# Databases

## Storing Data

| Format | Where |
|---|---|
| Fixed field record | Used in punch cards, each record can store 80 characters, some of the 80 characters are allocated to different fields. |
| Comma separted value record | More flexible than fixed field records, as there is no character limit to the value of a field. |

A **simple associative store** will have the **API formal specification**.

```
store: string * string -> unit
retrieve: string -> string option (* either the value stored, or nothing at all *)
```

**Definitions**

| Keyword | Meaning |
|---|---|
| Value | What is being stored. |
| Field | A place to hold a value. |
| Schema | A strongly typed specification of the database. |
| Key | The field(s) used to locate a record. |
| Index | A derived structure used to quickly find relevant record. |
| Query | A lookup function. |
| Update | A modification of data. |
| Transaction | An atomic change of a set of field. |

An update is often implemented as a transaction.

## Abstraction and Interfaces

We hope to have a fairly narrow interface that can map onto many operations. In an ideal world:

- All operations are done through the interface.
- The interface never change.

But in real world, large database projects are often a mess, and fixing the mess sometimes requires changing the interface. Changing interfaces break applications!

### Logical Arrangement of a Database

The **DBMS** uses the **disc drive**, which is an abstraction over the **secondary storage**. This gives a consistent, nonchanging API which applications can use.

This API is often in form of SQL queries, SQL injection is an example of insecurity in the interface.

### Operating System View of a Database

The **DBMS** is a process running in userspace.

- An **application** communicates with the **DBMS** through **kernel space communication**.
- Then the **DBMS** use **kernel space drivers** to access the **DBMS**.

An alternative implementation have the **DBMS** access a non-OS partition directly.

Data can be stored in primary store, secondary store, or distributed.

**Definition**

**Big data** is data too big to fit in primary store.

## DBMS Operations

| Operation | Description |
|---|---|
| **C**reate | Insert new data |
| **R**ead | Query the database |
| **U**pdate | Modify objects in the database |
| **D**elete | Remove data |

### Management Operations

- Create/change schema
- Create views
- Indexing/stats generation
- Reorganise data layout and backups

### Amount of Writing

Database implementation choice depends on the amount of writing.

- A database can have a lot of lookups but rarely changes (e.g. library catalogue).
- **Transaction optimised**: concurrent queries and updates, they will affect the consistency of reads.

**Definition**

**Atomicity** is where changes are apparent to all users at once. An overhead is needed to achieve this.

- **Append only journal**: inverse of a transaction optimised DB, data is never updated.

### Consistency Checks

Consistency rules includes value range check, foreign key referential integrity, value atomicity (all values are atomic), entity integrity (no missing fields).

## Types of Databases

### Relational Database

Consists of 2D tables.

- One row per record (a.k.a. a tuple), each with a number of fields.
- A table can have a schema.
- The ordering of the fields are unimportant.

### Distributed Databases

Database can be spread over multiple machines.

| Benefit | Description |
|---|---|
| Scalability | Dataset may be too large for a single machine. |
| Fault tolerance | Service can survive the failure of some machines. |
| Lower latency | Data can be located closer to the users. |

Downsides: overhead after an update to provide a consistent view.

| Keyword | Meaning |
|---|---|
| **C**onsistency | All reads return data that is up to date. |
| **A**ccessibility | All clients can find some replica of the data. |
| **P**artition tolerance | The system continues to operate despite arbitrary message loss or failure of parts of the system. |

It is impossible to achieve all 3 in a distributed database. *(why?)*

System that offers **eventual consistency** will eventually reach a consistent state once activity ceases.

## ER Model

ER model is an implementation independent technique of describing the data we store in a database.

**Definitions**

| Keyword | Meaning | Symbol |
|---|---|---|
| Entity | Model things in the real world | Rectangle |
| Attributes | Represent properties | Oval |
| Key | Uniquely identifies an entity instance. | Underline |

A key can be **composite**. Examples of **natural keys** are a person's name and nation ID unumber. They may not be unique, it is often better to use a **synthetic key** - beware of using keys that are out of your control.

**Definition**
A **synthetic key** is auto-generated by the database and only has meaning within the database.

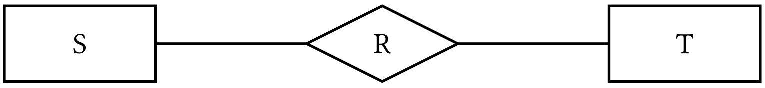
The **scope** of the model is the limited subset of all the real world attributes of the object.

### Relationship Cardinalities

### Many-to-many Relationships

- Represented in undecorated lines.
- Any $S$ can be related to 0 or more $T$.
- Any $T$ can be related to 0 or more $S$.

Note relations can also have attributes.



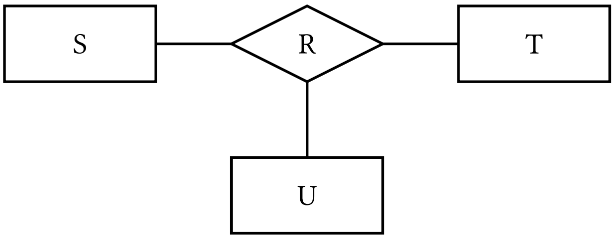
### Rules for Modelling

- An attributes exists at most once for any entity or relation.
- Rule of atomicity: every value in a box must be atomic (a.k.k. 1NF)

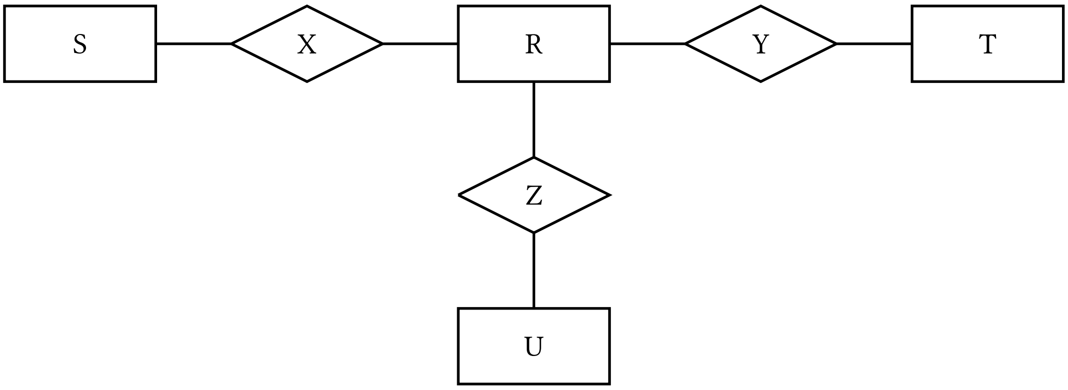
Do not put a comma separated list in the value, more likely than not we will need to break up the list again when searching, which requires text processing.

### Tenary Relationships

They may be appropriate, depending on use.



What about 3 binary relationships?



Which one is appropriate depends on the situation, each entity should represent a real world object.

- **Many-to-one** relationships are denoted by an arrow towards the one end.
- **One-to-one** relationships are denoted by two arrow, one at each end.

If $R$ is both many-to-one and one-to-many between $S$ and $T$, it is one-to-one between $S$ and $T$.

**Definition**
A **bijection** is where every member in one set is related to one member of the other set.

One-to-one cardinality doesn't mean one-to-one correspondence: not all elements in $S$ are related to an element in $T$.

### Weak Entity

**Definition**
The **weak entity** cannot exist without the **strong entity** existing.

- E.g. an alternative title (weak) for a movie (strong).
- Has **discriminators**, not keys. Combined with the **key** from the strong entity uniquely defines the weak entity.

### Entity Hierarchy (OO-like)

- Subentities inherit the attribues and relations of the parent entity.
- Multiple inheritance is possible.
- We represent the relation with an *IS A* relation (upside down triangle in diagram).

## The Relational Model

Before the relational model, you need to know about the data's low level representation to write a database application.

- We give the user a model of data and language for manipulating the data that is independent of the implementation details.
- The model is based on mathematical relations.

### Mathematical Relations

**Definitions**

- The **cartesian product** is the set of all possible pairs.

$$S \times T = \{(s, t) \mid s \in S, t \in T\}$$

- A **binary relation** of $S \times T$ is any set $R$ with

$$R \subseteq S \times T$$

- $R$ is a **binary relation**.
- $S$ and $T$ are referred to as **domains**.

We are interested in finite relations $R$ that are explicitly stored in a database, which *does not include* infinite relations like the modulus. (e.g. $6 \bmod 5 = 1$)

### N-ary Relations

If we have $n$ domains, then $n$-ary relation $R$ is the set

$$R \subseteq S_1 \times S_2 \times \cdots \times S_n = \{(s_1, s_2, ..., s_n) \mid s_1 \in S_1, s_2 \in S_2, ..., s_n \in S_n\}$$

**Definition**

**Tabular presentation** of a relation is where each tuple has a column number.

| **1** | **2** | **...** | $n$ |
|---|---|---|---|
| $a_1$ | $b_1$ | ... | $x_1$ |
| $a_2$ | $b_2$ | ... | $x_2$ |
| $a_3$ | $b_3$ | ... | $x_3$ |
| $a_4$ | $b_4$ | ... | $x_4$ |

### Named Columns

Because referring to tuples by column number is annoying.

- Associate an attribute name $A_i$ with each domain in $S_i$.
- Use records instead of tuples.

**Definition**

A **database relation** is the finite set

$$R \subseteq \{\{(A_1, s_1), (A_2, s_2), ..., (A_n, s_n)\} \mid s_i \in S_i\}$$

We specify $R$'s schema as $R(A_1 : S_1, A_2 : S_2, ...A_n : S_n)$.

### Example: Students

Using the schema *Students(name: string, sid: string, age: integer)*:

```
Students = {
  { (name, "sirius"), (sid: "12345"), (age: 19) }
}
```

## Query Language

- The input of a query language is a collection of relatin instances $R_1, R_2, ..., R_k$.
- The output is a single relational instance $Q(R_1, R_2, ..., R_k)$.

We want to create a high level query language $Q$ that is independent of the data, there are many ways to do that.

### The Relational Algebra Abstract Syntax

It is generally a tree structure

$Q ::=$

| | |
|---|---|
| $\mid R$ | base relation |
| $\mid \sigma_{p(Q)}$ | selection |
| $\mid \pi_{x(Q)}$ | projection |
| $\mid Q \times Q$ | product |
| $\mid Q - Q$ | difference |
| $\mid Q \cup Q$ | union |
| $\mid Q \cap Q$ | intersection |
| $\mid \rho_{M(Q)}$ | rename |

- $p : x \mapsto$ bool is a boolaen predicate.
- $X = \{A_1, A_2, ...\}$ is a vector (a set of attributes).
- $M = \{A_1 \mapsto B_1, A_2 \mapsto B_2, ...\}$ is a list of mappings that takes in one attribute name, and output another attribute name.

**Definition**

$Q$ is **well formed** if all attribute names of the result are distinct.

- $Q_1 \times Q_2$ requires the subqueries to share no column names.
- $Q_1 \cup Q_2$ requires the subqueries to share all column names.

### Mapping RA to SQL

RA is based directly on the underlying set theory.

- Formal semantics/specification works by mapping the query language back to set theory.

### Operators

| RA | SQL | Notes |
|---|---|---|
| $\sigma_{A>12}(R)$ | `SELECT DISTINCT *`<br>`FROM R`<br>`WHERE R.A > 12` | - |
| $\pi_{B,C}(R)$ | `SELECT DISTINCT B, C`<br>`FROM R` | - |
| $\rho_{\{B\mapsto E, D\mapsto F\}}(R)$ | `SELECT A, B as E, C, D as F`<br>`FROM R` | If we want to swap 2 rows, we will need to rename them one at a time. |
| $R \cup S$ | `(SELECT * FROM R) UNION`<br>`(SELECT * FROM S)` | $R$ and $S$ must have the same schema. |
| $R \cap S$ | `(SELECT * FROM R) INTERSECT`<br>`(SELECT * FROM S)` | |
| $R - S$ | `(SELECT * FROM R) EXCEPT`<br>`(SELECT * FROM S)` | In set theory, $R - S = R \cap (-S)$, but this doesn't make sense on databases, as $-S$ would mean everything that is not in $S$. |
| $R \times S$ | `SELECT A, B, C, D`<br>`FROM R`<br>`CROSS JOIN S` | In set theory, the cartesian product is a set of tuples. In SQL, that tuple is flattened. |

**Note**

The `DISTINCT` keyword removes duplicates: each $A, B, C, D$ might be unique, but $B, C$ may be not. Not being distinct is fine in SQL but not fine in set theory as a set contains no duplicates.

### The Natural Join

If we ignore domain types and write a relation schema as $R(A)$, where $A = \{A_1, A_2, ...\}$.

- $R(A, B) = R(A \cup B)$ assumes $A \cap B = \emptyset$
- $u.[A] = v.[A]$ means $u.A_1 = v.A_1 \wedge u.A_2 = v.A_2 \wedge ...$.
  - For the fields specified, the two records completely agree with each other.

For $R(A, B)$ and $S(B, C)$, define the natural join $R \bowtie S$.

$$R \bowtie S = \{\ t \mid (\exists u \in R, v \in S)\ u.[B] = v.[B],\ t = u.[A] \cup u.[B] \cup v.[C]\ \}$$

```sql
SELECT *
FROM R
NATURAL JOIN S
```

- If `NULL` values exists, then equality gets more complicated (how?), there are further variations of natural joins (left/right/inner/outer).
- If we want to use a custom predicate, use `WHERE` instead of `NATURAL`.