Part II

Overview of Database Systems
Why a Database System?

Why not simply use OS files to keep the data?

Suppose you own a cocktail bar. You want to keep inventory of your cocktail ingredients:

<table>
<thead>
<tr>
<th>Ingredients</th>
<th>Alcohol</th>
<th>InStock</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange Juice</td>
<td>0.0</td>
<td>12</td>
<td>2.99</td>
</tr>
<tr>
<td>Campari</td>
<td>25.0</td>
<td>5</td>
<td>12.95</td>
</tr>
<tr>
<td>Bacardi</td>
<td>37.5</td>
<td>3</td>
<td>16.98</td>
</tr>
</tbody>
</table>

One way of storing these data could be:

Orange Juice:0.0:12:2.99
Campari:25.0:5:12.95
Bacardi:37.5:3:16.98
Why a Database System?

What do you think of this approach?
(Think of problems that might occur. Judge the effort to solve them.)
Databases provide **abstractions** to avoid many of these problems:

Some databases work on top of operating system files, others access raw disk partitions or network-attached storage directly.
Abstraction 1: Data Model

- Rather than exposing bits and bytes of the underlying storage, databases present a high-level **data model** to the outside.
- By far the most popular data model today is the **relational model**:

<table>
<thead>
<tr>
<th>Ingredients</th>
<th>Name</th>
<th>Alcohol</th>
<th>InStock</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orange Juice</td>
<td>0.0</td>
<td>12</td>
<td>2.99</td>
</tr>
<tr>
<td></td>
<td>Campari</td>
<td>25.0</td>
<td>5</td>
<td>12.95</td>
</tr>
<tr>
<td></td>
<td>Bacardi</td>
<td>37.5</td>
<td>3</td>
<td>16.98</td>
</tr>
</tbody>
</table>

- Other data models: hierarchical model, object-oriented model, object-relational model, XML.
Database Schema:

- Formal definition of the **structure** of the database contents.
- **Defined once** (when database is created).
- Restricts the possible contents that can be put into the database.

~~ In a programming language, this corresponds to the **declaration** of a variable:

```
unsigned int i;
```

Database State (Instance of the Schema):

- Contains the **actual data**, structured according to the schema.
- **Changes often**

~~ Current **value** of a variable in a programming language:

```
i = i + 42;
```
Physical vs. Conceptual Schema

- What we just saw is only the **user’s understanding** of the data representation, the **conceptual schema** (also: logical schema).
- The **physical representation** is at the DBMS’s discretion.

The physical schema may use different **file organizations** or **access mechanisms** (indexes) to improve performance.
The **external schema** provides **views** on top of the conceptual schema.

- Tailored to different users or applications
- Alternative data models (*e.g.*, XML over relational data)
The separation of views on the same data allows for data independence.

**Physical data independence:**
- Change physical storage layout or create indexes.
  → Changes invisible to conceptual schema (and external schema)—only performance might have improved.

**Logical data independence:**
- Change the logical representation of the data, but leave external schema intact.
  → Existing applications still work as before.
Example: Logical Data Independence

As a bar owner, you want to better track where your cocktail ingredients are, so you create a table **Availabilities**:

<table>
<thead>
<tr>
<th>Name</th>
<th>InStock</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orange Juice</td>
<td>3</td>
<td>refrigerator</td>
</tr>
<tr>
<td>Orange Juice</td>
<td>9</td>
<td>warehouse</td>
</tr>
<tr>
<td>Campari</td>
<td>2</td>
<td>refrigerator</td>
</tr>
</tbody>
</table>

The **InStock** field can now be removed from the **Ingredients** table and computed on-demand instead. Applications will not notice the change.

```
ALTER TABLE IngredientsConceptual DROP COLUMN InStock;
CREATE VIEW IngredientsExternal AS
    SELECT i.Name, i.Alcohol, SUM(a.InStock) AS InStock, i.Price
    FROM IngredientsConceptual AS i, Availabilities AS a
    WHERE i.Name = a.Name
    GROUP BY i.Name, i.Alcohol, i.Price
```
Databases offer **declarative query languages**.

- Specify **which data** should be retrieved, rather than **how** they should be retrieved.

**Example:** Names and prices of non-alcoholic drinks, ordered by **Name**, expressed in **SQL (Structured Query Language)**:

```
SELECT Name, Price
FROM Ingredients
WHERE Alcohol = 0
ORDER BY Name
```

→ Compare this to a **program** that you’d have to write if you used OS files for storage.

→ Physical data independence would not allow use of indexes anyway.
Query Optimization

- Declarative languages *need* powerful optimizers.
- Declarative languages *allow* powerful optimizers.

Today’s query optimizers *are* really powerful.
- This releases you from worrying how you write your query “most efficiently,” but focus on the application problem instead.

Additional benefit:
- Once written, your query/application will automatically benefit from improvements in the physical schema, the database software, or the underlying hardware.
Databases help to keep the **integrity** of stored data.

- Sophisticated **access control** mechanisms support very fine-granular restrictions to read or modify data.

- **Integrity constraints** can be defined along with the conceptual schema and ensure plausibility of the stored data.

```
ALTER TABLE Availabilities
ADD FOREIGN KEY (Name)
REFERENCES Ingredients (Name)
```

- **Consistency**: The database system will check integrity constraints and ensures that every user sees a consistent database state.
Abstraction 4: Multi-User Support

Databases shield the programmer from many multi-user issues.

- Give each user the illusion that he/she is the only user at any time.
- Perform locking, and conflict detection automatically.

At the same time, the database helps handling problems or conflicts.

- Atomicity: a database transaction (i.e., a sequence of SQL commands) is executed atomically ("all or nothing" principle).
- Isolation: transactions cannot see the effects of co-running transactions; every user has the impression he/she is alone on the system.
Abstraction 5: Tolerance to Failures

Databases ensure **durability** of data modifications.

- A successful transaction will **never** get lost, whatever **failure** the system might encounter, including
  - **software crashes** on client or server side (also: OS crash);
  - **hardware failures** (hard disk crash);
  - **catastrophic failures** (fire, water, etc.).

- The database will apply necessary measures to guarantee durability:
  - **redundant storage** (write-ahead logging),
  - **backup/recovery** mechanisms.

- **Durability**: The effect of a successful transaction remain persistent and may not be undone for system reasons.
Search engines are related, but serve a different purpose.

<table>
<thead>
<tr>
<th><strong>database</strong></th>
<th><strong>search engine</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>structured data (e.g., relational)</td>
<td>unstructured data (“documents”)</td>
</tr>
<tr>
<td>tailor-made query language</td>
<td>natural language interface</td>
</tr>
<tr>
<td>expressive query language</td>
<td>limited expressiveness</td>
</tr>
<tr>
<td>exact-match queries</td>
<td>ranking-based queries (top-(n))</td>
</tr>
<tr>
<td>deterministic result</td>
<td>probabilistic result</td>
</tr>
</tbody>
</table>

Application demands increasingly fall between those two extremes.

→ Content-aware search (e.g., email search)
→ Full-text indexes in databases
→ Semi-structured data (e.g., XML)
Key-value stores are not databases in the sense discussed here.

- *E.g.*, Cassandra, Dynamo, Memcached
- Designed for **massive scalability** in **cloud environments**
  - **CAP Theorem**: Cannot have such scalability **and** strong transaction guarantees.
- **Much** simpler data/query model: key/value lookups only
  - Think of them as a back-end on top of which database functionality could be built.
Databases are typically used in a **three-tier architecture**.

A database system forms the heart of virtually any business application!