

1-Database Fundamentals

Author: [Vincent Lau](#)

Note: This material is intended for educational purposes only. All rights reserved. Any unauthorized sharing or copying of this material, in any form, to any individual or party, for any use without prior permission, is strictly prohibited.



Learning Objectives

- Understand the purpose and core concepts of DBMS.
- Understand the DBMS structure.

- Understand the difference between relational and non-relational databases.
- List the various types of serialization formats.

Getting Started

What is a database?

- Data
 - Some values referring to real world facts
 - May be in various formats, e.g. text, image, audio file, video file, etc.
- Database
 - A large collection of inter-related data
- Database Management System (DBMS)
 - **DBMS = database(s) + a set of programs that store and access the data**

Drawbacks of Storing DataBase in File Systems

1. Difficulty in accessing data

- It may be **inefficient** to locate a piece of information.

2. Data **redundancy** and **inconsistency**

- Data got updated in one place but not some others.

3. Data isolation

- Because data is scattered in various files, and **files may be in different formats**, writing new programs to retrieve the appropriate data is difficult.

4. Atomicity problems (transaction control/ rollback)

- How can we **ensure that a transfer of money is done completely**?
- What would happen if the system crashes right after money is deducted but not deposited?

5. Concurrent access anomalies (table/ record lock)

- Inconsistency can occur, e.g. **two customers reading and updating a balance** at the same time.

6. Integrity problems (Code to health check if data is consistency)

- Programmers need to **enforce consistency constraints** by adding appropriate code to the various application programs, which is hard to manage.

Database Management Systems

- **Big DBMS vendors: Oracle, IBM DB2, Microsoft, Sybase etc.**



- **Open source DBMS: PostgreSQL, MySQL.**



The Three Powerful Concepts Supported by a DBMS

1. Data Abstraction

- **Physical level** - How data are actually stored, structured or compressed.
- **Logical level** - How data are modeled, and how the relationships among data are maintained.
- **View level** - Different users may have different view of the same database depending on their use case.

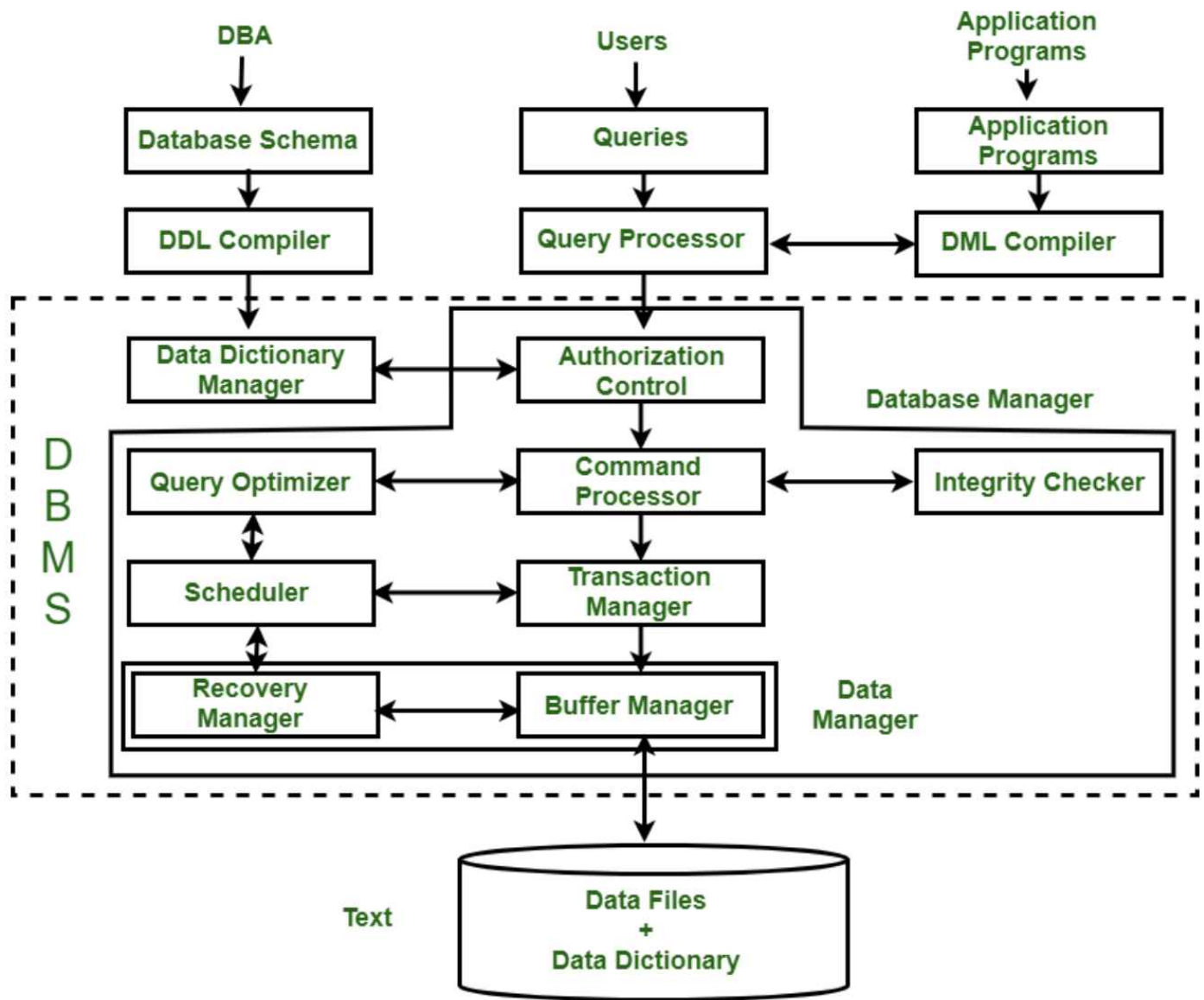
2. Data Modeling

- Data modeling is used to describe data in a systematic way.
- A collective tool for describing **data, data relationships, data semantics, and data constraints.**

3. Database Languages

- **DDL** (Data Definition Language)
- **DML** (Data Manipulation Language)
- **DCL** (Data Control Language)
- **TCL** (Transaction Control Language)

DBMS Structure Overview



Source: [Structure of Database Management System - GeeksforGeeks](#)

Relational Databases

Key Concepts

- Database is divided into different **tables**. A table works like a spreadsheet.
- Each data entry is a **row**.
- **Columns** are the different properties data entries can have.
 - Usually columns impose a **data type** that they can contain.
 - Columns can also specify other restrictions:
 - Whether it's **mandatory** for rows to have a value in that column
 - Whether the value in the column must be **unique** across all rows in the table

- And more...
 - Columns are most commonly referred to as `fields`.
- The **combination of fields and restrictions** is called the table's `schema`.
 - Different tables can use different types of fields.
 - The organization of a database table is given by its fields and the restrictions they enforce.
 - Limitation of the relational model - The database system won't accept a row into a table if it violates the table's schema.

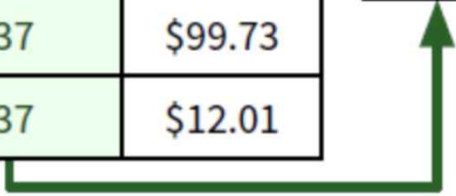
Relationships

- Imagine a database of invoices contained in a single table. For each invoice, we must store information about the order and the customer. When storing more than one invoice for the same customer, information gets duplicated:

| Date | Customer Name | Customer Phone Number | Order Total |
|------------|----------------|-----------------------|-------------|
| 2017-02-17 | Bobby Tables | 997-1009 | \$93.37 |
| 2017-02-18 | Elaine Roberts | 101-9973 | \$77.57 |
| 2017-02-20 | Bobby Tables | 997-1009 | \$99.73 |
| 2017-02-22 | Bobby Tables | 991-1009 | \$12.01 |

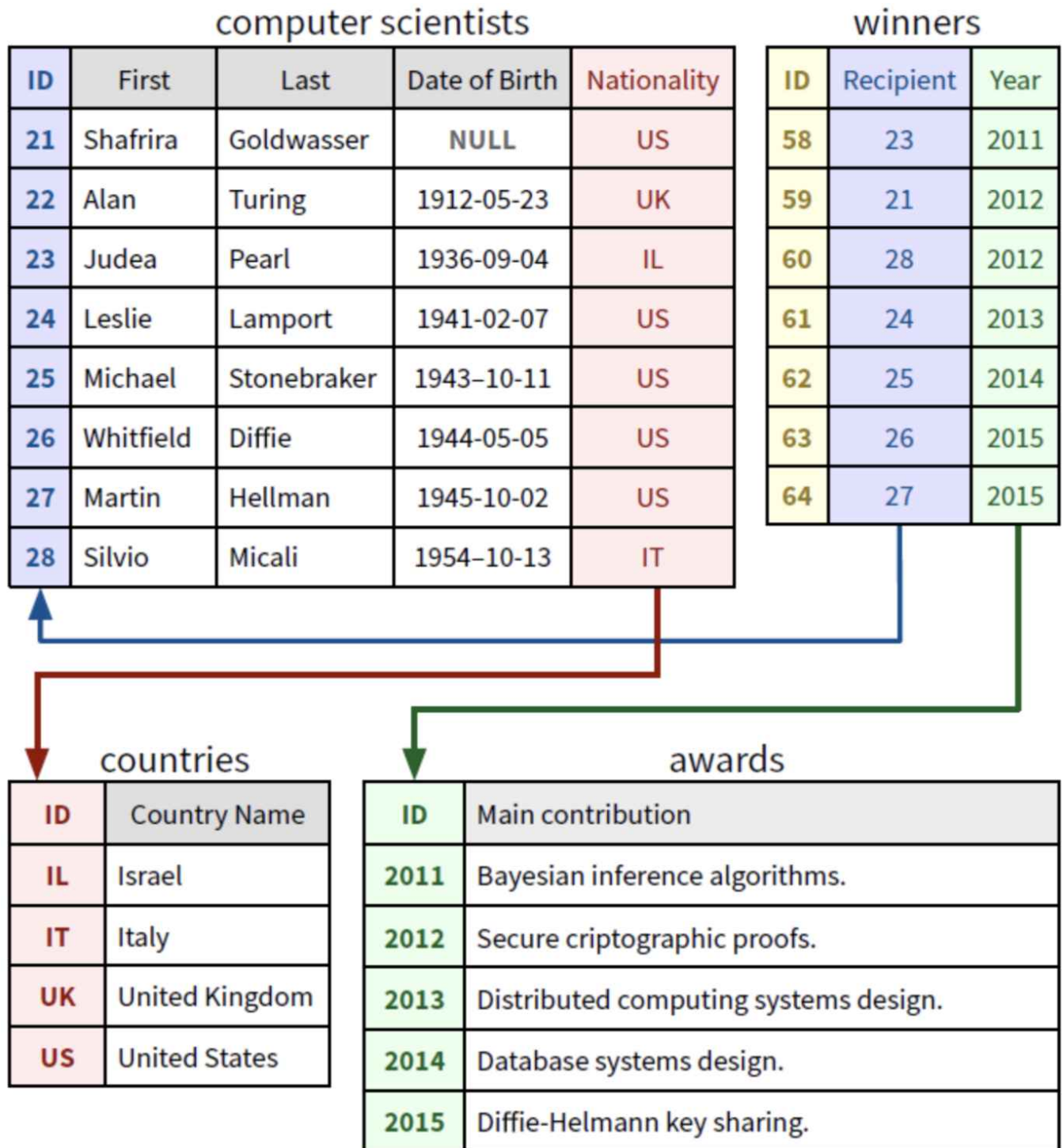
- Duplicated information is hard to manage and update. To avoid it, the relational model splits related information in different tables. By referring to the *customers* table by ID, we remove data duplication in the *orders* table.

| orders | | | | customers | | |
|--------|------------|----------|---------|-----------|----------------|----------|
| ID | Date | Customer | Amount | ID | Name | Phone |
| 1 | 2017-02-17 | 37 | \$93.37 | 37 | Bobby Tables | 997-1009 |
| 2 | 2017-02-18 | 73 | \$77.57 | 73 | Elaine Roberts | 101-9973 |
| 3 | 2017-02-20 | 37 | \$99.73 | | | |
| 4 | 2017-02-22 | 37 | \$12.01 | | | |



- To support relationships, every table has a special identification field or ID. We use ID values to refer to a specific row within a table. These values must be **unique**.

- The ID field of a table is also known as its **primary key**. A field that records references to other rows' IDs is called a **foreign key**.



- When a database is organized in a way that is completely free of duplicate information, we say that the database is **normalized**. The process of transforming a database with replicated data to one without is called **normalization**.

Schema Migration (DDL)

- When we need to change the database structure, we create a **schema migration** script. It automatically **upgrades** the schema and transforms existing data accordingly.

- These scripts can also undo their changes, which allows to easily restore the database structure to match a past working version of the software - `rollback`.
- For example, you modify a column width (from 100 to 120) in an existing table. Afterwards, you want to downsize it to 100. It may not be successful due to data issues (some existing data length > 100 already inserted in table).

The SQL Language

- The query language almost every relational Database Management System (DBMS) works with is called `Structured Query Language (SQL)`.
- Initially developed by IBM in the early 1970s, SQL comes in many flavors today, e.g. *PL/SQL* by Oracle, *T-SQL* by Microsoft, and *SQL PL* by IBM DB2. Some popular relational databases include IBM DB2, MariaDB, *MySQL*, Sybase, Oracle, and *PostgreSQL*.

SQL - Wikipedia

- SQL is more often pronounced *sequel*, but saying *S-Q-L* isn't incorrect.
- This is what a typical SQL SELECT query looks like. To get all fields in the table, you can write "`SELECT *`" (although it is best avoided for performance purpose).

```
1 SELECT <field name> [, <field name>, <field name>, ...]
2 FROM <table name>
3 WHERE <condition>;
```

- To get all fields from the *instructor* table, filtering by *dept_name* and *salary* fields, you can write:

```
1 SELECT *
2 FROM instructor
3 WHERE dept_name = 'Physics' AND salary > 80000;
```

- You can make a SQL `SELECT` query without specifying a `WHERE` clause. This causes every customer to be returned.
- There are also other query operators you should know:
 - `ORDER BY` - sorts the results according to the specified field(s)
 - `GROUP BY` - used when you need to group the results in boxes and return per-box aggregated results.

```

1 SELECT dept_name, AVG(salary)
2 FROM instructor
3 GROUP BY dept_name
4 ORDER BY dept_name ASC;

```

- `SELECT DISTINCT` - causes each customer to be returned only once
- To fetch data from multiple tables, you can use the `JOIN` keyword:

```

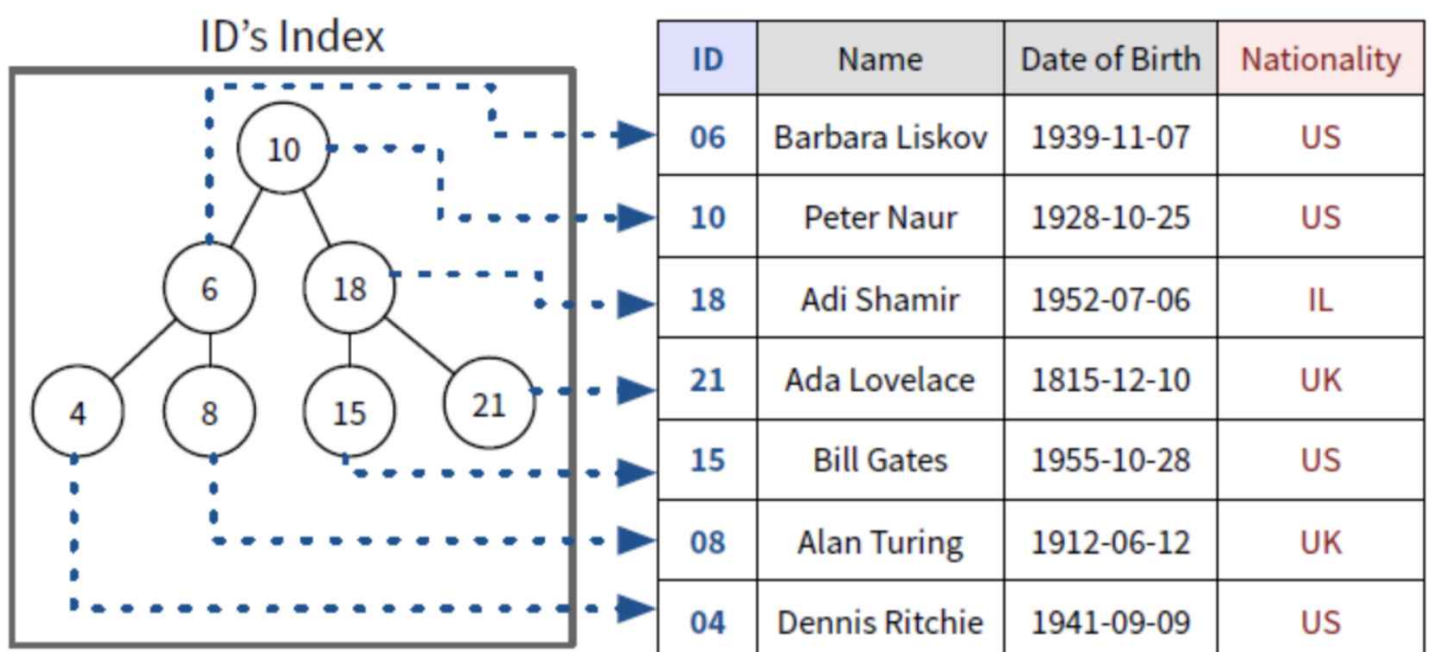
1 SELECT T.course_id, I.name AS instructor_name
2 FROM teaches T
3 JOIN instructor I ON I.ID = T.ID
4 WHERE T.semester = 'Spring' AND T.year = 2010
5 ORDER BY T.course_id ASC;

```

- However, joins are expensive and sometimes infeasible to compute, especially when the tables are large. The `JOIN` is the greatest power and, at the same time, the major weakness of relational databases.

Indexing

- To speed up lookup, the DBMS builds an auxiliary `index`, mapping row IDs to their respective addresses in memory.
- An index is essentially a `self-balancing search tree` (strictly speaking, a **B+ tree**, the tree diagram has been simplified for demonstration purpose).
- Each row in the table corresponds to a node in the tree.



- Node keys are the values in the fields we index. To find the record with a given value, we search for the value in the tree. Once we find a node, we get the address it stores, and use it to fetch the record.
- Usually, an index is created by the DBMS for each primary key in the database. If we often need to find records by searching other fields (for instance, if we search customers by name), we can instruct the DBMS to create additional indexes for these fields too.
- **Uniqueness Constraints** - Indexes are often automatically created for fields that have a uniqueness constraint. With an index, we can quickly search if the value we are trying to insert is already present. Indexing fields that have a uniqueness constraint is necessary to be able to insert items fast.
- **Sorting** - When using `ORDER BY` in a field without an index, the DBMS has to sort the data in memory before serving the query. Indexes help to fetch rows in the index fields' sorted order without any extra calculations.
- **Joint Indexes** - When sorting by two fields is required, `joint indexes` are used. They index multiple fields and won't help finding items faster, but they make returning data sorted by multiple fields a breeze.
- **Performance** - Why don't we have indexes for *all* fields in every table then? The problem is when a new register is inserted to or removed from the table, all its indexes must be updated to reflect that, which can become computationally expensive (remember tree balancing). Moreover, indexes occupy disk space.
- The **Explain Tools** - Use "`explain`" tools to check your queries and add indexes only when it makes a difference in performance. The "explain" tool reports which indexes a query used, and how many rows had to be sequentially scanned to perform a query.

Transactions

- Imagine a bank transfer from person A to person B - two operations must be performed on the bank's database: *subtracting* from one balance, and *adding* to another.
- Problems - What if someone wires fund to person A at the same time person A's balance is being deducted (race condition)? What if someone queries the total balance of all accounts *after* a subtraction is recorded but *before* the corresponding addition is (dirty reads)? What if the system loses power between the two operations (system crashes)?
- We need a way for the database system to perform either *all* changes from a multi-part operation, or keep the data unchanged. To do this, database systems provide a functionality called `transactions`.
- Transaction - a list of database operations that must be executed `atomically` (all or nothing).

```
1 START TRANSACTION;
2 UPDATE vault SET balance = balance + 50 WHERE id=2; -- SUCCESS
3 UPDATE vault SET balance = balance - 50 WHERE id=1; -- SUCCESS
4 COMMIT;
5
6 -- ROLLBACK;
```

Non-Relational Databases

- The `non-relational model` ditches tabular relations.
- Since the non-relational database systems use query languages other than SQL, they are also referred to as `NoSQL databases`.

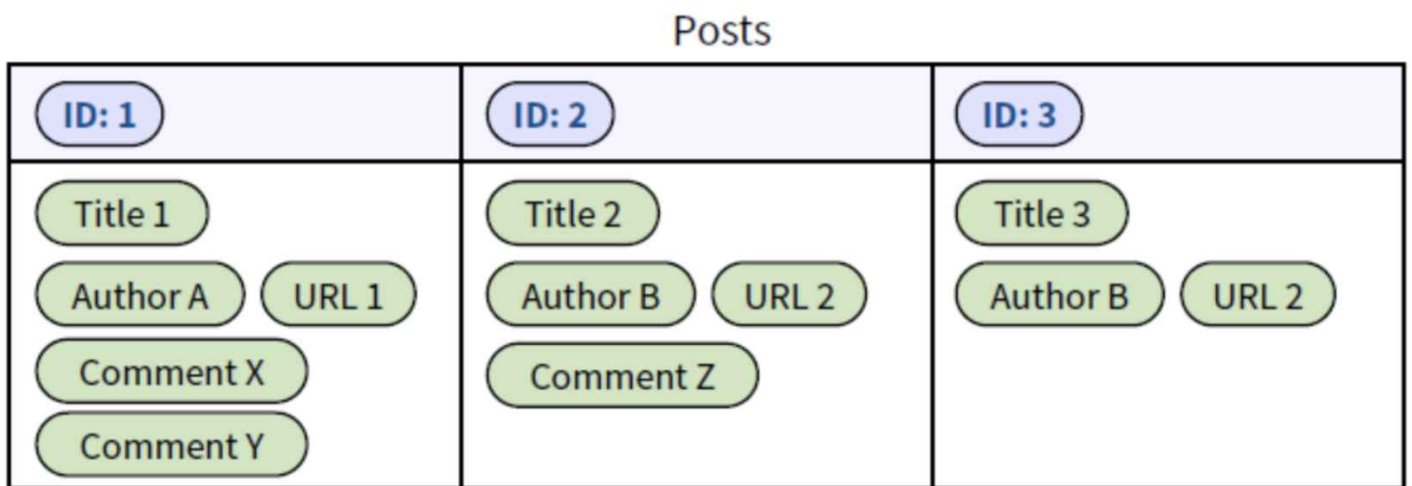
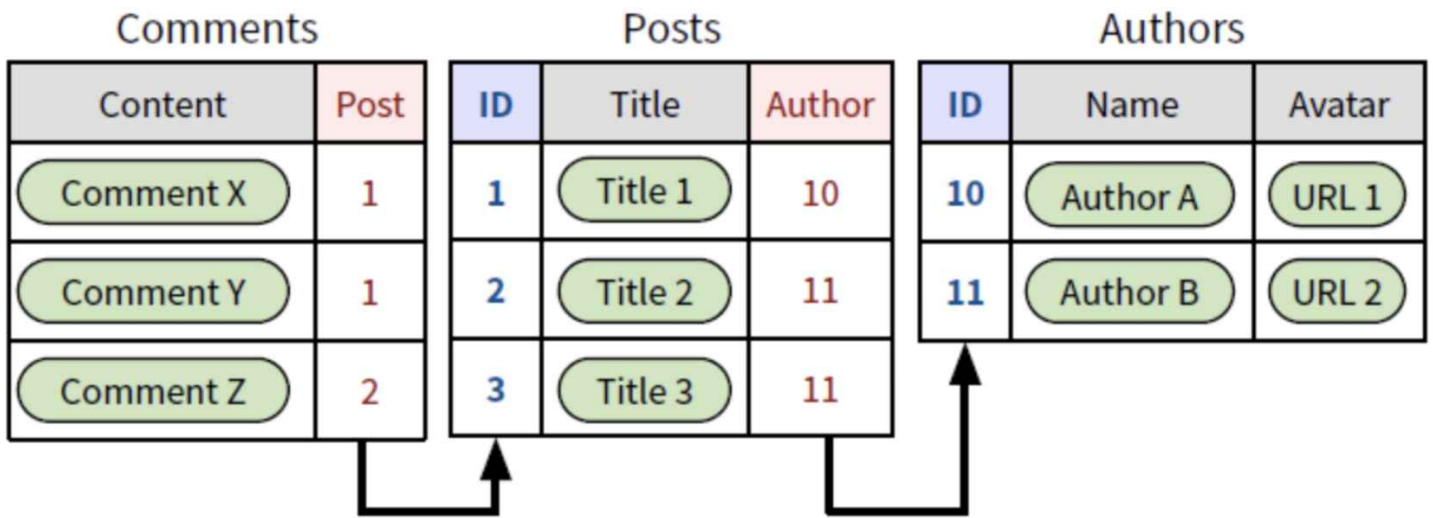
[NoSQL - Wikipedia](#)



Courtesy of <http://geek-and-poke.com>.

Document Stores

- The most widely known type of NoSQL database, e.g. *Elasticsearch*, *MongoDB*.
- Data entries are kept *exactly the way they are needed* by the application.
- To compare the tabular way (top) and the document way (bottom) to store posts in a blog:



- The non-relational model *expects* duplication.
- While duplicated data is hard to manage and update, by grouping relevant data together, document stores offer more flexibility:
 - No joins are needed.
 - No fixed schemas are needed.
 - Each data entry can have its own configuration of fields.
- There are no "tables" and "rows" in document stores. Instead, a data entry is called a `document`. Related documents are grouped in a `collection`.
- Documents have a primary key field, so relationships across documents are possible.
- But JOINS are not optimal on document stores. Sometimes they are not even implemented.
- NoSQL databases create indexes for primary key fields. You can also define additional indexes for fields that are often queried or sorted.

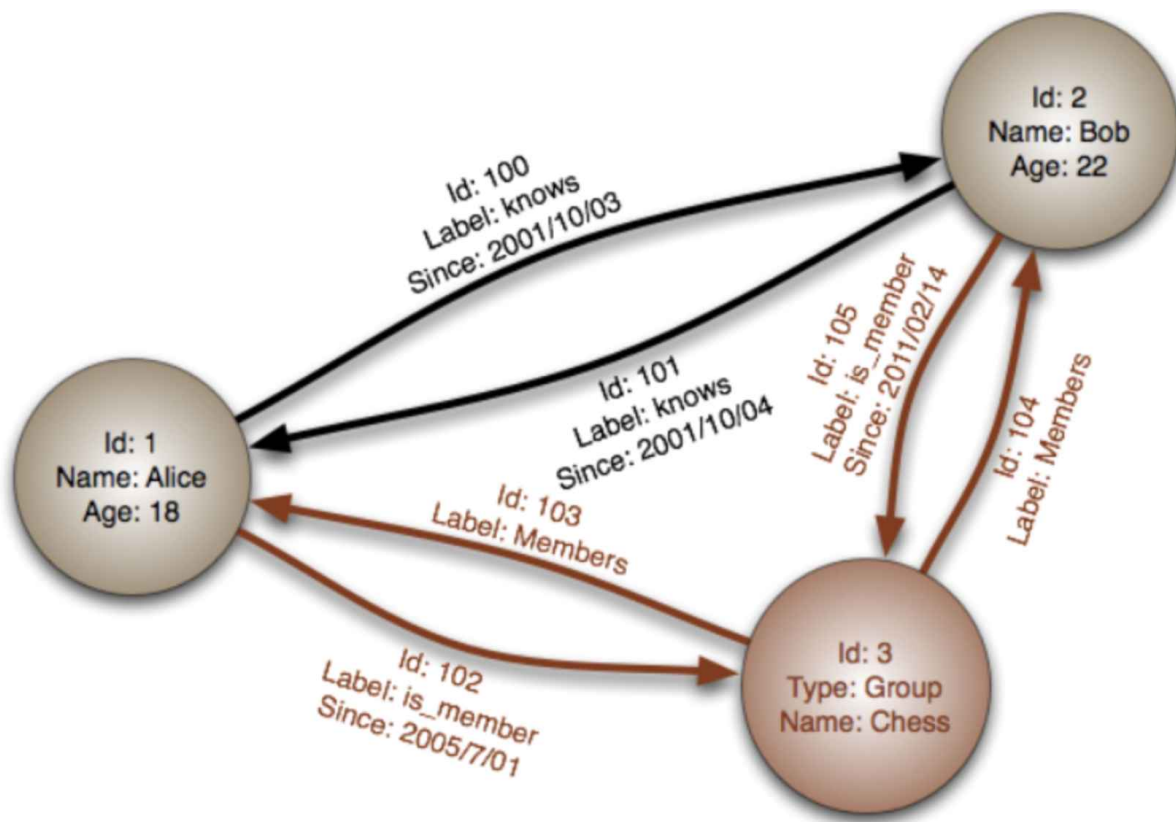
Key-Value Stores

- The simplest form or organized, persistent data storage, mainly used for `caching`, like `Redis`.

- By caching results of slow operations and retrieving them from the key-value store, we reduce the load of the database and speed up responses, especially for high-traffic websites.

Graph Databases

- The most flexible type of database, e.g. **Neo4j** and **OrientDB**.
 - Data entries are stored as nodes, and relationships as edges.
- Nodes are not tied to a fixed schema and can store data flexibly.
- The graph structure makes it efficient to work with data entries according to their relationships, e.g. finding the fastest or shortest route between two given locations.



SQL vs NoSQL

- Whether an application should use SQL or NoSQL databases depends on the business context.
- **NoSQL**
 - Generally, NoSQL is preferred for:
 - Graph or hierarchical data
 - Data sets which **change fast** and **do not have fixed schemas**
 - In terms of use cases, this might translate to social networks, Content Management Systems (CMS) or streaming analytics.
- **SQL**

- SQL is more appropriate when data is:
 - Conceptually modeled as tabular
 - In transactional systems where **consistency is critical**
- Think accounting systems, sales and inventory systems, and payment systems.

Serialization Formats

- How can we store data outside of our database, in a format that is interoperable across different systems? For instance, we might want to backup the data, or export it to another system.
- To do this, the data has to go through a process called `serialization`, where it is transformed according to an `encoding format`. The resulting file can be understood by any system that supports that encoding format.
- Commonly used encoding formats for data serialization include:
 - `SQL` - *Structured Query Language*. The most common format for serializing relational databases. An SQL-serialized file consists of a series of SQL commands that replicate the database and all its details. Most **relational database systems include a "dump" command to export such file of your database, and a "restore" command to import such "dump file" back into the database system.**
 - `XML` - *eXtensible Markup Language*. A way to represent structured data that doesn't depend on the relational model or a database system implementation. XML was created to be interoperable among diverse computing systems, and to describe the structure and complexity of data.

```
1 <empinfo>
2   <employees>
3     <employee>
4       <name>John Doe</name>
5       <age>40</age>
6       <gender>Male</gender>
7     </employee>
8     <employee>
9       <name>Jane Doe</name>
10      <age>30</age>
11      <gender>Female</gender>
12    </employee>
13    <employee>
14      <name>Joan Doe</name>
```



```

15         <age>35</age>
16         <gender>Female</gender>
17     </employee>
18 </employees>
19 </empinfo>

```

- **JSON** - *JavaScript Object Notation*. The serializing format most systems are converging to. It can represent relational and non-relational data, in an intuitive way to developers.

```

1 {
2   "empinfo": {
3     "employees": [
4       {
5         "name": "John",
6         "age": 40,
7         "gender": "Male"
8       },
9       {
10        "name": "Jane Doe",
11        "age": 30,
12        "gender": "Female"
13      },
14      {
15        "name": "Joan Doe",
16        "age": 35,
17        "gender": "Female"
18      }
19    ]
20  }
21 }

```

- **CSV** - *Comma-Separated Values*. Arguably the simplest format for data exchange. Data is stored in textual form, with one data element per line. The properties of each data element are separated by a comma, or some other character that doesn't occur in the data. CSV is useful for dumping simple data, but it gets messy to represent complex data using it.

```

1 name,age,gender
2 John Doe,40,Male
3 Jane Doe,30,Female
4 Joan Doe,35,Female

```

- **Protobuf** - *Protocol Buffers*. Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data - think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.

```
1 message EmpInfo {
2     repeated Employee employees = 1;
3 }
4
5 message Employee {
6     required string name = 1;
7     required int32 age = 2;
8     required string gender = 3;
9     optional string phoneNumber = 4;
10 }
```

Questions

- What is the purpose of DBMS?
- What are the core concepts supported by DBMS?
- What does the DBMS structure comprise?
- What is the difference between relational and non-relational databases?
- What are the various types of NoSQL databases?
- What are the various types of serialization formats?