# CO130 - Databases

## Prelude

The content discussed here is part of CO130 - Databases (Computing MEng); taught by Thomas Heinis, in Imperial College London during the academic year 2018/19. The notes are written for my personal use, and have no guarantee of being correct (although I hope it is, for my own sake).

## Material Order

These notes are primarily based off the slides on CATe. This is the order in which they are uploaded (and I'd assume the order in which they are taught).

1. *Introduction.pdf*
2. *ER Modelling.pdf*
3. *ER to RM.pdf*
4. *Relational Algebra.pdf*
5. *Tutorial ER.pdf*
6. *Tutorial Translation.pdf*
7. *Relational Algebra.pdf*
8. *Functional Dependencies.pdf*
9. *Tutorial Relational Algebra.pdf*
10. *Normalisation.pdf*
11. *SQL.pdf*
12. *Data Definition.pdf*
13. *Data Manipulation.pdf*
14. *Advanced SQL.pdf*
15. *Functional Dependency Tutorial.pdf*
16. *Transactions.pdf*
17. *SQL Tutorial.pdf*
18. *Storage.pdf*
19. *Indexing.pdf*
20. *NoSQL.pdf*
21. *MapReduce.pdf*

## Introduction

We use databases as it's more organised; hence it's easier to model and manage. It's more efficient, as it's fast to search, and update, and integration allows us to minimise data duplication. Concurrent (and therefore multi-user) access allows multiple people to access the database at the same time (will require some techniques).

**Transactions** are sequences of database actions that execute in a coherent, and reliable way - the classical properties are **ACID**. Consider the two transactions `T1`: $A = A - 100; B = B + 100$, and `T2`: $B = B - 100; A = A + 100$, we can observe ACID properties as follows;

- **atomicity** if one part of a transaction fails, the entire transaction fails

  on completion of `T1`, either $A' = A - 100$, and $B' = B + 100$, or $A' = A$, and $B' = B$, where the former is a successful transaction, and the latter is in the case of a failure.

- **consistency** transactions don't leave the database in an inconsistent state

  the sum of the balances must remain the same, such that $A' + B' = A + B$; we can also have more constraints such as keeping balances positive, or limiting the amount a transfer can do at once

- **isolation** transactions run as if no other transactions are running (may need to wait)

  given the two concurrent transactions `T1`, and `T2`, one has to be completed before the other can start

- **durability** results of successful transactions aren't lost on system failure

  the new values, $A'$, and $B'$ must persist if the transaction completes, even if the system fails (disk failure etc.)

A **Database Management System (DMBS)** creates new databases via a **Data Definition Language (DDL)**, which specifies the structure (**schema**). It also queries, and manipulates through a **Data Manipulation Language (DML)**. Examples of this include *PostgreSQL*, *MySQL*, *SQLite*, and can also fall under *NoSQL*, however SQL remains as the most widespread technology (as of writing this).

It lets us define, query, and manipulate databases with a high-level declarative language (**Structured Query Language**). It's standardised by the ISO, but each DBMS implements its own variation of the standards, which may be costly if it's complex.

### Relational Model

Consider the following model, represented as a table;

| heading | `title:string` | `year:int` | `length:int` | `genre:string` |
|---|---|---|---|---|
| body | Gone with the Wind | 1939 | 231 | Drama |
| | Star Wars | 1977 | 124 | Science Fiction |
| | Wayne's World | 1992 | 95 | Comedy |

We have the columns be the attribute, with the top row being the heading, and the rest being the body. The attributes are in the format `name:type`. The rows (of the body) are referred to as tuples. A relation is the heading as well as the body (the entire table). The heading is an **unordered set** of attributes, and an attribute is the name as well as the type (typically indivisible types). The body is an unordered set of tuples, and a tuple is the set of attribute values. The schema is for the entire relation is the name of the relation, and the heading, in this case, we'd have; `movies(title:string, year:int, length:int, genre:string)`, and a database is a collection of relations. A schema for a database is the schemas for all relations.

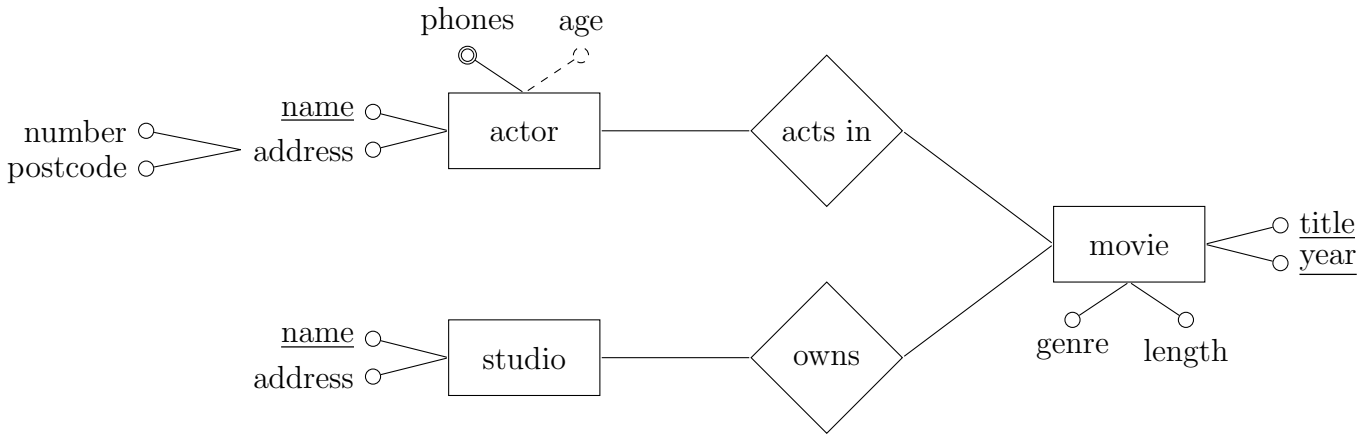
In mathematics, with a set of sets; $S_1, S_2, ..., S_n$, a relation $R$ is a set of tuples $T_1, T_2, ..., T_n$, where $T_k \in S_k$, therefore $R \subset S_1 \times S_2 \times ... \times S_n$. As the idea of relations stems from set theory, it's important to note that the order in which we represent the attributes, and tuples in unimportant. In the Relational model we have **attributed** tuples, rather than **ordered**; $R$ is the set of tuples $(A_1 : S_1 = T_1, ..., A_n : S_n = T_n)$, with $T_k \in S_k$. It's important to note that relations aren't 2-dimensional tables, even though it's more convenient to draw it on paper. We should instead consider them as a set of $n$-dimensional values, such that we have (`title:string`=StarWars, `year:int`=1997, `length:int`=127, `genre:string`=ScienceFiction), as a 4-dimensional movie value.

## Entity Relationship Modelling

When a new database is being developed, it's important to try and model the real-world situation, instead of trying to refine it into an implementation, such as a relational model. In Entity-Relationship modelling, we try to create a diagram which represents the information needed for the database (the Entity Relationship Diagram). As there is no universally accepting notation for ER diagrams, we will use the following notation;

| type | description | shape |
|---|---|---|
| entity sets | a set of distinguishable entries that share the same set of properties, can be physical (a room etc.), an event (flight, sale, etc.) - they normally correspond to nouns | rectangle |
| relationship sets | captures how **two or more** entity sets are related (e.g. owns, tutors), we can also have more than one relationship set between entity sets, and they can also have a relationship set on the same entity - they sometimes correspond to verbs | diamonds |
| attributes | properties of an entity; relationship sets can also have attributes, and primary keys are underlined | small circles |

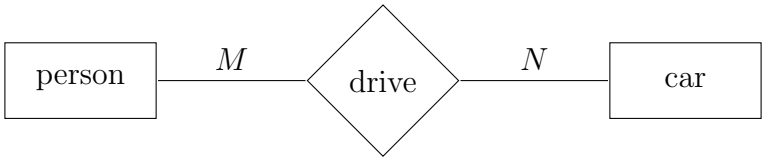
The movie example is represented below. Note that I've also extended it to contain different types of complex attributes. You can see that the address field is subdivided into number, and postcode, we have a multi-valued attribute in phones, and a derived attribute in age (can be calculated from date of birth)




### Cardinality Constraints

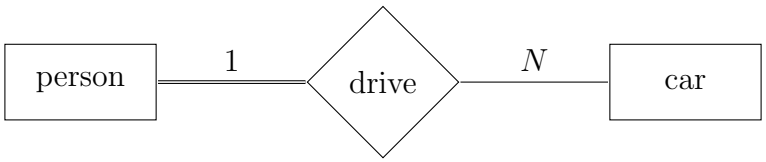
A relationship between two entity sets can be seen as one of the following (using the car example, where a person can drive $N$ cars, and a car can be driven by $M$ people);

- one-to-one $M = 1, N = 1$
- one-to-many $M = 1$
- many-to-many no restrictions

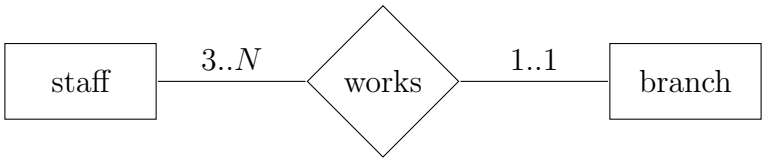



### Participation

Using the same example, we can force participation, which means that all entities must participate in a relationship, with a double line. The example below, with the cars, suggests that everyone drives at least 1 car, and car can have at most 1 driver (note that it doesn't mean that every car has a driver).



Sometimes, instead of using double lines, E-R notation may allow explicit bounds. The below diagram suggests that staff work in exactly 1 branch, and that each branch must have at least 3 members of staff;




### Fan, and Chasm Traps

See the diagrams in *ER Modelling.pdf* for examples.

We need to ensure that we do not allow ambiguous paths between entities. For example, if we say `staff` works for `faculty` which operates `department`, if some member of staff works for Engineering, which operates both Computing, and EE, we can't follow a path from the staff member to their department. This can be solved by having the staff work for the department, which is operated by some faculty.

If we can't follow a path between two entities, we may need to add another relation between them to specify the relationship.

### Weak Entities

Entities which cannot be uniquely identified by their own attributes are called **weak** entities, in contrast to **strong** entities which have primary keys. A weak entity has to be defined by a string entity. Weak entities are drawn as a double rectangle, and the relationship to the strong entity is drawn as a double diamond. The key for a weak entity is formed by combining the primary key of the strong entity with attributes of the weak entity (denoted by a dashed underline). For example, a room in a building is a **weak** entity, with a room number, and can be uniquely identified with the name of the building combined with the room number.

## ER to RM

Keys are used to join information when we do queries. The primary key is the unique identifier of a tuple, and a foreign key is the primary key of **another** table.

Once we have an Entity Relationship Model, we can then map it to a relational schema. The schemas can then be refined with functional dependencies, and implemented with relational languages. A high level data model isn't the only concern for database design (considered a simpler aspect).

If we consider a strong entity set, with simple attributes, we can easily create a table from it (see the example, `movie`, drawn above).

```
-- movie(title, year, length, genre)

CREATE TABLE movie {
  title VARCHAR(120),
  year INT,
  length INT,
  genre CHAR(20),

  PRIMARY KEY (title, year)
}
```

Composite attributes are also fairly easy to represent, since we just flatten it, therefore store each of the sub-attributes as their own field - consider the actor example (note that if the primary key is composite, we mark all the sub-attributes as primary keys). This example also shows how multi-valued attributes are stored - we create their own relation, and link back to the entity set with a foreign key constraint;

```
-- actor(name, number, postcode)
-- actor_phones(actorName, phoneID, other attributes)

CREATE TABLE actor {
  name VARCHAR(60),
  number INT,
  postcode VARCHAR(10),
  PRIMARY KEY (name)
}

CREATE TABLE actor_phones {
  actorName VARCHAR(60),
  phoneID VARCHAR(10),
  ...
  PRIMARY KEY (actorName, phoneID),
  FORIEGN KEY (actorName) REFERENCES actor.name
}
```

The relational model doesn't allow us to specify derived attributes, and we're therefore expected to calculate them with queries.

We can also consider how many-to-many relationships can be represented (using the car example again). The first example is when a car can be driven by many people, and a person can drive many cars;

```
-- person(ID, other attributes)
-- car(regno, other attributes)
-- drive(personID, regno, other attributes)

CREATE TABLE drive {
  personID VARCHAR(10),
  regno VARCHAR(12),
  ...
  PRIMARY KEY (personID, regno),
  FORIEGN KEY (personID) REFERENCES person.ID ON DELETE CASCADE,
  FORIEGN KEY (regno) REFERENCES car.regno ON DELETE CASCADE
}
```

However, it's easier to represent a one-to-many relationship, as we can simply include the primary key of the "one" relation as a foreign key in the many (where a car can be driven by 1 person, and a person can drive many cars). The representation for a one-to-one is similar, but the foreign key can be in either one, which is up to the designer;

```
-- person(ID, other attributes)
-- car(regno, personID, other attributes)

CREATE TABLE car {
    regno VARCHAR(12),
    personID VARCHAR(10),
    ...
    PRIMARY KEY (regno),
    FORIEGN KEY (personID) REFERENCES person.ID
}
```

In a weak entity, we include the primary key of the strong relation as a foreign key, but also use it as part of the composite primary key for the weak entity. A delete cascade is also used, and finally the we use `not null` to ensure total participation;

```
-- building(name, other attributes)
-- room(no, buildingname, other attributes)

CREATE TABLE room {
    no VARCHAR(120),
    buildingname VARCHAR(50) NOT NULL,
    ...
    PRIMARY KEY (no, buildingname),
    FOREIGN KEY (buildingname) REFERENCES building.name ON DELETE CASCADE
}
```

A multiway relationship can be mapped as several binary relationships, or we can generalise it to have the primary key consist of the relationship be composed of the primary keys of the many entity sets, and have the primary keys of the one entities be attributes. For roles, we map each role as a foreign key attribute in the entity set;

```
-- movie(ID, other attributes)
-- sequelof(originalID, sequelID, other attributes)

CREATE TABLE sequelof {
    originalID INT,
    sequelID int,
    ...
    PRIMARY KEY (orignalID, sequelID),
    FOREIGN KEY originalID REFERENCES movie.ID,
    FOREIGN KEY sequelID REFERENCES movie.ID
}
```

### Extended Models

Newer models extend the classic model by supporting features from **object-oriented design**. We can look at how E-R models handle specialisation, or generalisation with **is-a hierarchies**. For example, we can specify that a `cartoon` **is-a** `movie`, which means that it inherits all the attributes of movie, but can have more attributes. and relations. Once this extends on multiple levels, we begin to form a hierarchy.

## Relational Algebra

We can use **relational algebra** to construct new relations from existing ones, the operators include selection, projection, intersection, union, difference, and product. Consider the examples below, `a`, `b`, and `c` are all integers. Intersection, union, and difference will be omitted, since those are fairly self-explanatory.

| $R$ | | | | | | $\pi_{\texttt{a,c,e}}(R)$ | | | $S$ | | | $\sigma_{\texttt{a>3\|b<5}}(S)$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | f | a | c | e | a | b | c | a | b | c |
| 1 | 2 | 3 | 4 | 5 | 6 | 1 | 3 | 5 | 3 | 2 | 3 | 3 | 2 | 3 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 7 | 8 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 8 | 2 | 2 | | | |
| 1 | 2 | 3 | 4 | 5 | 8 | | | | 1 | 2 | 9 | | | |

The syntax in use here for projection is $\pi_{\texttt{attributes}}(T)$, and likewise for selection it's $\sigma_{\texttt{condition}}(T)$. The former returns a relation with only the listed attributes, and the only takes rows which satisfy the given condition. Neither will return duplicate rows. Here is an example of a Cartesian product;

| $R$ | | $S$ | | | $R \times S$ | | | | |
|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | a | b | c | d | e |
| 1 | 2 | 1 | 2 | 3 | 1 | 2 | 1 | 2 | 3 |
| 3 | 4 | 4 | 5 | 6 | 1 | 2 | 4 | 5 | 6 |
| | | 7 | 8 | 9 | 1 | 2 | 7 | 8 | 9 |
| | | | | | 3 | 4 | 1 | 2 | 3 |
| | | | | | 3 | 4 | 4 | 5 | 6 |
| | | | | | 3 | 4 | 7 | 8 | 9 |

We can take a natural join, where the resulting relation contains all the tuples that have matching attributes in $R$, and $S$. This is less common, and we'd typically use something closer to $R \bowtie_{R.\mathtt{b}=S.\mathtt{b}} S$. Note that in this case, they are the same, since that's the only attribute that overlaps;

| $R$ | | $S$ | | | $R \bowtie S$ | | | |
|---|---|---|---|---|---|---|---|---|
| a | b | b | c | d | a | b | c | d |
| 1 | 2 | 1 | 2 | 3 | 1 | 2 | 5 | 6 |
| 1 | 3 | 2 | 5 | 6 | 3 | 4 | 8 | 9 |
| 3 | 4 | 4 | 8 | 9 | 3 | 4 | 9 | 9 |
| | | 4 | 9 | 9 | | | | |

Attributes can also be renamed with the notation $\rho_{\mathtt{new/old},...}(R)$ notation, which is fairly self-explanatory.

## Functional Dependencies

Relational database schemas should be normalised, which helps reduce redundancy, and avoids update, and deletion anomalies. The basis for normalisation is the concept of **functional dependency**, and **keys**. Consider the following example, which can experience update anomalies;

| producer | city | product no. | price | quantity |
|---|---|---|---|---|
| 3 | London | 52 | 65 | 4 |
| 22 | Birmingham | 10 | 15 | 5 |
| 22 | Birmingham | 12 | 4 | 11 |
| 3 | London | 44 | 43 | 32 |
| 3 | London | 43 | 3 | 27 |

If an existing producer changes address, and tuples aren't updated, then it leads to incoherence (as the data would now be invalid). If a new tuple / row is inserted, and the producer already exists, but with an older address, we will once again have incoherence. If a producer doesn't have any open orders, say rows 1, and 5 didn't exist, then data about producer 3 would be lost. The price of the product doesn't depend on the producer, and the location of the producer doesn't have anything to do with the product. There is also redundant data, since the address is duplicated. Ideally, the data should be stored like this;

| Open Orders | | |
|---|---|---|
| producer no. | product no. | quantity |
| 3 | 52 | 4 |
| 22 | 10 | 5 |
| 22 | 12 | 11 |
| 3 | 44 | 32 |
| 3 | 43 | 27 |

| Producer | |
|---|---|
| producer no. | city |
| 3 | London |
| 22 | Birmingham |

| Product | |
|---|---|
| product no. | price |
| 52 | 65 |
| 10 | 15 |
| 12 | 4 |
| 44 | 43 |
| 43 | 3 |

A functional dependency is a constraint that if two tuples of a relation $R$ agree on a set of attributes $A_1, A_2, ..., A_n$, then they must also agree on the set of attributes $B_1, B_2, ..., B_m$. This can be written as $A_1, A_2, ..., A_n \rightarrow B_1, B_2, ..., B_m$, and therefore the $B$ set is functionally dependant on the $A$ set, or the $A$ set functionally determines the $B$ set. Therefore, for any set of values $A$, then there is only one set of values $B$. The normal forms a designer can use are defined in terms of their functional dependencies. This can be summarised by the following examples;

| functional dependency | for all tuple pairs, $x$, $y$, if | then assert |
|---|---|---|
| $A \to K$ | $x.A = y.A$ | $x.K = y.K$ |
| $A, B \to K$ | $x.A = y.A$<br>$x.B = y.B$ | $x.K = y.K$ |
| $A \to K, L$ | $x.A = y.A$ | $x.K = y.K$<br>$x.L = y.L$ |
| $A, B \to K, L$ | $x.A = y.A$<br>$x.B = y.B$ | $x.K = y.K$<br>$x.L = y.L$ |

Note that if we have two functional dependencies, $A, B \to K$, and $A, B \to L$, it's trivial to combine them as $A, B \to K, L$ (or to split them). If we have the functional dependency $A, B, C, D, E \to A, C, X, Y, Z$, we can remove all the attributes on the right hand side, that are on the left hand side, such that the aforementioned FD reduces to $A, B, C, D, E \to X, Y, Z$. A **trivial** FD is one where all the attributes on the RHS are on the LHS.

### Keys

If we have some set of attributes $A_1, A_2, ..., A_n$, functionally determine all the remaining attributes of the relation, then the set $A$ is a **superkey**. This also implies that two tuples cannot have the same superkey values, due to the uniqueness property. Let there be a set of all attributes in the relation, $K$, and the superkey set $S$. Therefore we can state $S \to \{x \in K \mid x \notin S\}$. A single relation may have more than one superkey, and superkeys can contain attributes that aren't strictly required. Generally, we are interested in superkeys where there isn't a subset of the superkey (hence the smallest superkey, by the irreducibility property), and this minimal superkey is referred to as the **candidate key**. If we have more than one candidate key, we can choose one of them to act as the primary key. A candidate key is also often referred to as just a **key**.

### Closure

The set of all attributes functionally determined by a set $L$, under a set of dependencies $F$, the closure of $L$, denoted as $L^+$. If $L^+$ contains all the attributes of $R$, then it follows that $L$ is a superkey of $R$. If RHS $\subseteq$ LHS$^+$, then LHS $\to$ RHS holds.

In order to compute the closure of a set of attributes, $L$, under a set of FDs given in the form LHS $\to$ RHS, we keep adding all the LHSs which exist as a subset of $L$, and add the RHS to $L$. We repeat this until we can no longer add anymore RHSs. The final value is referred to as $L^+$. Given the starting value of $L = \{A, B\}$, and the following FD set;

(1) $A, B \to C$

(2) $B, C \to A, D$

(3) $D \to E$

(4) $C, F \to B$

Starting with (1), both $A$, and $B$ are in $L$, so we can add $C$. Now that $C$ is in $L$, and $B$ was already there, we can add $D$ (no need to add $A$) from (2). Since we have $D$ in $L$, we can add $E$. There is nothing else we can do, since $F$ is not in $L$. Hence the value of $L^+ = \{A, B, C, D, E\}$.

### Armstrong's Axioms

This is a sound (doesn't generate incorrect FDs), and compete (allows us to derive all valid FDs) axiomisation of FDs. Given attributes $A, B, C, ...$, and sets of attributes $\alpha, \beta, \gamma$, we have the following axioms;

- reflexivity (trivial FDs) — $\alpha \to \beta$ always holds if $\beta \subseteq \alpha$
- augmentation — if $\alpha \to \beta$, then it follows $\alpha\gamma \to \beta\gamma$

- transitivity — if $\alpha \to \beta$, and $\beta \to \gamma$, then $\alpha \to \gamma$
- (derived) union — if $\alpha \to \beta$, and $\alpha \to \gamma$, then $\alpha \to \beta\gamma$
- (derived) decomposition — if $\alpha \to \beta\gamma$, then $\alpha \to \beta$, and $\alpha \to \gamma$
- (derived) pseudotransitivity — if $\alpha \to \beta$, and $\delta\beta \to \gamma$, then $\delta\alpha \to \gamma$

### Closure, and Covers

Similar to attributes, we can find the closure of a FD set, which is the set of all FDs that can be inferred, denoted as $F^+$. This can be approached by applying Armstrong's axioms. We start by initialising $F^+$ to the set $F$. By applying reflexivity, and augmentation, we can add new FDs to $F^+$. Transitivity can then be applied to suitable FDs, and adding the derived ones to $F^+$. This is repeated until no more changes occur.

We can say two FD sets $F_1$, and $F_2$ are equivalent if they each imply the other - if any relation instance satisfying $F_1$ also satisfies $F_2$, they are said to be covers of each other. A cover is canonical / minimal / irreducible if each LHS is unique, and no FD can be deleted from the cover, and still maintain an equivalent FD set. Additionally, no attributes can be deleted from any FD set, and still have an equivalent set. Therefore, the canonical cover has no redundant dependencies, nor attributes.

The method for testing whether some attribute, $X$, is extraneous depends on whether it's on the LHS, or RHS. For an attribute $X$ on the LHS, we can say $X$ is extraneous if RHS $\subseteq \{\text{LHS} - X\}^+$ under the FD set - note that here we are referring to closure on attributes, not FDs. On the other hand (no pun intended, I'm so depressed), an attribute $X$ on the RHS is extraneous if $X \in \text{LHS}^+$ under the FD set, where $X$ is removed from the RHS of all the FDs in the set.

To compute a canonical cover $F$ for an FD set, we can use the following algorithm;

```
let F := FD set
do
    replace dependencies of α → β and α → γ, with α → βγ
    remove all extraneous attributes one at a time
until F doesn't change
```

## Normalisation

As previously mentioned, in the previous section, normalisation reduces data duplication, and anomalies. One of the approaches taken to develop a normalised schema is to start with a few big relations, and then decompose them into a suitable normal form. The ones we'll look at are **Boyce-Codd Normal Form (BCNF)**, and **Third Normal Form (3NF)**. Consider the following example, which is just a "big" relation;

| title | year | length | genre | studio | actor |
|---|---|---|---|---|---|
| Star Wars | 1977 | 124 | Science Fiction | Fox | Carrie Fisher |
| Star Wars | 1977 | 124 | Science Fiction | Fox | Mark Hamill |
| Star Wars | 1977 | 124 | Science Fiction | Fox | Harrison Ford |
| Gone with the Wind | 1939 | 231 | Drama | MGM | Vivien Leigh |
| Wayne's World | 1992 | 95 | Comedy | Paramount | Dana Carvey |
| Wayne's World | 1992 | 95 | Comedy | Paramount | Mile Myers |

You'll notice straight away that there is a huge amount of redundant, repeated data, which is present in multiple rows. Updating the `length` of Star Wars would require 3 updates (for the relation to remain consistent), which is error prone. Inserting a new actor for a movie would require duplicating a fair bit of data, and would require checks to maintain consistency. Finally, deleting Vivien Leigh would delete the entire row, which deletes information about the movie.

### Decomposition

Given a relation $R(A_1, A_2, ..., A_n)$, we can decompose it into two projected relations $S$, and $T$, such that $\text{attr}(R) = \text{attr}(S) \cup \text{attr}(T)$. This also means that $S = \pi_{\text{attr}(S)}(R)$, and similar for $T$.

When we decompose a relation, it's crucial for us to ensure we can recover the original relation by joining the decomposed relations, and also preserve the FDs of the original relation. Note that the latter isn't always possible with BCNF.

Given $R$ decomposed into $S$, and $T$, the decomposition is **lossless** if at least one of the following FDs hold in the closure of FD set of $R$;

- $\text{attr}(S) \cap \text{attr}(T) \rightarrow \text{attr}(S)$
- $\text{attr}(S) \cap \text{attr}(T) \rightarrow \text{attr}(T)$
- this means the common attributes of $S$, and $T$ form a superkey of either of the decomposed relations

If we're able to check the FDs of $R$ without joining the decomposed sets, then the decomposition is **dependency preserving**.

Consider the following example, on $R(A, B, C)$, with the FD set $\{A \rightarrow B, B \rightarrow C\}$;

| decomposition | lossless? | dependency preserving? |
|---|---|---|
| $S(A, B)$<br>$T(B, C)$ | $B \rightarrow A, B$ doesn't hold<br>$B \rightarrow B, C$ holds $\therefore$ lossless | yes, $A \rightarrow B$ is checked with $S$, and $B \rightarrow C$ with $T$ |
| $S(A, B)$<br>$T(A, C)$ | $A \rightarrow A, B$ holds $\therefore$ lossless<br>$A \rightarrow A, C$ holds also | no, $B \rightarrow C$ cannot be checked without joining |

### Boyce-Codd Normal Form

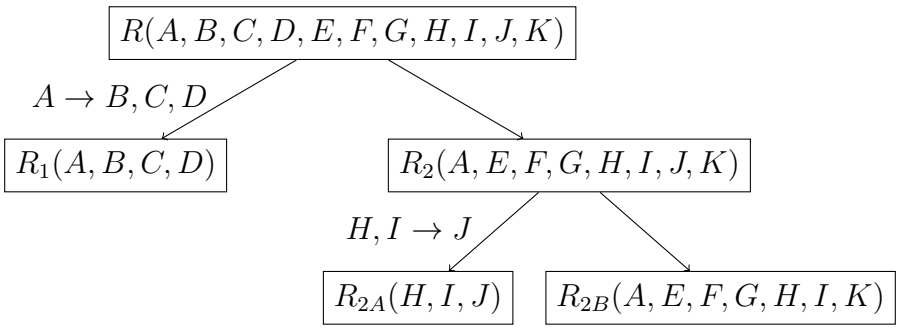
**A relation $R$ is in BCNF, if and only if, for all non-trivial FDs (including derived FDs) of the relation, the LHS of every FD is a superkey (i.e. contains a key)**

For example, if we have the FD `title`, `year` $\rightarrow$ `length`, `genre`, `studio`, it's not in BCNF, as `title`, `year` doesn't functionally determine `actor`.

In order to convert something into BCNF, we first initialise a set of relations with just $R$. While the set contains relations that violate the rules of BCNF, take one of them, let it be $V$. Let us take some non-trivial FD for $V$, of the form LHS $\rightarrow$ RHS; fulfilling the conditions $\text{LHS} \cap \text{RHS} = \varnothing$ (therefore we may need to check a derived FD for decomposed relations), and $\text{LHS} \rightarrow \text{attr}(V)$ does not hold for the $\text{FD}^+$ of $R$ (therefore, LHS is not a superkey of $R$). Now remove $V$ from the set of relations. Add a new relation $R_k(\text{LHS} \cup \text{RHS})$, and another relation $R_l(\text{attr}(V) - \text{RHS})$ to the set of decompositions. This process is then recursively applied to the child relations. Consider the following example;

$R(A, B, C, D, E, F, G, H, I, J, K)$, and the following FDs;

- $A \rightarrow B, C, D$
- $H, I \rightarrow J$
- $A, E, F, G \rightarrow H, I, K$



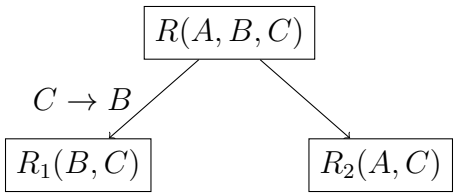
Honestly, just practice this.

### Recovering Data

When we decompose into BCNF, it's crucial that we're able to recover the original relation by joining the decomposed ones. As long as the FDs hold for the original relation, it will always be possible. However; if the FDs don't hold, we'd end up with false data; as the join would generate tuples that don't exist in the original relation.

### Dependency Preservation

As previosuly mentioned, we know that BCNF doesn't always preserve dependencies, which can be shown here. Let there be a relation $R(A, B, C)$, and the FD set; $\{\{A, B\} \to C, C \to B\}$. We wouldn't be able to check $A, B \to C$ without the RDBMS joining the relations.




### Third Normal Form

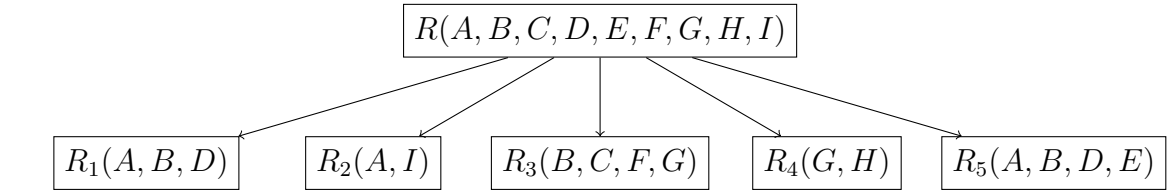
**A relation $R$ is in 3NF, if and only if, the LHS of every nontrivial FD is a superkey (same as BCNF test), or if every attribute on the RHS of a FD is prime (if it is a member of any key of the relation)**

When a BCNF decomposition doesn't preserve the original FDs, we can either "live with" the violating dependencies, or use a weaker normal form which preserves FDs, but has redundancy. With **3NF**, there always exists a dependency preserving, lossless decomposition.

In order to decompose a relation $R$, and a set of FDs $F$ for $R$, we can generate a set of decomposed relations $D$. We first need to find $C$, a canonical cover for $F$, such that it's a minimal FD set for $R$. Initialise $D$ to be $\varnothing$. For every canonical FD, in the form LHS $\to$ RHS, we add a new relation (LHS$\cup$RHS) to $D$. Now, for every relation in $D$, if it is a subset of another relation in $D$, we remove it. Finally, if none of the relations in $D$ include a key for $R$, we add a new relation to $D$, which contains a key.

Consider the following relation $R(A, B, C, D, E, F, G, H, I)$, a set of keys $\{\{A, B\}, \{A, D\}, \{B, D\}\}$ with the following FD set (and its canonical cover on the right);

| | |
|---|---|
| • $A, B \to D, E$ | $A, B \to D$ |
| • $A \to I$ | $A \to I$ |
| • $B \to C, F, G, H$ | $B \to C, F, G$ |
| • $G \to H$ | $G \to H$ |
| • $A, D \to B, C, E, F, G, H$ | $A, D \to B$ |
| • $B, D \to A, I, E$ | $B, D \to A, E$ |




dropped, as $R_1 \subseteq R_5$

## SQL

**Structured Query Language** is the most common language for relational databases, with use in more than 99% of applications. It supports schema creation, modification, data insertion, retrieval, updates, deletion, and a lot more. Translating from the relational algebra we previously done, to SQL, can be done with the table below;

| relational algebra | SQL | comments |
|---|---|---|
| $R \cup S$ | $R$ `UNION` $S$ | |
| $R \cap S$ | $R$ `INTERSECT` $S$ | |
| $R - S$ | $R$ `EXCEPT` $S$ | |
| $\pi_{\texttt{attributes}}(R)$ | `SELECT attributes FROM` $R$ | |
| $\sigma_{\texttt{condition}}(R)$ | `FROM` $R$ `WHERE condition` | |
| $R \times S$ | $R, S$ or $R$ `CROSS JOIN` $S$ | |
| $R \bowtie S$ | $R$ `NATURAL JOIN` $S$ | |
| $R \bowtie_{\texttt{condition}} S$ | $R$ `JOIN` $S$ `ON condition` | |
| relation | table | persistent relations stored on disk |
| relational expression | views | based on other relations, not normally stored |
| tuple | row | also called a record |
| attribute | column | also called a field |
| domain | type | includes `char`, `int`, `float`, `date`, `time`, etc. |

However, SQL differs from the theory we've studied so far, as it is based on multi-sets, and therefore can contain duplicate rows / tuples. It's still best practice to avoid them. Not all attributes need to be filled in, and therefore can be left `null`, once again this should be avoided; because of this, booleans are three-valued (true, false, and unknown). Most implementations have a wide range of types, and therefore numbers have multiple types - arithmetic opertors are usually available. Strings can either be a fixed length, and padded with spaces, or varying length. The concatenation operator used in SQL is `||`, and we can use pattern matching, with `_` denoting any character, and `%` matching zero, or more characters. We can store bits, bytes, and large binary objects (blobs) to hold images, movies, and other files.

With booleans, we have additional comparison operators such as `BETWEEN`, `NOT BETWEEN`, `IN`, and `NOT IN`, all of which should be fairly self-explanatory. The truth values for 3VL (three valued logic) can be computed with the following mapping;

- 1 - true
- $\frac{1}{2}$ - unknown
- 0 - false
- $x$ `AND` $y = \min(x, y)$
- $x$ `OR` $y = \max(x, y)$
- `NOT` $x = 1 - x$

It's also important to remember that any arithmetic which uses a `null`, will result in `null`. Any comparisons that use `null` will result in unknown.

### Queries

In order to save space, all comments will be done inline, in the SQL code listing below;

```
1 -- movie(title, year, length, genre, studio, producer)
2 -- casting(title, year, name)
3
```

```
-- note that the results of a selection is still a relation, and therefore we can use it as a subquery in other expressions
SELECT title, length/60 AS hours -- projected attributes (and a rename)
FROM movie -- the relation
WHERE studio='fox' AND year > 1990 -- the condition
ORDER BY year DESC, title ASC -- sorts first by year
-- note that we're using attributes which aren't in the projection, the order of operations is FROM, WHERE, ORDER, SELECT
-- note that we can also prefix the fields, to make it more readable
```

In order to join two relations, we can use a `JOIN`;

```
SELECT title, year, name
FROM movie JOIN casting ON (movie.producer=casting.name)
-- we can do a theta join with an ON, and a condition

SELECT title
FROM movie JOIN casting USING (title, year)
-- note that this is the same as ON (movie.title=casting.title AND movie.year=casting.year)

SELECT c1.name, c2.name
FROM casting AS c1 JOIN casting c2 -- note how we can omit AS for redability
ON c1.address = c2.address AND
   c1.name < c2.name
-- we can do a self join, but we have to name the relation, these are called correlation names
```

We can also consider the different types of joins, either natural (by mathcing attributes), or theta (by condition);

- inner join — returns tuples when there is at least one match in both sides
- left outer join (`LOJ`) — similar to inner join, but will include all tuples from the left relation
- right outer join (`ROJ`) — similar to inner, but includes all tuples from the right relation
- full outer join (`FOJ`) — similar to inner, but includes all unmatched tuples

| $L$ | | | $R$ | | | $L$ `LOJ` $R$ | | | | $L$ `ROJ` $R$ | | | | $L$ `FOJ` $R$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | a | b | c | a | b | c | d | a | b | c | d | a | b | c | d |
| 1 | 2 | 3 | 2 | 3 | A | 1 | 2 | 3 | A | 1 | 2 | 3 | A | 1 | 2 | 3 | A |
| 4 | 5 | 6 | 2 | 3 | B | 1 | 2 | 3 | B | 1 | 2 | 3 | B | 1 | 2 | 3 | B |
| 7 | 8 | 9 | 6 | 8 | C | 4 | 5 | 6 | N | N | 6 | 8 | C | 4 | 5 | 6 | N |
| | | | | | | 7 | 8 | 9 | N | | | | | 7 | 8 | 9 | N |
| | | | | | | | | | | | | | | N | 6 | 8 | C |

```
SELECT DISTINCT title, year, name -- selects only unique tuples
FROM movie LEFT OUTER JOIN casting ON
    movie.producer=casting.name AND movie.year=casting.year
```

We also have a set of aggregation functions, which can be used to calculate a single value, for example;

```
SELECT department
       COUNT(*) as professors,
       SUM(salary) as totalsalary,
       AVG(salary) as averagesalary,
       MIN(age) as youngest,
```

```
6         MAX(age) as oldest
7  FROM employee
8  WHERE position='Professor'
9  GROUP BY department
10 HAVING COUNT(*) >= 10
11 ORDER BY totalsalary DESC
```

This creates a tuple for each department; the results are then sorted in descending order by total salary. Any non-aggregates used in the projection, or the `HAVING` filter **must** be included in the `GROUP BY` list. Only departments with at least 10 professors are listed.

### Subqueries

A powerful feature of `SELECT`s is that we can also use them as subqueries in expresisons, by wrapping them in brackets. The types supported by SQL are;

- scalar subquery

  a subquery that produces a single value, normally uses an aggregate function

```
1  SELECT title,
2      (SELECT count(name)
3       FROM casting
4       WHERE casting.title=movie.title) AS numactors
5  FROM movie
6  -- note how the outer query is related to the subquery; this is an example of
       a correlated subquery that has to be evaulated for each outer tuple
7
8  SELECT title, count(name) as numactors
9  FROM movie JOIN casting USING (title)
10 GROUP BY title
11 -- this is equivalent, and is clearer
```

- set subquery

  creates a set of distinct values (a single column), typically used for set membership with `(NOT) IN`, or set comparisons with `ALL` or `SOME`

```
1  SELECT title
2  FROM movie
3  WHERE studio IN (SELECT name
4                   FROM studio
5                   WHERE address LIKE 'C%')
6  SELECT name
7  FROM casting
8  WHERE (title, year) NOT IN (SELECT title, year
9                              FROM movie
10                             WHERE genre='sf')
11 -- note how we're able to match on tuple values
12 SELECT title
13 FROM movie m1
14 WHERE year < SOME(SELECT year
15                   FROM movie m2
16                   WHERE m2.title=m1.title)
17 -- the keywords ALL, and SOME can be used with any comparator
18
19 SELECT name
```

```
FROM employee
WHERE salary <> ALL(SELECT salary
                    FROM employee
                    WHERE position='Professor'
```

- relation subquery

  produces a relation, which is typically used as an operand, used with operators `(NOT) EXISTS` to check if it's empty, or operators `(NOT) UNIQUE` to test for duplicate tuples

```
SELECT title
FROM movie m1
WHERE NOT EXISTS(SELECT *
                 FROM movie m2
                 WHERE m2.title=m1.title AND m1.year<>m2.year)
```

## SQL Data Definition

SQL's DDL is concerned with schema creation, as well as the specification of any constraints. This course mainly focuses on base relations (which are stored tables), and lightly touches on derived relations (views). The constraints we deal with in SQL include type, primary / foreign key constraints, uniqueness, checking a value exists (not null), check constraints, as well as assertions.

### Creating a Relation

Given some relation `movie(title, year, length, genre)`, the following SQL instruction will create the table;

```
CREATE TABLE movie {
    title VARCHAR(120),
    year INT DEFAULT 2011,
    length INT DEFAULT 0,
    genre CHAR(20),
    PRIMARY KEY (title, year)
}
```