age`**, and **`city`**. The **`user_id`** column is the primary key for the table, which determines how the data is partitioned and distributed across a cluster.

The CQL sentence below represents a query to obtain the e-mail addresses of all users in a given village:

```
SELECT email FROM user WHERE poblacion = 'Baiona';
```

The **SELECT** statement reads one or more columns from one or more rows of a table. It returns a result set with the rows matching the request, where each row contains the values of the selection corresponding to the query. In addition, functions including aggregations can be applied to the result. For example,

```
SELECT count (*) FROM user WHERE poblacion = 'Baiona';
```

Returns the total number of users in "Baiona".

Columnar databases like Cassandra are well-suited for analytical workloads that require querying large datasets. They can handle high write and read loads and scale horizontally by adding more nodes to a cluster. However, they may not be as efficient for transactional workloads that require frequent updates or inserts of individual records. We can say that column-oriented databases share aspects of relational databases and document-oriented databases.

Another columnar data management system would be Apache HBase, a distributed database system that provides real-time read/write access to large data sets.

4.4 Hierarchical Databases

An example of a hierarchical database is IBM's Information Management System (IMS). IMS is a hierarchical database management system designed for high-performance transaction processing applications. It is used primarily in large-scale mainframe environments, such as banking, finance, and insurance.

In a hierarchical database like IMS, data is organized in a tree-like structure, with each record having one parent and potentially multiple children. The parent-child relationships create a natural hierarchy of data. For example, in a banking application, a customer record may be the parent of multiple account records, and each account record may be the parent of multiple transaction records.

Here is an example of how a hierarchical database might be structured in IMS:

```
CUSTOMER
    |
    +-- ACCOUNT
           |
           +-- TRANSACTION
           |
           +-- TRANSACTION
           |
           +-- TRANSACTION
    |
    +-- ACCOUNT
           |
```

```
          +-- TRANSACTION
          |
          +-- TRANSACTION
```

In this example, the **CUSTOMER** record is the top-level parent, with each **ACCOUNT** record being a child of the **CUSTOMER** record. Each **TRANSACTION** record is a child of an **ACCOUNT** record.

Hierarchical databases like IMS are efficient for handling large amounts of data with high-performance requirements. However, they can be more difficult to manage than other database models and can be less flexible in terms of querying and reporting capabilities.

4.5 Graph Stores

Graph databases can be considered special NoSQL databases optimized for data with many relationships. If there are no relationships among entities, it makes little sense to use this type of database.

Here is a simple example using Neo4j, perhaps the most popular platform today for graph management. Suppose that we are developing a social network and we want to model the relationships between users as a graph. Using Neo4j, we can represent users as nodes and relationships between them as edges. To create a new user node in Neo4j, we use a query in Cypher language:

```
CREATE (:User {name: "John", age: 30, sex: "male"})
```

Cypher is a declarative query language used in Neo4j, designed to facilitate querying and manipulating data in graphs with a syntax similar to that of SQL. It supports the creation of nodes and relations (i.e., edges), as well as filtering and searching for specific nodes and relations. The Cypher query above creates a new user node with the properties **name**, **age** and **sex**.

It is also possible to create relationships between nodes using Cypher queries. For example, to create a relationship **FRIEND_OF** between two users, we can use the following query:

```
MATCH (u1:User {name: "John"}), (u2:User {name: "Mary"})
CREATE (u1)-[:FRIEND_OF]->(u2)
```

This query matches user nodes with names “John” and “Mary” and creates a **FRIEND_OF** relationship between them.

Once the data is stored in a Neo4j graph, we can perform a broad range of queries and analysis on the data. For example, we can use Cypher to find all users who are friends with a particular user:

```
MATCH (u1:User {name: "Jack"})-[:FRIEND_OF]->(u2:User)
RETURN u2.name
```

This query searches for the user named “Jack” and returns the names of all users who are friends with him. More generically, search queries for related nodes have the format:

```
MATCH (n:Label)-[r]->(m)
RETURN n, r, m
```

Where “Label” is the label of the nodes in which we are interested , and “r” is the relationship between nodes “n” and “m”. Keyword **MATCH** specifies the pattern to compare in the graph, and keyword **RETURN** specifies the actual data to be fetched.

Other graph store systems include FlockDB or InfiniteGraph. FlockDB is a distributed graph database system developed by Twitter and designed to store and query large-scale graphs with billions of nodes and edges. FlockDB uses a simple graph data model that represents nodes as unique 64-bit integers and edges as one-way relations between nodes. InfiniteGraph is designed to store and query large-scale graphs and, unlike other graph databases, it uses a hypergraph model that allows nodes to be connected to any number of other nodes via edges that can have any number of endpoints. This provides a very flexible data model that can represent complex relationships between entities.

4.6 Key-Value Pair Stores

Redis is a popular in-memory data store that supports various data structures, including key-value pairs. The following is a simple example of key-value pair management using Redis.

Let’s assume that we have a Web application where storage and retrieval of user sessions is required. We can store session data in the form of key-value pairs using the **SET** command, which defines a value for a given key:

```
SET session:<session_id> <session_data>
```

session_id is a unique identifier for the session (i.e., the key), and **session data** is the data associated with that session (i.e.,the value associated with the key). For example, to store a session with identifier “12345” and data

```
{"user_id": "123", "username": "John"}
```

we can use command:

```
SET session:12345 '{"user_id": "123", "username": "John"}'
```

To retrieve a user session with Redis, we can use command **GET**:

```
GET session:<session_id>
```

For example, to retrieve the session with ID “12345”:

```
GET session:12345
```

This command will return session data (i.e., the value associated with key “12345”) as a string in JSON format.

In addition to basic operations such as **`SET`** and **`GET`**, Redis provides many other commands to manage key-value pairs, including operations to increment or decrement numeric values, append values to existing values, etc. In summary, Redis provides a simple and effective way to manage key-value pairs, well suited for applications that require high performance and scalability.

Other schemes for managing key-value pairs would be for example Couchbase, a data management system that provides distributed key-value storage with query capabilities, or MemcacheDB, based on a data model in which data is stored as key-value pairs and distributed among several nodes in a cluster. It provides a simple API for querying and updating data using popular programming languages such as Java, Python and Ruby. Memcachedb is a persistence-enabled variant of Memcached, an open source, high-performance, distributed memory object caching system.

5 Conclusion

Database systems are a crucial component of today’s computer systems, as they are used to store, manage and retrieve large amounts of data. A database is a collection of data organized and stored in a predefined format, supporting data access and manipulation. There are several types of database systems. In this introduction, we discussed briefly relational databases, document-oriented databases, object-oriented databases, columnar databases, graph stores, key-value pair stores and hierarchical databases. Each type of database has its strengths and weaknesses, depending on the specific requirements of each application.

Relational databases are the most common type of database today. Based on the relational model, with a well-established and robust theoretical foundation, data is stored in tables whose rows represent data records. They are used for many different types of applications, from small-scale solutions to large enterprise systems. Document-oriented databases, on the other hand, are designed to handle unstructured data and can be more flexible and scalable than relational databases. Object-oriented databases are used to store objects and exploit the properties of the object paradigm (i.e, inheritance, polymorphism and encapsulation). Column-oriented databases combine properties of relational and NoSQL databases, graph-oriented databases exploit relationships between entities, key-value stores with their simple data model are well suited to applications requiring high availability and real-time data access, and finally hierarchical databases organize data in a tree-like structure.

Modern databases are usually implemented using specialized software, such as MySQL, Oracle, MongoDB, db4o, Neo4j, Redis, IMS or Cassandra, among others. In general, any database management system provides a variety of functions in addition to information storage and retrieval, such as data replication, backup and failover, and performance optimization. A well-designed database helps to improve application performance, reduce data redundancy and increase security.

The following are some basic references for further study in the field of databases.

- “Database System Concepts” by Abraham Silberschatz, Henry F. Korth and S. Sudarshan (Silberschatz, Korth, & Sudarshan, 2010) is a widely used textbook on database

systems, covering topics such as relational database design, query processing and optimization, concurrency control, and transaction management.

- "An Introduction to Database Systems" by C. J. Date (Date, 2004) provides a comprehensive introduction to the theory and practice of database systems, including relational database design, normalization, and SQL.
- "Database Management Systems" by Raghu Ramakrishnan and Johannes Gehrke (Ramakrishnan & Gehrke, 2003) covers a wide range of topics related to database systems, including relational database design, query processing and optimization, transaction management, and distributed databases.
- "Modern Database Management" by Jeffrey A. Hoffer, Ramesh Venkataraman and Heikki Topi (Hoffer, Venkataraman, & Topi, 2019) covers the basics of database systems, including database design, normalization, and SQL, as well as more advanced topics such as distributed databases, data warehousing, and business intelligence.
- "Database Systems: A Practical Approach to Design, Implementation, and Management" by Thomas Connolly and Carolyn Begg (Connolly & Begg, 2014) offers a practical, hands-on approach to database systems, covering topics such as database design, SQL, transaction management, and database security.
- "Fundamentals of Database Systems" by Ramez Elmasri and Shamkant B. Navathe (Elmasri & Navathe, 2016) is a textbook that also covers the basics of database systems, including database design, normalization, SQL, transaction management, and concurrency control.
- "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence" by Martin Fowler and Pramod Sadalage (Fowler & Sadalage, 2012) is an excellent introduction to the world of NoSQL databases. The book provides a clear and concise overview of the different types of NoSQL databases, including document-oriented, key-value, column-family, and graph databases, and explains when and why they might be a good fit for certain applications. The authors also cover important topics such as consistency, availability, and partition tolerance, and explain how they are addressed in NoSQL databases.
- "Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement" by Eric Redmond and Jim R. Wilson (Redmond & Wilson, 2012) is a comprehensive guide to the rapidly evolving world of NoSQL databases. The book takes a hands-on approach, providing readers with an opportunity to work with seven different NoSQL databases over the course of seven weeks. The authors provide a brief introduction to each database, along with practical examples of how to use it.

These books are just a few examples of the many resources available on database systems. Depending on the specific interests and needs, there may be other books, articles, and online resources that may be more suitable.

Good tutorials and courses to get started can also be found on the Internet. For example:

- W3Schools SQL Tutorial. W3Schools is a popular website that offers tutorials on various web development technologies, including SQL. Their SQL tutorial covers the basics of SQL, including creating tables, inserting data, querying data, and modifying data.
- SQLZoo is a free online tutorial that offers interactive SQL exercises. The tutorial covers SQL basics, such as SELECT statements and joins, as well as more advanced topics such as subqueries and aggregate functions.
- The Stanford Database Course is a free online database course that covers the basics of database systems, including relational database design, SQL, and transaction management. The course includes video lectures, quizzes, and programming assignments.

- MongoDB University offers free online courses on various topics related to MongoDB, including database administration, data modeling, and application development with MongoDB. These courses include video lectures, quizzes, and hands-on exercises.
- Cassandra Tutorial. This W3Schools tutorial on Cassandra presents the basics of this column-oriented database and the CQL query language. It also provides a comparison with HBase (another popular column-oriented database) and relational database systems.
- Getting started with Redis is a tutorial available on the official site of this in-memory structure store. It covers the installation of Redis on various systems, the use of its command interpreter and the integration of Redis with the most popular programming languages.
- Neo4j Tutorials is the official Neo4j tutorial page. Provides tutorials to illustrate data import capabilities and to set up and use a database in various scenarios.
- The db4o Tutorial is a free online course on this object-oriented database system. It covers the basics of dbo4, database design and various programming tasks with practical exercises using all the features offered by the db4o API.
- Udemy Database Courses. Udemy is an online learning platform that offers a wide range of learning resources, both free and paid. The content available on Udemy about database technologies is huge.
- Coursera Database Courses. Coursera is an online learning platform that offers courses from top universities and organizations. There are several hundreds of database courses available on Coursera, both relational and NoSQL, including courses on query languages, database design, data warehousing, data science, ...

References

Connolly, Thomas and Carolyn Begg (2014). *Database Systems: A Practical Approach to Design, Implementation, and Management*. Pearson.

Date, C. J. (2004). *An Introduction to Database Systems*. Addison-Wesley.

Elmasri, Ramez and Shamkant B. Navathe (2016). *Fundamentals of Database Systems*. Pearson.

Fowler, Martin and Pramod Sadalage (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional.

Hoffer, Jeffrey A., Ramesh Venkataraman, and Heikki Topi (2019). *Modern Database Management*. Pearson.

Ramakrishnan, Raghu and Johannes Gehrke (2003). *Database Management Systems*. McGraw-Hill.

Redmond, Eric and Jim R. Wilson (2012). *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. Pragmatic Bookshelf.

Silberschatz, Abraham, Henry F Korth, and S Sudarshan (2010). *Database System Concepts*. McGraw-Hill.