# Databases And Advanced Data Techniques (CM3010)

**Course Notes**

Felipe Balbi

July 15, 2021

# Contents

# Week 1

Key Concepts

- Find, describe and evaluate sources of data
- Understand different forms in which data may come
- Evaluate data-related access and reuse rights

## 1.005 Reading list

- Chen, P. 'The Entity-Relationship Model – Toward a Unified View of Data', ACM Transactions on Database Systems 1(1) 1976, pp.9–36.
- Codd, E. 'A relational model of data for large shared data banks', Comms of the ACM 13/6 1970, pp.377–87.
- Codd, E. 'Normalized data base structure: a brief tutorial'. In Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET'71). Association for Computing Machinery, New York, NY, USA (1971) pp.1–17
- Date, C.J. Database Design and Relational Theory. (Healdsburg, CA: Apress, 2019) Chapter 4. FDs and BCNF (informal)
- Härder, T and A. Reuter 'Principles of Transaction-Oriented Database Recovery', ACM Surveys, 15/4 1983
- Katie Rawson and Trevos Muñoz, 'Against Cleaning' from Matthew K. Gold and Lauren F. Klein Debates in the Digital Humanities, 5 (University of Minnesota Press, 2019).
- Lewis, D. CO2209 Database systems

## 1.101 Where does data come from?

Data can come from different sources:

**New Data** created for the sole purpose of the current application

**Pre-existing Data** data that already existed prior to the application being created. Perhaps it's internal *legacy* data, or it's external data that can be acquired from another supplier.

When it comes to new data, we can take different approaches:

**Adding data on-demand** For example, a hairdresser has bookings with clients. Either of these appointments is a new datum that gets added to the database *on-demand*, i.e. only the customer makes an appointment.

**Bulk data entry** Some systems can't afford to have only parts of the data available. In such cases, we can either pay for data entry services or rely on some form of crowd-sourcing.

**Pre-existing data** Whenever we have pre-existing data, it usually needs to be manipulated somehow in order to fit the new system. Some forms of data manipulation are:

**Extraction** data may already be in a spreadsheet or database and needs to be recovered, or extracted from the original source.

**Conversion** data may need to be converted into a new format or structure in order to fit new requirements.

**Cleaning** data may contain erroneous or unnecessary information. These need to be removed in order to prevent problems.

External sources of data are interesting because they amortize the cost of data entry or quality checks. When data is *purchased* from a supplier, it comes pre-cleaned and in a format that's easy to consume. Moreover, we can also have the opportunity of acquiring data produced by experts in a given field.

Conversely, when we acquire data from an external source, we relinquish control over the quality of the data and its structure. The data may also be incomplete and/or ambiguous from our point view; i.e. the level of detail to which a particular piece of information is encoded may be different from what we need. As a final concern, there may be concerns of trustworthiness with regards to the data.

## 1.103 Ordering some data: What's on the menu?

- Post 1: Trevor Munoz, 'What IS on the menu'
- Post 2: Trevor Munoz, 'Refining the problem'

## 1.105 What does your data look like?

When modelling real-life data, we must consider what sort of information is necessary for the application.

To motivate the problem, we look at the example of a book. The data required for a book may be:

| | |
|---|---|
| **Type** | Book |
| **Weight** | 557g |
| **Height** | 172mm |
| **Colour** | Red and Green |
| **Title** | Gardener's Calendar |
| **Authors** | Thomas Mawe, John Abercrombie |
| **Date** | 1803 |
| **Edition** | $17^{\text{th}}$ |

Some questions arise when it comes to which form of e.g. the title to store. From the point of view of finding it in a shelf "Gardener's Calendar" is enough, from the point of view of comparison against other similar titles, a long form may be required.

## 1.201 Bringing data sources together

- Linked Jazz
- Pratt Institute, How Mapping Relationships Between Jazz Musicians Elevates Unsung Histories

## 1.203 Licenses, sharing and ethics

In academic and government circles, it's common to make data as openly available as possible. That, however, doesn't apply to all parts of government of commercial world. There are legal restrictions regarding the use of data which need to be considered.

The Linked Open Data Cloud project produces a graph of all the data openly available published in the Linked Data format. Considering the size of the graph which contains but a subset of all openly available data, the question to ask is *Why is so much data being shared for free if information is so valuable?*

To put into perspective, a furniture catalog from any given furniture company will contain many details about every item: price, sizes, materials, photos. In principle, the furniture could be copied from information that can be gathered from catalogues and manuals. However, the furniture company needs their products to be easy to find if they want to sell them. The same argument can be used for many other industries: music industry, electronics, streaming services, etc.

To summarize some of the reasons to share open data:

- To drive sales
- For the common good
- Contract requirements

- Interoperability

Conversely, here are some reasons **not** to share open data:

- Restrictions on source data
- Control of use
- Value of the data

## 1.204 Licensing

- Alex Ball, How to License research data

# Week 2

Key Concepts

- Find, describe and evaluate sources of data
- Understand different forms in which data may come
- Evaluate data-related access and reuse rights

## 1.301 What shape is your data? Introduction

Data is structured in some form, and we have to be concerned about that. There are different *levels* of structure which can be considered:

**Programming Languages** Data types (`float`, `int`, `double`, etc) impose a certain structure to the data.

**Data Models** Relations between different data. Think databases.

**Data Serialization** Data formats used for transmission using e.g. a network connection.

**Exchange Protocols** Some form of standardization for information exchange using e.g. Unix Sockets, Named Pipes, shared memory or similar methods.

**User Interfaces** Data is user interfaces is structured in a way that's comfortable for humans to consume.

Some of the *shapes* of data we will deal with are:

- Tables
- Trees
- Graphs
- Media (raw data)
- Documents & objects

Table 1: Sample Table

| **Food** | **Water (g)** | **Fat (g)** |
|---|---|---|
| Avocado | 72.5 | 19.5 |
| Butter | 14.9 | 82.2 |

## 1.302 What shape is your data? Tables

A table has cells with a number of rows and columns. In our case, every row represents a *thing*. Each column represents a type of information about that *thing*. Table 1 shows an example of such a table:

Tables are easy to understand and structure. They're also very direct in how they communicate information. Tables are very important to Relational Databases. However, they're not very good at communicating or structuring data that branches or has hierarchy. A better suited representation for such data would be Trees.

## 1.304 What shape is your data? Trees

A tree is Computer Science is based on the metaphor of a real tree. Figure 1 below shows an example of a simple tree structure. Every tree has a root node, every branch in the tree has a path to the root.

Some vocabulary is necessary, the following refers to the tree from figure 1.

- The *root* of the tree is node $a$
- Nodes $e$, $g$, $i$, $k$ $l$, $m$, $n$, $o$, $p$, $r$, $s$, and $u$ are *leaf nodes*
- Node $f$ is a parent of $l$, $m$, and $n$
- Nodes $l$, $m$, and $n$ are children of node $f$
- Nodes $a$, and $b$ are ancestor of nodes $l$, $m$, and $n$
- Nodes $i$, $j$, and $k$ are sibblings
- Nodes $b$, $c$, $d$, $h$, and others are internal nodes

## 1.306 What shape is your data? Other

One limitation of trees is that each and every node can have a single parent node. What happens when we need to represent a node with more than one parent? Perhaps we can reach the same child node through different paths. If we were dealing with a filesystem, whenever we add a *symbolic link* to a file, we would break the representation of the filesystem as a tree. It's clear we need another structure to represent these sorts of structures. That structure is a graph.

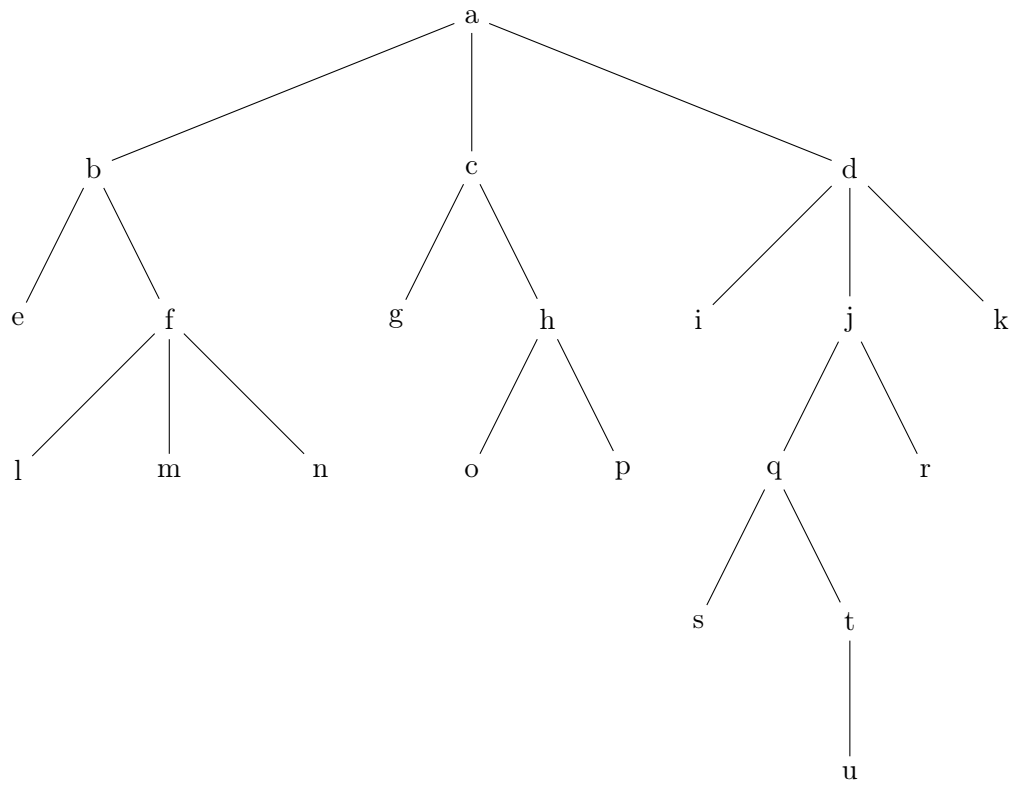


Figure 1: Sample Tree

Figure 2 shows a $K_8$ complete graph. A complete graph is that where each vertex is connected to every other vertex. The vertices in a graph could be web pages and the edges could be links between them, or perhaps each node is a file with the edges being a filesystem path.

Blobs are *raw* data representations without a perceivable structure. Raw sound samples fall into this category. Features are pieces of information derived from blobs, for example the sample rate from a raw audio file.

Table 2 shows a summary of the structures discussed so far:

Table 2: Summary of structures

| **Structure** | **Description** |
|---|---|
| Table | General purpose |
| Tree | Heterogenous and hierarchical, structured data |
| Graph | Heterogenous, non-hierarchical, structured data |
| Blobs | Inaccessible data for storage |
| Features | Searchable information derived from blobs |
| Documents | Rich, but not interrelated data |

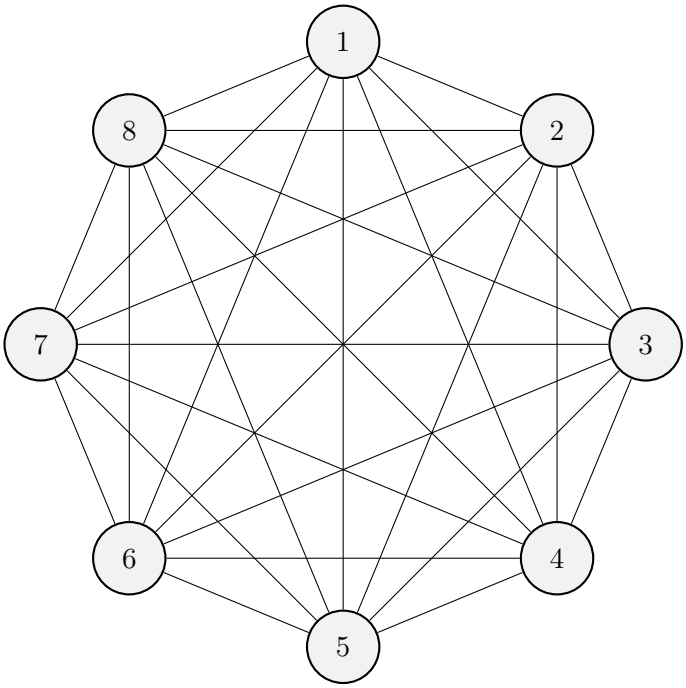


Figure 2: Complete graph

## 1.402 Further reading

- Katie Rawson and Trevos Muñoz, 'Against Cleaning' from Matthew K. Gold and Lauren F. Klein Debates in the Digital Humanities, 5 (University of Minnesota Press, 2019).

# Week 3

Key Concepts

- Create and explore relational databases using SQL
- Design a database using Entity/Relationship diagrams
- Explain core concepts of relations and relational theory

## 2.001 Welcome to Relational Databases

A relational database implements the Relational Model. By model we mean by model is that it serves as an abstraction to the complex real-world. Usually we will abstract data.

A relational database uses tables to represent data. Relation, in this case, as defined by E. F. Codd., is a set of tuples $(d_1, \ldots, d_n)$ where each element $d_j$ is a member of $D_j$

Relational Databases are an implementation of Relational Algebra, a theory for modelling data and defining queries on such data.

The following, summarizes a set of rules for Relational Databases.

1. Everything is a **relation**
    - All operations use the relational model
    - All data is represented and accessed as relations
    - Table and database structure is accessed and altered as relations
2. The system is unnaffected by its implementation
    - If the hardware changes
    - If the operating system changes
    - If the disks are replaced
    - If data is distributed

The Relational Model is not the same as the Entity Relationship Model. An ER Model helps us model concepts, usually as part of the design of a Relational Database.

SQL is a *partial* of the relational model.

## 2.101 Drawing a database I: Basic Entity-Relationship diagrams

An *Entity* is the thing we want to model, it must be uniquely idenfiable and it may have attributes. Of this, there are two subtypes:

**Weak Entity** its existence depends on the continued existence of other entities. For example, a customer's bank account depends on the existence of the account holder. This entity type is depicted in figure 3.

**Strong Entity** the one which is **not** weak. This entity type is depicted in figure 4.


Figure 3: Weak Entity Type


Figure 4: Strong Entity Type

An *Attribute* is an information that describes one aspect of an entity. Attributes can be characterised in various ways (described below and depicted in figure 5):

**Simple vs composite** A **simple attribute** is atomic or scalar (a simple integer or string). A **composite attribute** has internal structure that can be broken down into further attributes. For example, a *Date* attribute can be broken down into *day*, *month*, and *year*.

**Single or multi-valued** A **single-valued attribute** is one that won't change. For example a student is unlikely to have multiple full names at the same time. A **multi-valued attribute** is that which an entity can have multiples of it. It's depicted in a diagram with a double border line. An easy example is a phone number: a student can have multiple phone numbers.

**Base or derived** a **derived attribute** can be deduced from other attributes already present. They are depicted in a diagram with a dotted or dashed border line. An example would be someone's age can be computed from their date of birth. A **base attribute** cannot be deduced from other attributes.

**Primary key** an attribute that uniquely identifies an instance of the entity type. In a diagram, it's shown underlined.

A *Relationship* is a connection or dependency between two entities. Entities involved in a relationship are referred to as *participants*. A relationship is depicted as a diamond labelled with the name of the relationship. If one entity in the relationship is strong and the other weak, we draw the diamond with double line. Figure 6 shows the relationship types in a simple ER Diagram.

## 2.104 Speaking to databases I: Basic SQL

*SQL* has commands for manipulating structure, such as `CREATE`, `DROP`, `TRUNCATE`, `ALTER`; as well as commands for manipulating data, such as `INSERT`, `SELECT`, `UPDATE`, and `DELETE`.

To retrieve information from the database, we use the SQL `SELECT` command:

```
SELECT PlanetName FROM Planets;
```

We can add a constraint to this query:

```
SELECT PlanetName FROM Planets WHERE DayLength > 200;
```

To create a table in an existing database, we use the `CREATE` query:

```
CREATE TABLE Planets (
       PlanetName        CHAR(8),
       DayLength        INT,
       YearLength        INT,
       PRIMARY KEY (PlanetName)
);
```

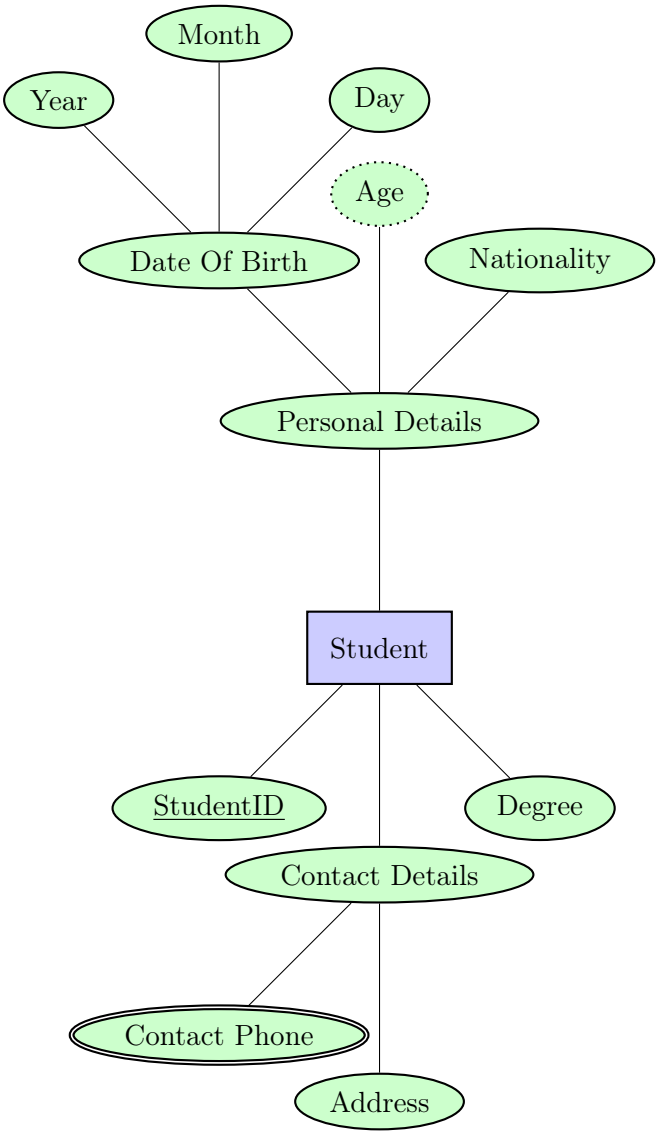


Figure 5: Attribute Types

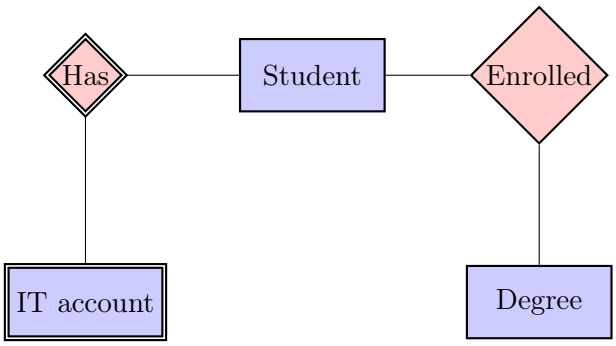


Figure 6: Relationships

# Week 4

Key Concepts

- Create and explore relational databases using SQL
- Design a database using Entity/Relationship diagrams
- Explain core concepts of relations and relational theory

## 2.201 Introducing Joins

Joins are used to make queries that collect data from two tables. Figure 7 shows a diagram for the tables we will use to illustrate how to use Joins.

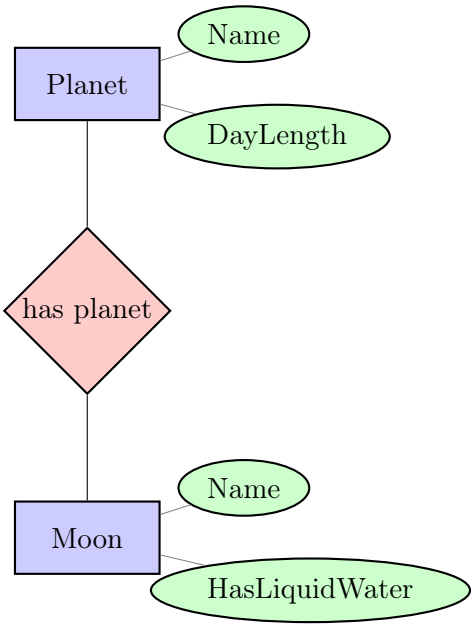


Figure 7: ER Diagram: Planets and Moons

The simplest form of a `JOIN`, called a *Cross Join* or a *Cartesian Join*, consists of simply listing all the tables we want to collect data from:

```
SELECT Lead.name, Rhythm.name,
       Bass.name, Drums.name
FROM   Lead, Rhythm, Bass, Drums;
```

The total number of results is the product of all entries in all tables, i.e. it essentially carries out a Cartesian Product of all the sets (tables) involved. E.g. if we have 3 Lead guitarists, 2 Rhythm guitarists, 5 Bass guitarists, and 7 Drum players, the total number of results will be $3 \cdot 2 \cdot 5 \cdot 7 = 210$ rows of results. Because the number of results grows so fast, we should carefuly consider our constraints in the `WHERE` clause to limit the results.

The example below is another way of executing a `JOIN`, called an *Inner Join*.

```
SELECT Planet.Name, Moon.Name, HasLiquidWater
FROM   Planet, Moon
WHERE  Planet.Name=Moon.HasPlanet
AND    DayLength < 11;
```

A more explicit version of the *Inner Join* is shown below

```
SELECT Planet.Name, Moon.Name, HasLiquidWater
FROM   Planet INNER JOIN Moon
ON     Planet.Name=Moon.HasPlanet
WHERE  DayLength < 11;
```

The *Outer Join* is another type of `JOIN`. Its syntax is shown below.

```
SELECT Planet.Name, Moon.Name, HasLiquidWater
FROM   Planet LEFT JOIN Moon
ON     Planet.Name=Moon.HasPlanet;
```

Figures 8, 9, and 10, give a visual representation of some what results will be returned for the joins.

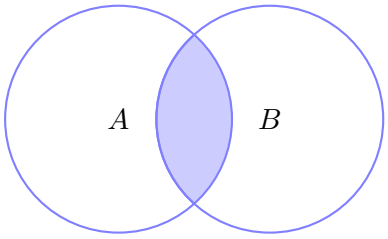


Figure 8: *SELECT * from A JOIN B ON A.B_id = B.id*

## 2.203 Drawing a database II: More about joins

Cardinality tells us how many rows in each of the tables participating in the join matches with how many rows on each of the table. It's often expressed in terms of a ratio, some of which are shown below:

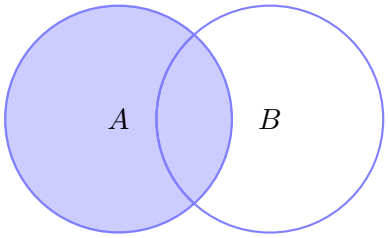


Figure 9: *SELECT * from A LEFT JOIN B ON A.B_id = B.id*

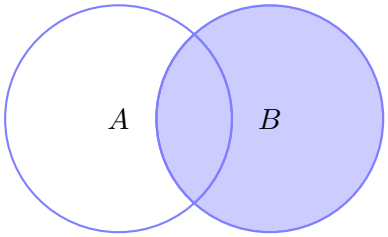


Figure 10: *SELECT * from A RIGHT JOIN B ON A.B_id = B.id*

**1:n** one row in table $x$ joins with **zero, one or more** rows in table $y$.

In this case we want to use a Foreign Key by placing the primary key of the table $x$ into the table $y$. Figure 11 shows a depiction of this case in an ER Diagram. If we want to show that at least one moon will be available, we use a double edge to connect the entity to the relation, as shown in figure 12.

In this situation, the Moon table would be declared like so:

```
CREATE TABLE Moons (
        MoonName   CHAR(20)
        PlanetName CHAR(10),
        Diameter   INT,

        PRIMARY KEY (MoonName),
        FOREIGN KEY (PlanetName)
        REFERENCES  Planets(PlanetName)
);
```

**1:1** one row in $x$ joins with exact one row in table $y$

This can be implemented as the case above, but we should consider implementing it as a single entity with attributes. Figure 13 shows an ER Diagram for this case.

The Project table would be declared like so:

```
CREATE TABLE Projects (
        Student VARCHAR(100)
        Title   VARCHAR(100),
```

```
4         Mark    INT,
5
6         PRIMARY KEY (Student),
7   );
```

**m:n** any number of rows from table $x$ joins with any number of rows in table $y$

This is impossible to implement with the Relational Model. We must add a new entity/relation. A depiction of the ER diagram is shown in figure 14. The fix for this case is shown in figure 15, it consists of adding a *Link Table* to the model.

The link table is created as shown below:

```
1   CREATE TABLE TutorRole (
2         Student VARCHAR(100)
3         Tutor   VARCHAR(100),
4
5         PRIMARY KEY (Student, Tutor),
6   );
```

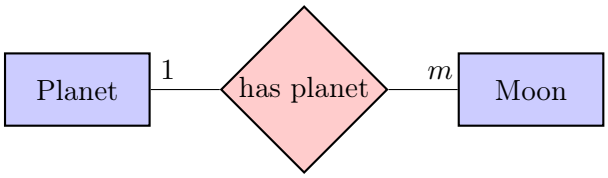


Figure 11: ER Diagram: Planets and Moons Cardinality $1 : n$

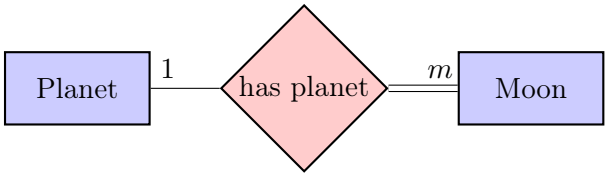


Figure 12: ER Diagram: Planets and Moons Cardinality $1 : n$, $n$ at least 1

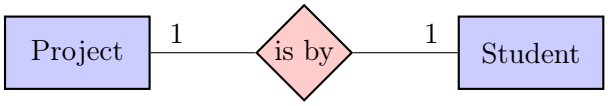


Figure 13: ER Diagram: Students and Projects Cardinality $1 : 1$

## 2.204 E/R diagrams summary


- Lewis, D. CO2209 Database systems. (London: University of London, 2016).


Figure 14: ER Diagram: Students and Tutors Cardinality $m : n$

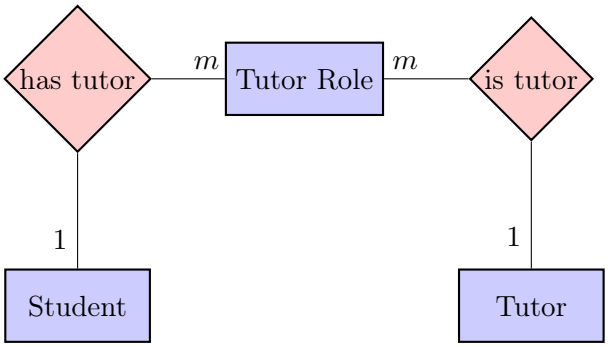


Figure 15: ER Diagram: Students and Tutors Cardinality $m : n$ with fix

## 2.301 Database integrity and the role of keys

By analyzing what could go wrong, we can design a database system that guarantees some error patterns won't happen. To motivate the discussion, we look at our planets and moons again. Assume we have entry shown below:

```
MoonName: Deimos
PlanetName: Mars
Area: 495
```

*PlanetName* could be mistyped as *Mers* or set to *NULL*. This planet doesn't exist, so our queries will produce erroneous results. *Area* should never negative, so we should disallow negative values. Some of these problems that can arise are detectable and preventable if we design the database for that.

## 2.303 Speaking to Databases II: SQL for joins and keys

Integrity Constraints can help us solve a few of the errors proposed before. Below we can find a list of common errors and their solution with integrity constraints.

**Join fields must match** We should use a `FOREIGN KEY`. A subsequent `INSERT` with wrong value will fail.

**One some values of a field are valid** Use `CHECK` column constraint.

**Tables values should not be inconsistent** Avoid repeating information in a database. `PRIMARY KEY` guarantees uniqueness. Avoid storing calculated values.

**Changes should not cause inconsistency** Use `FOREIGN KEY` rules to enforce correct behavior (i.e. `ON DELETE CASCADE`).

**Table values should not be inconsistent** Remove **functional dependencies**.

## 2.402 Further reading

- Chen, P. 'The Entity-Relationship Model – Toward a Unified View of Data', ACM Transactions on Database Systems 1(1) 1976, pp.9–36.

# Week 5

Key Concepts

- Control database access with appropriate security policies
- Explain other risks for data reliability and their management
- Describe the risks of repeated data in databases and design using normalisation as a tool to reduce those risks

## 3.001 Introduction to data Integrity and security

Sources of Error:

**Bad input data** Automated integrity checks greatly improve the situation

**Poor application logic** Can be mitigated with normalization

**Failed database operations** Usually cause the biggest problems. Easy to handle for atomic operations; very hard otherwise. Database snapshots and transactional database helps

**(Malicious) User activity** Control of user privileges help

## 3.103 Normalisation and the normal forms I

"*Non-loss decomposition* is the decomposition of a single relation into two or more relations such that a join on the separate relations reconstructs the original."

"*Functional dependency* states that an attribute $Y$ is functionally dependent on attribute $X$ if for any legal value of $X$ there is exactly one associated value of $Y$".

## 3.104 Normalisation and the normal forms II

Given a dataset, we can follow a progressively tighter list of constraints to ensure the data is sound while importing. That list of constraints, or requirements, is referred to as Normal Forms.

Before looking into the Normal Forms proper, we need a few extra concepts:

**Heath's Theorem** states that a relation with attributes $A$, $B$, and $C$ with a functional dependency $A \to B$ is equal to the join of $\{A, B\}$ and $\{A, C\}$. In other words if $A \to B$, then $B$ can be moved to a separate look-up table.

**Transitive Dependency** if $A \to B$, and $B \to C$, then $A \to C$.

**Multi-value dependency** $A$ and $B$ are two disjoint sets of attributes in a relation. There is a multi-value dependency if the set of values for $B$ depend only on the values of $A$.

The Normal Forms are:

**First Normal Form (1NF)** The table is a relation. All of its attributes are scalar values.

**Second Normal Form (2NF)** The table is in 1NF. Every non-key attribute is **irreducibly** dependent on the primary key.

**Third Normal Form (3NF)** The table is in 2NF. Every non-key attribute is non-transitively dependent on the primary key.

**Boyce-Codd Normal Form (BCNF)** Table is in 3NF. All non-trivial functional dependencies depend on a super key.

**Fourth Normal Form (4NF)** Table is in 3NF. For every muti-value dependency $A \twoheadrightarrow B$, $A$ is a candidate key.

## 3.105 On the normal forms


- Lewis, D. CO2209 Database systems. (London: University of London, 2016).
- Codd, E. 'A relational model of data for large shared data banks', Comms of the ACM 13/6 1970, pp.377–87.

# Week 6

Key Concepts

- Control database access with appropriate security policies
- Explain other risks for data reliability and their management
- Describe the risks of repeated data in databases and design using normalisation as a tool to reduce those risks

## 3.201 On ACID: Guaranteeing a DBMS against errors

To motivate the discussion, we create a scenario of a banking application handling money transfers. The stages for a transfer of £100 from A to B might be similar to the one illustrated in figure 16 below.

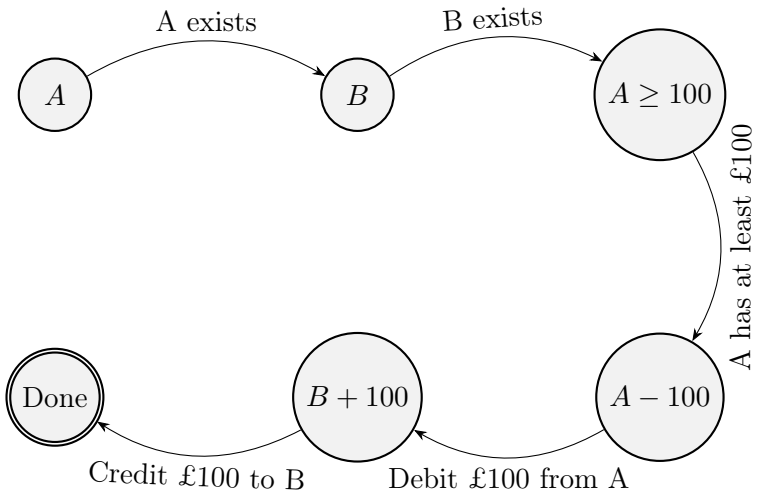


Figure 16: Transferring £100 from A to B

It should be clear that any one of these states can fail, at a minimum we can lose power mid-transfer. The issue here is the entire set of state transition only makes sense as a full block. If anything fails, we need to roll everything back, somehow.

Another bigger problem would be if A has two outstanding transactions happening at the same time. What happens if, e.g., A has exactly £900 in the account and schedules two simultaneous transactions, one for B of £100 and another for C of £900?

The formalization for the resolution of this class of problems is given by the *ACID properties*.

*ACID* is an acronym for Atomicity, Consistency, Isolation, Durability, which is a set of properties of database transactions intended to guarantee validity of data despite any possible errors that may arise, including power outages.

A group of databases operations that, together, satisfy the ACID properties is referred to as a *transaction*.

Further describing the purpose of each of the 4 tenets of ACID we have:

**Atomicity** guarantees a group of operations is treated as a single unit. This means that if any operation in the transaction fails, the entire transaction is considered to fail and the database is left unchanged.

**Consistency** guarantees that a transaction can only migrate a database from one **valid** state to another **valid** state, maintaining any invariants.

**Isolation** guarantees that concurrent transactions leave the database in the same state as if they were executed sequentially.

**Durability** guarantees that once a transaction is committed it will remain committed even in the evren of system failure (i.e. transactions are recorded in non-volatile memory).

## 3.203 Transactions and serialisation

In figure 17 we should a 3-stage transaction.

The database is only valid on *green* states, that is, either before starting operation 1 of after completing operation 3. The other two intermediary states must be considered invalid states and the database system should protect against those in the event of system failure.

Starting with *Isolation*, we can restrict access to data that might be affected by any operation in the block. In practice, the database system implements a lock that must be held in order to modify that particular set of data. This lock guarantees that no concurrent access to the data happens, thus forcing serialization.

*Atomicity* requires the implementation of rollback, as shown in figure 18. This ensures that in case an operation in a block fails, we must be able to rollback to the state immediately before the block was started. In other words, we guarantee that the database returns to a *green* state in the event of a failed operation.

Durability means that valid states are reliably recorded. The obvious way of achieving that is to write both states to persistent (i.e. non-volatile) storage as shown in figure 19 below.

Consistency comes as a result of restricting access to intermediate states of the database. Only store initial and final states. In practice, we only give operations access to the database if we know they won't suffer side effects from this known inconsistent (intermediary) state of the database. The mechanism to do that is called a *Transaction*. Once

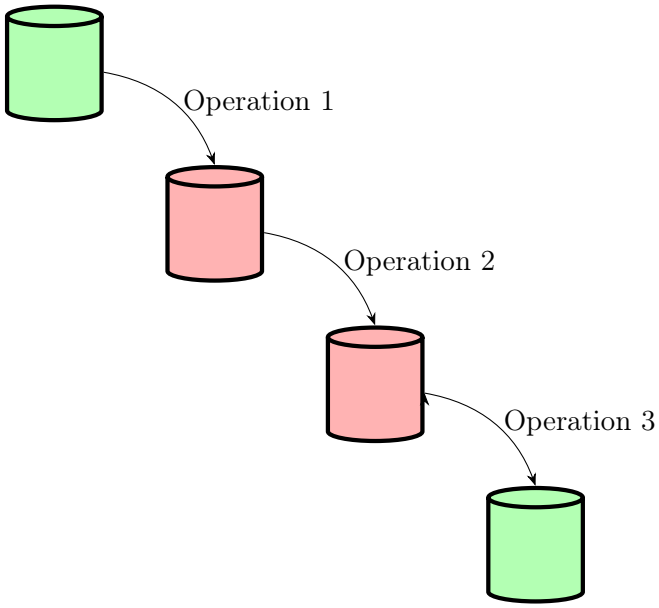


Figure 17: 3-stage Transaction

we decide that group of operations must be atomic, or carried out as a block, we declare them as a transaction by using `START TRANSACTION;` command. With that, any operations placed next until a call to `COMMIT;` will be treated as intermediate states of the database. In case of any errors, we can use the `ROLLBACK;` command to undo the inconsistent states.

Some details to keep in mind:

- Data Definition Language can cause problems
- Checkpoints may not be as frequent as transactions
- Table locking is not absolute

*Inconsistent Analysis* is when two transactions access the same data. One has multiple queries which give inconsistent information.

## 3.204 More depth on ACID and integrity risks

- Lewis, D. CO2209 Database systems. (London: University of London, 2016).

## 3.301 Malice and accidental damage

If we make our structures and logic explicit, it's far easier for human or system errors to be handled appropriately or avoided. Some actions can still cause trouble to the system, examples of which are discussed below:

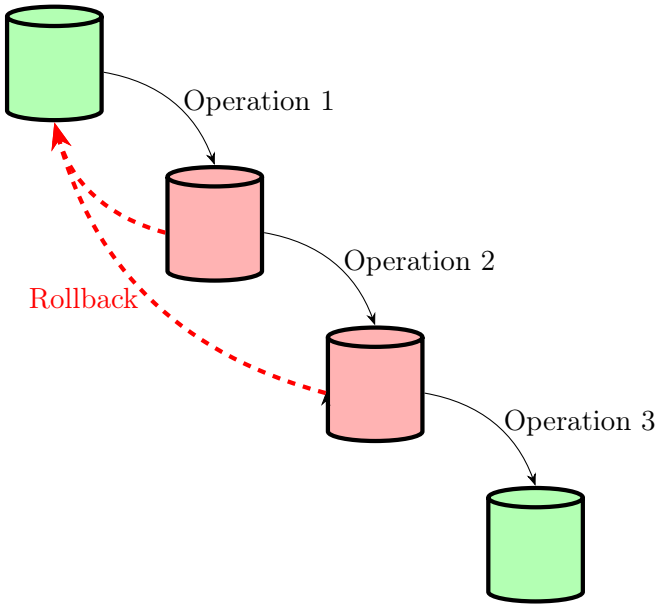


Figure 18: 3-stage Transaction With Rollback

**SQL Injection** adding malicious code into normal operations

**Privilege Escalation** malicious agent gaining direct access to the database

**User Error** use intends to do one thing but does something entirely different

**Non-confidential data sharing** confidential data being shared inappropriately

## 3.303 Security and user policies with SQL

Users in SQL have fine grained control of privileges:

- Create, edit users
- Create, edit, use databases
- Create, edit, use tables
- Create, edit, use data

A user policy must be defined in advance if we want to avoid common pitfalls with regards to access control. A consideration should be given to whether a particular user needs separate *roles*.

The format for controlling access permissions is very simple:

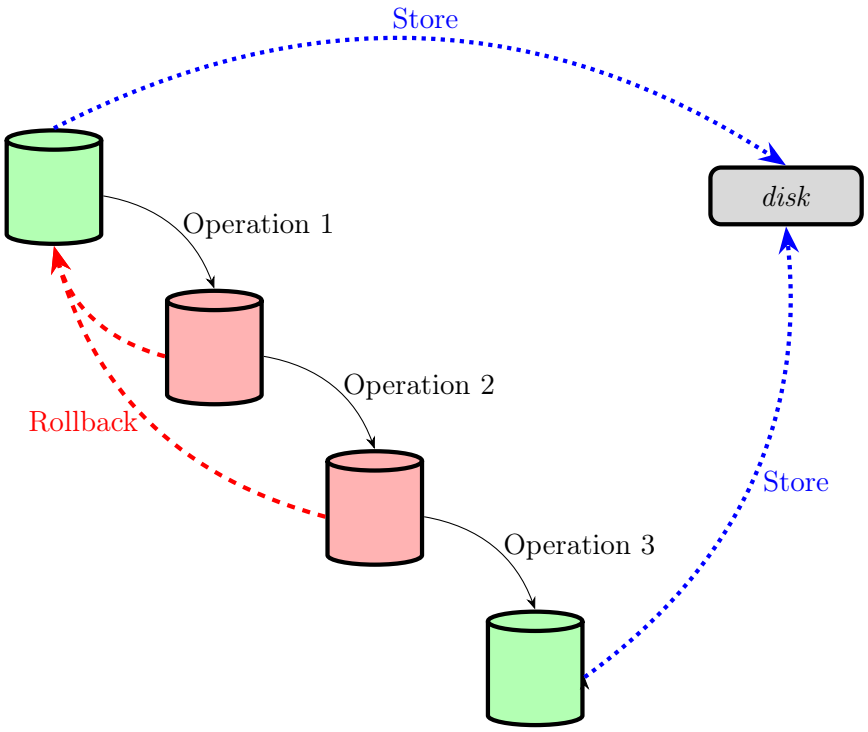


Figure 19: 3-stage Transaction Obvious Durability

```
GRANT privilege
ON    resource
TO    username;
```

For example, if we want user `JonDoe` to be able to `SELECT` data from tables `Planets` and `Moons`, we would write:

```
GRANT SELECT
ON    Planets, Moons
TO    JonDoe;
```

If `JonDoe` also requires permissions to `INSERT` data into these tables, the previous can be amended to:

```
GRANT SELECT, INSERT
ON    Planets, Moons
TO    JonDoe;
```

A `DROP`[1] permission would allow `JonDoe` to delete those tables:

[1] `DROP` permission should be granted with care

```
GRANT SELECT, INSERT, DROP
ON    Planets, Moons
TO    JonDoe;
```

With the addition of `WITH GRANT OPTION`, we let `JonDoe` grant the same privileges to other users:

```
GRANT SELECT
ON    Planets, Moons
TO    JonDoe
WITH GRANT OPTION;
```

Privileges can be revoked with the `REVOKE` command:

```
REVOKE ALL
ON    Planets, Moons
FROM  JonDoe;
```

Roles can be created to streamline user access control. Instead of granting each permission to every relevant table, we can create a role and just assign users to that role. This means that any user assigned to that role will have the exact same set of permissions.

For example, if we're building a system for Astronomers, we may create a role `Astronomer`:

```
CREATE ROLE Astronomer;
```

And assign permissions to that role:

```
GRANT INSERT, SELECT
ON    Planets, Moons
TO    Astronomer;
```

Whenever a new Astronomer joins the team, we assign the username the `Astronomer` role:

```
GRANT Astronomer
TO    JonDoe;
```

As a final thought, the goal of all this is grant users the **minimum** set of privileges required to carry out the intended use of the system, thus reducing impact of error or malice.

## 3.402 Further reading

- Date, C.J. Database Design and Relational Theory. (Healdsburg, CA: Apress, 2019) Chapter 4. FDs and BCNF (informal)

# Week 7

Key Concepts

- Use database interactions in a data analysis context
- Use database queries in node and PHP web applications
- Connect to an SQL-based database from a range of clients

## 4.004 Getting practice with MySQL (Lab introduction)

SQL has a few Aggregate Functions which are very useful for generating summary of data. The general structure is:

```
1 SELECT   BoughtFor,
2          SUM(Price)
3 FROM     Shopping
4 GROUP BY BoughtFor;
```

Some of the available functions are:

**SUM** Computes a regular sum of the group

**AVG** Computes the average of the group

**STD** Computes the standard deviation of the group

**VARIANCE** Computes the variance of the group

**MAX** Produces the maximum values of the group

**MIN** Produces the minimum value of the group

**COUNT** Produces a count of the number of things we have aggregated in our group

**COUNT(DISTINCT)** Produces a count of the distinct items in the group

**GROUP_CONCAT** Valid for string data, concatenates the entire group into a single string

## 4.007 Connecting to an SQL RDBMS

Database libraries help us create a persistent connection to the database and exchange queries and results.

In most cases, to connect to database we will follow one of the two paradigms shown below:

```
conn = newConnection(host, username, password, database);
conn.connect();
```

or

```
conn = connect(host, username, password, database);
```

To send queries, the common paradigms are as follows:

```
resource = conn.execute(query);
resource.fetchData();
```

or

```
result = conn.query(query);
```

In general, a response is an iterable container.

# Week 8

Key Concepts

- Use database interactions in a data analysis context
- Use database queries in node and PHP web applications
- Connect to an SQL-based database from a range of clients

## 4.201 Using libraries to update data in a database

Any query to the database works the same as any other. For example, an `INSERT` can be carried out as shown below.

```
const addActor = `INSERT INTO Actors
Values ("${actor.name}, ${actor.gender},
         ${actor.birthdate}");`;

connect.query(addActor);
```

Any input must be sanitized to avoid SQL Injection attacks.

## 4.402 Further reading

- nodejs
- PHP
- Python

# Week 9

Key Concepts

- Evaluate and explain the strengths and limitations of Normalisation
- Analyse a database and assess strategies for optimisation for speed or reliability

## 5.101 Query efficiency

The most expensive operations in a database are:

**Searching** often involve checking values on every entry in a table

**Sorting** ordering data by a given column, ascending or descending

**Copying** reading and writing a subset of the data points

When our queries start to get slow, these are the things to look out for; i.e. these are the places where we're likely to find optimization opportunities.

For example, when searching, if we know our data is already sorted, we can use Binary Search for finding what the row we're looking for.

Using sorted tables like mentioned above has the benefits of being as fast as tree indexes and requiring no extra space, however we can only choose one column to be the primary key.

Another option is use indexes, which usually is implemented as a B-tree or as a Hash table. The index can also be much smaller than the table itself, which may let us keep it in memory, rather than in disk.

B-trees generalize the concept of Binary Search Trees to nodes with more than two children, as such, it maintains all BST properties of space and time complexity. These properties are summarized below:

| **Complexity** | **Average** | **Worst** |
|---|---|---|
| *Space* | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| *Search* | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |
| *Insert* | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |
| *Delete* | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |

B-trees also support range searching and approximate searching. In comparison, Hash tables are very fast, as summarized below.

| **Complexity** | **Average** | **Worst** |
|---|---|---|
| *Space* | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| *Search* | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ |
| *Insert* | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ |
| *Delete* | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ |

Hashing algorithms can be algorithms and hash tables can't support range searching or approximate searching.

## 5.103 Removing the safety net: denormalisation

Normalisation can reduce disk reads by only reading the portion of data that is necessary for our application. It can also reduce integrity checks and reduces storage requirements. However, it increases the use of joins which can be expensive.

One approach to mitigate the problem of joins is to cache some joined selects in memory. That way, future joined selects can be immediately returned from the cache, rather than hitting the backing store again. While this can give us some performance benefits, it's risky for highly dynamic data.

An alternative approach is to employ *Views*. A View will act as a virtual table from which we can request data using regular queries.

# Week 11

Key Concepts

- Explain the principles of distributed databases
- Describe the Map/Reduce algorithm
- Discuss relative merits of local, relational databases and distributed databases

## 6.001 Introduction to distributed databases and alternative database models

With the rise of Web and Big Data, grew the need for distributed databases. In a distributed database system, the user should not be aware that a distributed database is being used. It should behave the same way as if data was held all in one place.

There are a few reasons why distributing the database may become an interesting proposition. Some of which are shown below.

**Parallelization** By distributing the data, we can enable parallelization of our operations without locks getting in the way.

**No single point of failure** If one instance of the database fails, we can replace it with another system on the fly with minimal to no user impact.

**Dividing large dataset** In case of a large dataset, we may not want to process it all in one place.

## 6.101 Approaches to distributing RDBMS

Requirements for distributing databases.

**Local autonomy** Sites should operate independently, i.e. one site should not be able to interfere with another's operations. Moreover, no site should rely on another for its own operation.

**No centralization** No single site should be able to control transactions or operations. No single site failure should break the system.

**Continous operation** The system is available most of the time and reliable.

**Location independence** The user should not know where the data might be located. Data can be moved from one location to another without changing functionality.

**Partition independence** The user doesn't need to know how the data is partitioned and the data can be partitioned without changing functionality.

**Replication independence** Distributed databases often required duplicate copies of data. The user need not be aware that replication is used.

**Distributed queries** Execute queries close to the data.

**DBMS independence** In theory, we should be able to distribute data over different DBMS systems.

**Others** other important requirements

- Hardware independence
- OS independence
- Network independence

A few important concepts are summarized below:

**Partitioning** How will the data be divided? Sometimes called *fragmentation*.

**Vertical partitioning** Dividing table by columns

**Horizontal partitioning** Dividing table by rows

**Catalogue management** How is information about the data distributed?

**Recovery control** Transactions usually use *two-phase protocol*. This requires on site to act as a **coordinator** in any given transaction.

**Brewer's Conjecture** Three goals (CAP) in tension.

**Consistency** All parts of the database should converge on a consistent state

**Availability** Every request should result in a response eventually

**Partition tolerance** If a network flaw breaks the network into separate subnets, the database should run and recover

## 6.103 Distributed database tradeoffs: gains and losses

- Brewer, E. 'CAP Twelve Years Later: How the "Rules" have Changed', Computer 45(2) 2012, pp.23-29.

## 6.201 Key/Value databases and MapReduce

*If distributing databases is complex, simplify your data structures.*

When using key-value databases, we only have two columns. One column for the key and one for the value associated with that key. By doing this, we do away with the concept of integrity and foreign keys, which greatly simplifies the process of distributing the database.

Key-value databases imply the following:

1. Easy parallel processing
2. Easy partition
3. Partition is always horizontal
4. Processing must happen near the data where possible

MapReduce is an example of an algorithm for processing key-value datasets. It consists of a *map* procedure, which performs filtering and sorting, and a *reduce* procedure, which performation a summary operation.

The *map* phase is carried out with direct access to the data. Usually, it loops over all the data. The output of this phase is a new key-value set.

The *reduce* phase is carried out by reducer workers. They summarize the data based on a key. We can assign reducer workers based on key, which enables highly parallel processing.

MapReduce is a very simple algorithm which enables easy data distribution and parallel processing. It can also recover from failure of all bu the coordinating node easily.

## 6.202 Jeffrey Dean and Sanjay Ghemawat introducing MapReduce

- Dean, J. and S. Ghemawat 'MapReduce: a flexible data processing tool', Communications of the ACM 53 (1) 2010.

# Week 12

Key Concepts

- Describe the principles of document databases
- Read, write and manipulate MongoDB instance
- Explain the principles of distributed databases

## 6.301 Document databases and MongoDB

Key-value databases have a specific application. While the model doesn't work for all types of applications, they enable easier distribution of data and processing.

There is a need for a middle-ground between Relational Databases and Key-value pair Databases. That middle-ground is covered by Document Databases.

Document Structures:

- are less strict
- can be nested
- can be repeated
- can be order-sensitive

There is an expectation that documents have less interlinking among them, or at least that interlinking is less important for that data retrieval.

Some of the document formats in existence:

- Markup languages for text
- Markup languages for other data
- Bespoke formats
- JSON

The JavaScript Objection Notation (JSON) language is useful for data persistence and data exchange. It's very similar to JavaScript itself with a few caveats.

MongoDB is one implementation of a Document Database. It's also capable of distributing the by means of *sharding* – i.e. taking horizontal partitions of the data –, and it's open source(*-ish*).

# Week 13

Key Concepts

- Read and understand simple XML schemata
- Read, write and manipulate XML data
- Explain what is distinctive about a semantic database

## 7.101 Semantic databases: What does a table actually tell us?

Sharing data can be an involved process. We have to deal with different formats and ensuring the data is imported properly. For example, if someone shares a CSV file, we don't have enough information of the relations in the database. Another option would be share a database dump as produced by MySQL, that will contain all the necessary SQL statements to reconstruct the entire database, including relations and constraints, however this is still limited.

If we want to get more *meaning* out of our data, we need a better format that lets us encode more meaning.

For example, if we're dealing with a Movies database where each movie has 3 actors, we may have columns `Actor1`, `Actor2`, and `Actor3`. However, we can encode more meaning here. We know actors are input in the system by their names and we know the name refers to an actor.

Perhaps the table also has the `Year` column encoding the year of release. We can assume `year` to be a positive integer within a certain range. We can also assume that it's a Calendar Year, i.e. the year number is a valid number within some Calendar structure.

In other words, we have layers of meaning:

- Data Type
  - String
  - Integer
  - Float
- Data Domain
  - Place
  - Person

  - Date

- Data Semantics
  - Person acted in film

If we have a database encoding semantics, there is the potential for the system itself to make use of logic and implement automated reasoning[1] for data retrieval.

## 7.103 Shared meaning in the real world

Common Semantics

**Shared documents** we can share documents that don't carry semantics and communicate the semantics separately. Ideally, we'd want the document itself to carry its semantics, but that's not essential

**Formal Specifications** say how documents are made and gives structure to them

**Human-readable Definitions** regardless of how much machine-readable semantics is encoded, we still need to specify semantics for humans

## 7.105 XML: Documents with semantics

The *eXtensible Markup Language*, or XML, is a language that defines rules to encode documents that are both human-readable and machine-readable.

[1] Section 4.4 of SICP discusses this very idea. See `https://sarabander.github.io/sicp/html/4_002e4.xhtml#g_t4_002e4`

# Week 14

Key Concepts

- Read and understand simple XML schemata
- Read, write and manipulate XML data
- Explain what is distinctive about a semantic database

## 7.204 Transforming XML: XML Pipelines

Transforming XML is the equivalent of a query. We can transform XML into another XML or into any other textual format. There are two main languages to perform these transformations:

- XSLT
- XQuery

The eXtensible Stylesheet Language Transformations, XSLT, encodes how we want to transform the XML source. In order to use it, we need the input XML, the XSLT transformations, and a XSLT Processor, the output of which is the new transformed text. Something similar to figure 20 below.

XQuery works in a way that's closer to how SQL works. We provide a template with interspersed XQuery code. In XQuery terms, we talk about FLWOR (pronounced as *flower*) where:

**F** `for` clause

**L** `let` clause

**W** `where` clause

**O** `order by` clause

**R** `return` clause

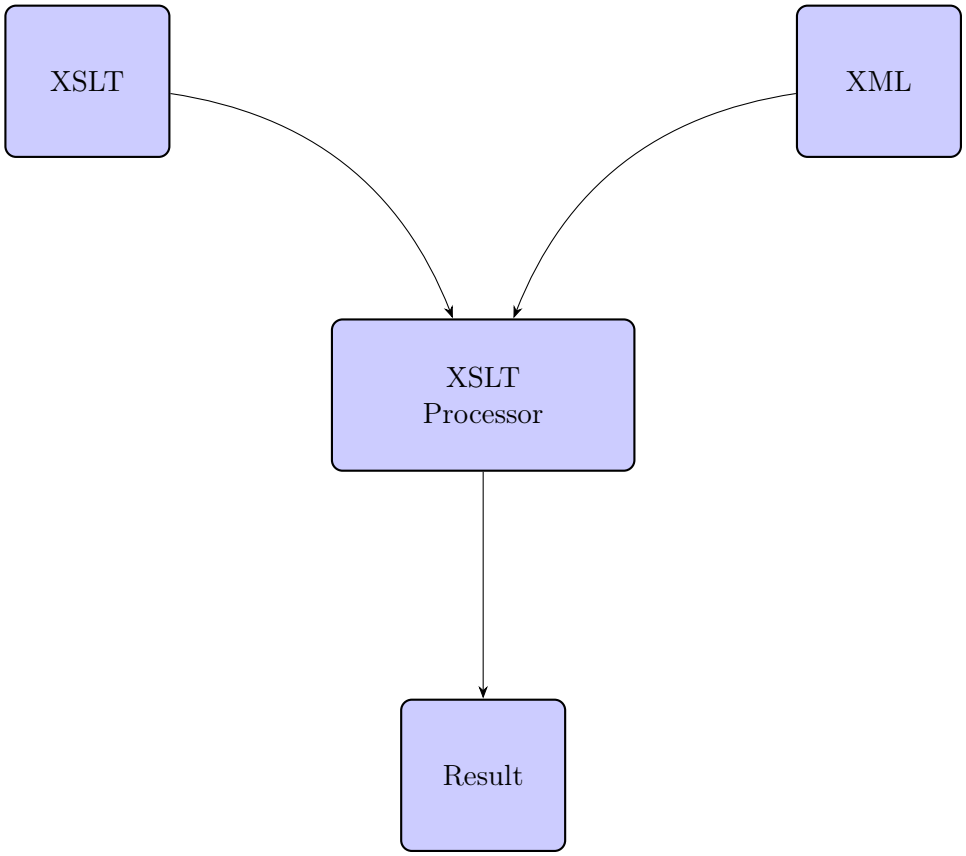


Figure 20: XSLT Processor

## 7.301 XML Schemata: Syntax and semantics for XML

Schema Languages specify what can go in an XML file. There are a few schema languages that can be used, the main ones are:

**Document Type Definition (DTD)** the first schema language, inherited from SGML. There are known limitations which the other schema languages tried to solve

**XML Schema Definition (XSD)** recommended by the World Wide Web Consortium (W3C) for formally describint the elements in an XML document

**RELAX NG** The REgular LAnguage for XML Next Generation specifies a pattern for the structure and content of an XML document. RELAX NG is itself an XML document. It was defined by a committee specification of the OASIS RELAX NG Technical Committee.

**Schematron** it's a structural schema language written in XML using a small number of elements and XPath. In general, a Schematron schema is processed into a normal XSLT and consumed as such

In general, what every schema language has in common is that they all define which **elements** are used, what **can** they contain, what data is passed, what order, what **attributes** are used, waht structures are **equivalent**, and so on. In summary, they all formally specify the structure of an XML document.

This can used to validate an XML document, enforce integrity, and support debugging.

# Week 15

Key Concepts

- Design simple web ontologies
- Query semantic databases with SPARQL
- Design simple linked data systems

## 8.002 Open, Linked and Data

Openness

- Cost-free access
- Barrier-free access
- Restriction-free use

*FAIR* data

- Findable
- Accessible
- Interoperable
- Reusable

Web links

- Links are one-way
  - No permission needed
  - No central registry of links
- URL
  - Garanteed unique
  - Reponsibility of maintenance
  - Unlimited number of URLs
  - Unique ID independent of server

## 8.004 Tim Berners-Lee's Proposal – inventing the web

- `https://www.w3.org/History/1989/proposal.html`

## 8.101 RDF: the model and its serialisations

Web technologies are built around the Document Object Model (DOM). Linked data also has a model, called Remote Description Framework (RDF). Figure 21 shows a depiction of the RDF.

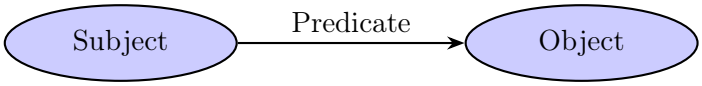


Figure 21: Remote Description Framework

For example, we can state that *Deimos orbits Mars* with the diagram shown in figure 22.

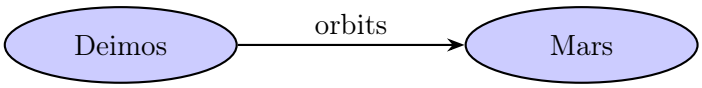


Figure 22: Deimos Orbits Mars

More relationships can be added, for example *Mars orbits The Sun*, *Mars is of type Planet*, *Deimos is of type Moon*, and so on. By combining these together, we can produce a much larger graph, as shown in figure 23.

The *type* is so common, that usually it's wrapped up in the diagram, as shown in figure 24.

There are challenges:

- Maintaining keys on the web
- Finding data on the web
- Sharing meaning
- Sharing entities

URLs are the key to solving these issues. Each node in our RDF graph represents a URL. The edges in the graph can be replaced with the matching Wikidata link. Similarly, each entity in the graph has an entry in Wikidata which means they can also be replaced with the matching wikidata link.

In summary, linking data requires that

- Subject and Predicate must be URIs

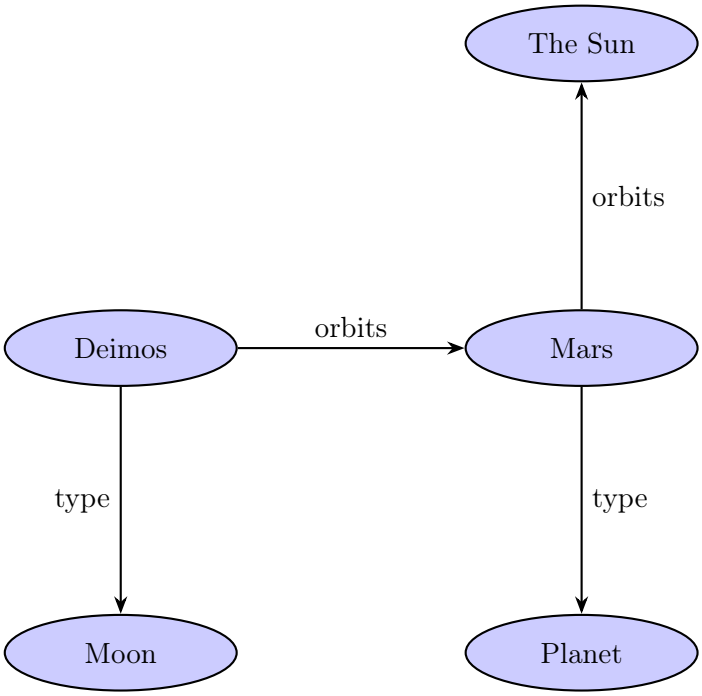


Figure 23: RDF Graph

- Object can be URI
  - Or string, number, date, etc
- sameAs predicates can connect URIs that represent the same thing

## 8.103 Thinking in graphs

- No hierarchy
- No Order
- Every circle-arrow-circle is a triple

## 8.201 Introduction to web ontologies

The way that semantics is shared in the web is through Web Ontologies. The simplest way to describe the syntax and semantics of our ontology is through RDF Schema.

- Simple structures
  - Syntax
  - Inheritance

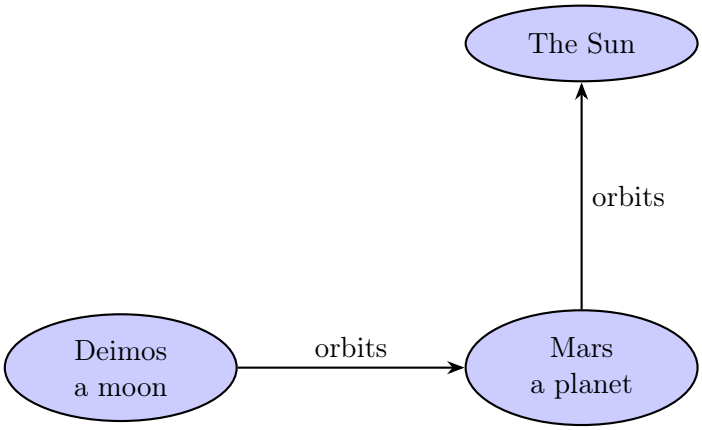


Figure 24: Type Combined

– Documentation

OWL – Web Ontology Language – allows us to encode the logic of the system.

**owl:ObjectProperty** Predicate connecting entities

**owl::DataProperty** Predicate connects entity to data

**owl::inverseOf** make entity be the inverse of another

**owl::sameAs** Connects data sources

## 8.205 Designing an ontology

Designing an ontology isn't much different from designing a SQL database. There is a need for extra care needed when working with Web Ontologies. If our ontology is good, other people may start using it and we can accidentally or deliberately create a standard. When using other people's ontologies, it's important to use it the way it was intended.

Whenever we're tasked with designing an Ontology, there are a few principles we should follow:

- Use existing ontologies where possible
- Combine effort with others
- Test with real data
- Don't get lost in rabbit holes, i.e. avoid adding unnecessary details
- Don't be wrong
- Designing good ontologies take time

- Multiple viewpoints are vital
- Drawing helps a lot
- Be as explicit as possible to draw out problems
- Try out protege for ontology specification
  - Webprotoge is a simpler version

# Week 16

Key Concepts

- Design simple web ontologies
- Query semantic databases with SPARQL
- Design simple linked data systems

## 8.301 Triplestores and SPARQL

The Semantic Web is hard to search efficiently, partly because there is no registry of information. Finding something requires a lot of indexing, which means we need crawlers.

A graph database can be built from a cache of triples. Triples can be index and inferences built from the triples can be cached as new triples.

*Triplestore* is one type of graph databases which use RDF to cache a chunk of the semantic web. We can search these databases using patterns which the search engine uses to look for complete or partial matches, returning them as the list of results. The language used to specify such patterns is *SPARQL*.

*SPARQL* stands for SPARQL Protocol And RDF Query Language. A sample query is shown below:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/>
SELECT ?friend
WHERE {
  ex:Alice foaf:knows ?friend.
}
```

The query above will produce a list of URLs of people who *Alice* knows. If we want to get the name of the person, rather the URL, we modify the query like so:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/>
SELECT ?fName
WHERE {
  ex:Alice foaf:knows ?friend.
  ?friend foaf:name ?fName.
}
```

If we want to produce a list of names who are connected to *Alice* by any number of connections, we use:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/>
SELECT ?fName
WHERE {
  ex:Alice foaf:knows+ ?friend.
  ?friend foaf:name ?fName.
}
```

To limit for unique names:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/>
SELECT DISTINCT ?fName
WHERE {
  ex:Alice foaf:knows+ ?friend.
  ?friend foaf:name ?fName.
}
```

SPARQL endpoints can be used programmatically. There are ready-made libraries in several languages. Much like SQL libraries, we just need to connect to the endpoint. YASGUI allow us to connect to any SPARQL endpoint and issue queries.

## 8.401 Deferencing URIs and following your nose

Dereferencing is not a necessary part of Linked Data, although it is a very useful one.

We know that a URI uniquely identifies a triple. When we want to find out more about that URI, we must *dereference* it, i.e. request the RDF document at the end of the URI. This document, references other URIs, which we can request the RDF document for those and so on, slowly *following our nose* and building a local knowledge dataset.