

Data Modelling & Databases

Janis Hutz
<https://janishutz.com>

June 23, 2026



“A curry can be written in many different ways”

- Prof. Dr. Gustavo Alonso, 2026

FS2026, ETHZ

Summary of the Lecture Slides

<https://systems.ethz.ch/education/courses/2026-spring/dmdb.html>

Contents

1	Introduction	3
1.1	IMS	3
1.2	Network Model	3
1.3	Relational Model	3
2	Relational Model, Logic & Algebra	4
2.1	Relational Model	4
2.1.1	Keys	4
2.1.2	Queries	4
2.2	Relational Algebra	5
2.2.1	Operators	5
3	SQL	6
3.1	Data Definition Language (DDL)	6
3.1.1	Creating Tables	6
3.1.2	Updating / Altering Tables	6
3.1.3	Deleting Tables	6
3.1.4	Constraints	7
3.2	Data Manipulation Language (DML)	8
3.3	Query Language	8
3.3.1	Basic operators	9
3.3.2	Join	10
3.3.3	Aggregations, Grouping, Sorting	11
3.3.4	Null	12
3.3.5	Views	12
4	Theoretical Background	13
4.1	Functional Dependency	13
4.1.1	Inference	13
4.1.2	Armstrong's Axioms	13
4.1.3	Other rules	13
4.1.4	Closure Algorithm	14
4.1.5	Using Functional Dependencies	14
	Glossary	15

1 Introduction

A note on the quote on the title page: He did not *actually* mean “curry”, but of course “query”, but he just pronounced it that way.

This summary assumes some basic knowledge about databases. All words written in **bold** should be clickable and should link you to the corresponding term in the glossary. If I did not forget a term that is, of course.

1.1 IMS

Early databases were not relational and they also required the user of the system to know *how the data was stored*. This of course is not ideal, as any update to the data structure also required an update to the code interacting with it. In addition, manual query optimization has to be done.

Non-relational databases commonly have data redundancy, due to non-hierarchical applications being forced into a tree-based model. This causes issues when updating the data, as one needs to update the data in all places where a copy of the data exists.

1.2 Network Model

In a network model, one **record** is fetched at a time, so it essentially traverses the network. However, we again do not have data independence, thus the application again needs to change when the physical data representation changes. In addition, manual query optimization has to happen again, thus, any change to the physical data representation may cause the need to redo the optimization.

1.3 Relational Model

This is the focus of this course

The relational model has the benefit of having data independence: The application doesn't know how the data is stored and represented. The queries stay the same, even if the data representation changes. In addition, the end user does not have to worry about optimization too much. This, of course, only applies to a limited extent.

2 Relational Model, Logic & Algebra

2.1 Relational Model

In the relational model, relations are sets of tuples (similar to **records**). They are denoted $R(f_1 : D_1, \dots, f_n : D_n)$, or sometimes simply $R(D_1, \dots, D_n)$. An **instance** is a **set** of tuples $I_R \subseteq D_1 \times \dots \times D_n$.

Intuition: You can think of the instance to be the rows in the table.

Important We define relations as **mathematical sets**. This means that they cannot contain duplicates and the order does not matter. Database engines may elect to do this differently, but this course assumes set semantics for the relational model.

2.1.1 Keys

Definition 2.1.1 (*Candidate Key*) The minimal set of fields that identify each tuple uniquely

Definition 2.1.2 (*Primary Key*) A single candidate key, i.e. just a single field. We mark the primary key using underlining in visual **schemas**. Formally, we write $R(\underline{k} : D_k, a : D_a, b : D_b)$ for a given relation, with all valid instances fulfilling

$$I \subseteq D_k \times D_a \times D_b \wedge \forall (k, a, b), (k', a', b') \in I : k = k' \rightarrow (a, b) = (a', b')$$

i.e. if the key is equal, so is the rest of the tuple (thus they are one and the same¹).

2.1.2 Queries

We write queries in a relational model using the following set notation (using **SQL** for an actual DB):

$$\{(r, p) \mid r \in \text{Supplies} \wedge p \in \text{Parts} \wedge r.\text{pid} = p.\text{pid} \wedge p.\text{name} = \text{"A"}\}$$

The query language thus processes an entire set at a time and applies set operations over the relations.

¹ Heidrun is her name

2.2 Relational Algebra

2.2.1 Operators

- **Union** $\cup$: $x \in R_1 \cup R_2 \Leftrightarrow x \in R_1 \vee x \in R_2$ (All tuples from both sets (Only valid for same schemas))
- **Difference** $-$: $x \in R_1 - R_2 \Leftrightarrow x \in R_1 \wedge x \notin R_2$ (Tuples that appear in R_1 , but not in R_2)
- **Intersection** $\cap$: $x \in R_1 \cap R_2 \Leftrightarrow x \in R_1 \wedge x \in R_2$ (Tuples that don't appear in both sets)
- **Selection** σ : $x \in \sigma_c(R) \Leftrightarrow x \in R \wedge c(x) = \text{true}$, with c a predicate on the passed in set. (Tuples that fulfil the predicate c)
- **Projection** Π : $\Pi_{A_1, \dots, A_n}(R)$ (Keep only a subset of the columns, e.g. for columns `name`, `pid`, $\Pi_{\text{name}}(R)$ returns only the column `name`)
- **Cartesian Product** $\times$: $(x, y) \in R_1 \times R_2 \Leftrightarrow x \in R_1 \wedge y \in R_2$ (Primarily used to express join operations. This however simply joins together the two tables (i.e. similar to expanding terms in maths))
- **Renaming** ρ : $\rho_{B_1, \dots, B_n}(R)$ (Change the name of the attributes of R to B_i)
- **Natural Join** $\bowtie$: $R_1(A, B) \bowtie R_2(B, C) = \Pi_{A, B, C}(\sigma_{R_1.B=R_2.B}(R_1 \times R_2))$. (Join two relations into a table on a column. Natural join joins on columns with same name in both (or all) relations) Edge cases:
 - No shared attributes $R(A, B, C), S(D, E)$: $R \bowtie S = R \times S$
 - All attributes shared $R(A, B, C), S(A, B, C)$: $R \bowtie S = R \cap S$
- **Theta Join** $\bowtie_\theta$: $R_1 \bowtie_\theta R_2 = \sigma_\theta(R_1 \times R_2)$ (Join with custom predicate θ)
- **Equi-Join** $\bowtie_{A=B}$: $R_1 \bowtie_{A=B} R_2 = \sigma_{A=B}(R_1 \times R_2)$ (Join column A with column B)
- **Semi-Join** $\ltimes_C$: $R_1 \ltimes_C R_2 = \Pi_{A_1, \dots, A_n}(R_1 \bowtie_C R_2)$, with $R_1(A_1, \dots, A_n)$ and $R_2(B_1, \dots, B_m)$ (Returns columns only from one side if there is a match in the join)
- **Relational division** $\div$: $R \div S = \Pi_{R-S}R - \Pi_{R-S}((\Pi_{R-S}R) \times S - R)$. In other words, $R \div S = T$, with T being the *largest* relation such that $S \times T \subseteq R$. (Find all rows that do not fulfil a condition)

Semi-Join Reduction A useful trick with semi-joins. It is especially useful in distributed DB, where R and S are on different machines. Suppose we want to Join $R(A, B)$ with $S(B, C)$ on B , then we can do the following $R \ltimes S = (R \times \Pi_B S) \ltimes S$.

The idea is to send the column on which the relations are to be joined from the second system to the first, there execute a semi-join (thus only returning the contents of R that matched with contents of S), then sending only that data to the second machine to finish the full join operation.

Of course, in the real world, users of DB systems don't write relational algebra, as in this case, the order of operations matters for performance, whereas we want database systems to figure this out by themselves.

3 SQL

SQL, or unabbreviated, Structured Query Language, is a **declarative** programming language, used to describe and operate on databases.

This is the point to mention [xkcd standards comic](#), because, as is the case with almost any standard, people deviate from it and new standards form.

Even though *most* **SQL** databases support the standard SQL operations, many have added a lot of features on top.

SQL is split up into the following parts:

- Data Definition Language, used to create, modify and delete schemas.
- Data Manipulation Language, used to insert, update and delete data
- Query Language, used to retrieve data

3.1 Data Definition Language (DDL)

The DDL is used to describe the schema of relations, in **SQL** called tables. For that, we provide a table name, the columns and their data types.

SQL supports the following data types (plus more):

- | | | |
|--------------|------------|--------------------------------|
| ▪ CHAR(n) | ▪ BIGINT | ▪ BLOB(n) (for large binaries) |
| ▪ VARCHAR(n) | ▪ SMALLINT | ▪ DATETIME |
| ▪ INT | ▪ FLOAT | ▪ MONEY |

3.1.1 Creating Tables

Use the DDL CREATE TABLE keyword to create a table. We can set default values for certain attributes, or can add constraints to them.

Since the primary key must be unique for each entry, it may be useful to configure the primary key attribute with the following constraints `PrimaryKeyField INT NOT NULL AUTO_INCREMENT`

File: `code/sql/ddl/create.sql`

```
1 CREATE TABLE TableName (  
2     Attribute integer,  
3     OtherAttribute varchar (30),  
4     NextAttribute character (2) default "AP", -- default value, if unset on insert  
5     PRIMARY KEY (Attribute) -- primary key, as in RA  
6 );
```

3.1.2 Updating / Altering Tables

Use the DDL ALTER TABLE keyword to update a table's schema. Be aware that if we ADD a column to the schema, unless a default value is provided, the column is set to NULL (see 3.3.4)

File: `code/sql/ddl/alter.sql`

```
1 ALTER TABLE TableName ADD COLUMN (col integer);  
2 ALTER TABLE TableName ADD COLUMN (AnotherColumn integer default "");  
3 ALTER TABLE TableName DROP COLUMN AnotherColumn;
```

3.1.3 Deleting Tables

File: `code/sql/ddl/delete.sql`

```
1 DROP TABLE TableName;
```

3.1.4 Constraints

SQL supports a number of constraints that can be put on the different attributes of a schema.

- **NOT NULL**: Disallows this column to be set to NULL, or have a NULL value
- **UNIQUE**: The value of each value in this column must be unique, but it can be NULL. An arbitrary number of columns can have NULL and still be valid
- **PRIMARY KEY**: Sets this column as the primary key. NOT NULL and UNIQUE is set on it implicitly. Contrary to common intuition, it can be set on multiple columns.
- **CHECK c**: Check that the values fulfil a condition. This condition has the same syntax as for the WHERE clause (see 3.3.1)
- **REFERENCES table**: A Foreign Key, used to refer to another tuple from a different relation. It is typically referencing the primary key of the other table.

For REFERENCES, there are many options for maintenance, which can be set on creating a reference:

- **Cascade**: Propagate UPDATE and DELETE
- **Restrict**: Prevent deletion of the primary key before the change, causes error
- **No Action**: Prevent modifications after attempting change, causes error
- **Set default, Set Default**: Set references to NULL or default value

Example 3.1.4

```
CREATE TABLE tab (id integer REFERENCES OtherTable ON DELETE cascade ON UPDATE cascade)
```

3.2 Data Manipulation Language (DML)

The DML is used to insert, update and delete data from the database.

Below some examples showing the use of the common operations. Note that you can use a WHERE clause from the query language to filter data for both DELETE and UPDATE statements.

File: code/sql/dml.sql

```
1 INSERT INTO TableName (
2     Attribute,
3     OtherAttribute
4 ) VALUES ( 1000, 'Value' );
5
6 DELETE FROM TableName WHERE Attribute > 50;
7
8 UPDATE TableName SET Attribute = Attribute + 1;
9
10 -- Import data from a CSV file (this is postgresql specific syntax)
11 COPY TableName FROM '/data.csv' WITH FORMAT csv;
```

3.3 Query Language

The basic structure of a **SQL** statement is SELECT I FROM T WHERE C, which corresponds to $\Pi_I(\sigma_C T)$.

We can set I = * if we want to return all columns for a table. To rename, we can set I = Column as Name, Column2 as Name2, etc

Theorem 3.3.1 Every SPJR RA expression can be written in SELECT ... FROM ... WHERE ... form:

Operation	Notation	SQL
Selection	$\sigma_c(R)$	SELECT * FROM R WHERE c;
Projection	$\Pi_{A_1, \dots, A_n} R$	SELECT A1, ..., An FROM R;
Cartesian Product	$R_1 \times R_2$	SELECT * FROM (R1, R2);
Rename	$\rho_{a,b,c} R$	SELECT A as a, ..., B as c FROM R;
Union	$R_1 \cup R_2$	R1 UNION R2;
Difference	$R_1 - R_2$	R1 EXCEPT R2;
Intersection	$R_1 \cap R_2$	R1 INTERSECT R2;

Since **SQL** implements **bag** and not set semantics, there is a DISTINCT keyword, which is used to remove duplicates. It is to be applied in the SELECT clause (I above), e.g.

```
SELECT DISTINCT name FROM Data;
```

For the last three operations, R1 and R2 typically are other **SQL** statements, example:

```
(SELECT name FROM FirstTable) UNION (SELECT name FROM SecondTable)
```

In addition, to apply *bag semantics*, as opposed to *set semantics* append ALL to the expression (e.g. UNION ALL)

We can also name the tables, e.g. T = One o, Two t, and then access columns from a specific table using I = o.Col1, b.Col1, or the like.

3.3.1 Basic operators

3.3.1.1 Logical Operators

The following **SQL** logical operators are available (may not be exhaustive for all systems, `cop` is a comparison operator):

Operator	Description
<code>P1 AND P2</code>	TRUE if P1 and P2 both are true
<code>P1 OR P2</code>	TRUE if one of P1 or P2, or both are true
<code>C cop ALL query</code>	TRUE if all values <code>x</code> returned in subquery query fulfil <code>C cop x</code>
<code>C cop ANY query</code>	TRUE if one (or more) values <code>x</code> returned in subquery query fulfil <code>C cop x</code> . SOME is logically equivalent
<code>C BETWEEN low AND high</code>	TRUE if numerical, text or date value of <code>C</code> is between <code>low</code> and <code>high</code> . Both ends inclusive, negatable
<code>C IN list</code>	TRUE if value of <code>C</code> is in the provided list. List format: <code>(val, val2)</code> . List can also be a subquery , negatable
<code>C LIKE R</code>	TRUE if value of <code>C</code> matches regex-like pattern <code>R</code> . (<code>%</code> matches any number of chars, <code>_</code> matches a single, arbitray character)
<code>EXISTS query</code>	TRUE if subquery query returns at least one row

3.3.1.2 Comparison & Arithmetic Operators

Comparison Operators `<`, `<=`, `=`, `>=`, `>` are defined as usually, `<>` is defined as not equal.

Arithmetic Operators `+`, `-`, `*`, `/`, `%` are defined as usual.

We can use the arithmetic operators like this:

```
SELECT Value * 1.1 FROM Data;
```

or in WHERE clauses like this:

```
SELECT Value FROM (Data a, Data b) WHERE a.Value = b.Value - 1;
```

Remark 3.3.2 You may have noticed that in the above query, the same table was used twice. This is referred to as a **Self-Join** and can come in handy when comparing values in a single table.

3.3.2 Join

Specifying two (or more) tables in the FROM clause will perform a cartesian product, e.g.

```
SELECT * FROM (Table1, Table2);
```

returns all rows of the tables, next to each other.

Here, naming the tables in the FROM clause may come in handy, to not write out the entire name of the tables:

```
SELECT * FROM (Table1 t, Table2 s) WHERE t.id = s.id;
```

```
SELECT * FROM (Table1, Table2) WHERE Table1.id = Table2.id;
```

Both of the above statements produce the same result.

3.3.2.1 Nested Queries

To not have the cartesian product executed in the FROM clause, or to get a much more complex starting table, we can use a **subquery**. These are parenthesized normal SQL queries.

We can create an alias for them using an AS name clause after the parenthesis as follows

```
SELECT r.name FROM (SELECT * FROM Table) AS r;
```

or alternatively, using a WITH clause (replace the dummy queries with real queries):

```
WITH CteName (SELECT * FROM Table) SELECT name FROM CteName;
```

3.3.2.2 Join Operations

If we don't specify the kind of join, an INNER JOIN is executed. The following three queries are equivalent

```
SELECT * FROM Student, Tests WHERE Student.PersNr = Tests.PersNr;
```

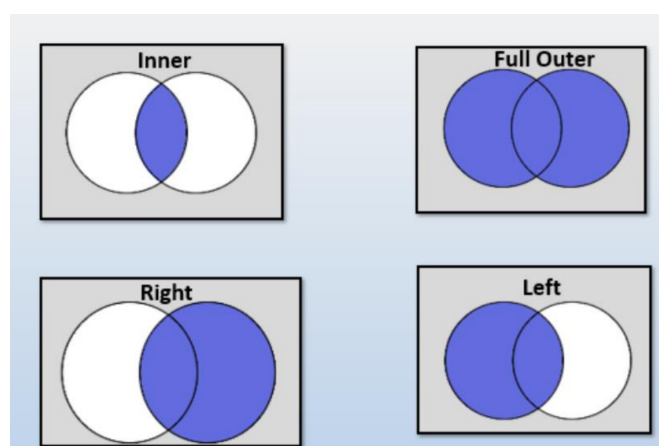
```
SELECT * FROM Student JOIN Tests ON Student.PersNr = Tests.PersNr;
```

```
SELECT * FROM Student JOIN Tests USING (PersNr);
```

For the latter of the three, the parenthesis around the column are important, as omitting them is invalid syntax.

Other types of JOIN are:

- NATURAL JOIN (no ON required), joins on columns with same name
- LEFT/RIGHT/FULL OUTER JOIN (requires ON), also returns non-matching rows from left, right or both sides, respectively



3.3.3 Aggregations, Grouping, Sorting

3.3.3.1 Aggregations

SQL supports (at least) the aggregation functions SUM, MIN, MAX, AVG, COUNT, whose output should be evident.

Please note that you cannot use the aggregation functions and still output another column, as they produce a single value.

For each of the functions, you can optionally specify DISTINCT.

We typically want to specify an AS clause, to make the output nicer to read

3.3.3.2 Grouping

Grouping is used to, well, group together the outputs on a column. Using SQL, this is achieved using the GROUP BY clause. We specify the columns on which the grouping should occur.

IMPORTANT We can only use aggregates and columns appearing in the GROUP BY clause in the SELECT clause.

Using a JOIN function, it is of course then possible to go back retrieve the other columns.

To apply filtering on the columns, the HAVING clause can be used. It applies a comparison to a grouping column or an aggregate as specified in the SELECT clause:

```
SELECT COUNT(name) as cnt FROM Students GROUP BY Department HAVING cnt > 1;
```

3.3.3.3 Sorting

We can sort the output of any query using the ORDER BY column DIRECTION clause, where DIRECTION is to be replaced with ASC or DESC:

```
SELECT name FROM Students ORDER BY name DESC;
```

3.3.3.4 Limit

We can limit the number of columns to return using the LIMIT clause. Example:

```
SELECT * FROM Exams LIMIT 10;
```

3.3.3.5 Example

Below some examples of SQL Queries using grouping, aggregations, sorting and grouping

File: code/sql/aggregations.sql

```

1 SELECT MAX(price) as MAX_PRICE FROM Products; -- Get the most expensive product
2 SELECT AVG(grade) as AVG_GRADE FROM Exams WHERE id = 0; -- Get the average grade for an exam
3
4 -- Aggregation with grouping (Remember, only aggregates and cols in GROUP BY clause can
5 -- appear in SELECT clause!)
6 SELECT id, AVG(grade) as "Average Grade", COUNT(grade) as "Count"
7 FROM ExamResults GROUP BY id; -- Average grade for each exam and number of students taking it
8
9 SELECT id, AVG(grade) as avg, COUNT(grade) as cnt FROM ExamResults
10 GROUP BY id HAVING cnt > 0 ORDER BY avg DESC LIMIT 20;
11
12 -- Get exam details for all exams as filtered before
13 SELECT * FROM (
14     SELECT id, AVG(grade) as avg, COUNT(grade) as cnt FROM ExamResults
15     GROUP BY id HAVING cnt > 0 ORDER BY avg DESC LIMIT 20
16 ) as filtered JOIN Exams e ON filtered.id = e.id;
```

3.3.4 Null

Whenever we have incomplete information, we can set a column to be NULL. If you have a column that has NULL values, when using WHERE clauses, these values are skipped. *However* if you are not filtering on the column, a NULL value is returned.

To check if a column is NULL, you can use the IS keyword:

```
SELECT * FROM Data WHERE name IS NULL
```

- In GROUP BY, all NULLs form a single group.
- COUNT(*), also counts rows with NULL
- COUNT(TheColWithNull) ignores rows with NULL
- Most other aggregations ignore NULL
- If all rows in an aggregations are NULL, so is the result

The OUTER JOIN operators may (and likely will) introduce NULL

3.3.5 Views

Views are a useful way of abstracting a complex query and is the result of a stored query.

We can create views using the following template:

```
CREATE VIEW name AS query;
```

We can then operate on it as if it were a table.

Common applications for views are privacy / access control, because you can limit access to tables only via certain views for every user; Usability is also often much better due to having less complex queries to write - you are just writing them once.

How a view is evaluated This is quite simple in essence, the VIEW's query is inserted into the query using the view. It then is then optimized as it normally would be, too. Of course, it is likely that more optimization can and has to be done on views.

3.3.5.1 Updatable Views

Updating Views refers to updating values of the tables in the views. This is not straight forward and thus, many restrictions are placed on what kind of views are considered updatable.

In **SQL**, a view is updatable if and only if it involves **one** base relation and its key, it does **not** involve aggregates, groups or duplicate elimination.

In short, if there is one-to-one mapping between the view and base relation, it is updatable.

4 Theoretical Background

4.1 Functional Dependency

As with relational algebra, we again define a schema to be a relation, e.g. $\mathcal{R}(A : D_A, B : D_B, C : D_C, D : D_D)$, with an instance of it being $R \subseteq D_A \times D_B \times D_C \times D_D$.

Then, we let $\alpha, \beta \subseteq \mathcal{R}$, i.e. they each have a subset of the columns of $\mathcal{R}$.

$\alpha \rightarrow \beta$ (β is a functional dependency of α) if and only if $\forall r, s \in R : r.\alpha = s.\alpha \implies r.\beta = s.\beta$

We write $R \models \alpha \rightarrow \beta$ if R satisfies $\alpha \rightarrow \beta$ semantically, and $R \vdash \alpha \rightarrow \beta$ if the same applies syntactically (i.e. we can apply inference rules over Functional Dependencies).

Intuition: This means that there is a function that maps the values of columns α to the values of columns β .

Definition 4.1.1 $\alpha \subseteq \mathcal{R}$ is a **superkey** if and only if $\alpha \rightarrow \mathcal{R}$.

Intuition: This means that if we know the values of columns α , we know the value of the rest of the columns in $\mathcal{R}$.

Definition 4.1.2 $\alpha \rightarrow \beta$ is minimal if and only if $\forall A \in \alpha : (\alpha - \{A\}) \not\rightarrow \beta$. These are denoted $\alpha \rightarrow \beta$.

4.1.1 Inference

A fundamental result for a given set of Functional Dependencies F is $F \models \alpha \rightarrow \beta \Leftrightarrow F \vdash \alpha \rightarrow \beta$ (assuming $\vdash$ is defined by Armstrong's Axioms)

The goal is to find new FDs that can be implied from a set of FDs F on scheme $\mathcal{R}$. Let $\alpha \rightarrow \beta$ be another FD on $\mathcal{R}$. Then:

- F implies $\alpha \rightarrow \beta$, if every relation instance R of $\mathcal{R}$ that satisfies all FDs in F also satisfies $\alpha \rightarrow \beta$.
- The **closure** F^+ of F is the set of all FDs implied by F .
- Let G be another set of FDs on scheme $\mathcal{R}$. F and G are **equivalent** ($F \equiv G$) if $F \models G$ and $G \models F$

4.1.2 Armstrong's Axioms

- **Reflexivity** $\alpha \subseteq \beta \implies \beta \rightarrow \alpha$, special case: $\mathcal{R} \rightarrow \alpha$ (referred to as *trivial FDs*)
- **Augmentation** $\alpha \rightarrow \beta \implies \alpha\gamma \rightarrow \beta\gamma$, with $\alpha\gamma = \alpha \cup \gamma$
- **Transitivity** $\alpha \rightarrow \beta \wedge \beta \rightarrow \gamma \implies \alpha \rightarrow \gamma$

These three axioms are all complete and sound. All other possible FDs can be implied from these axioms.

- F **derives** $f = \alpha \rightarrow \beta$, denoted $F \vdash f$, if there is a derivation for f using only Armstrong's axioms.
- F **implies** f , denoted $F \models f$, if for every relation instance R of $\mathcal{R}$ that satisfies all FD in F also satisfies f .

Definition 4.1.3 (Soundness) $F \vdash f \implies F \models f$

Definition 4.1.4 (Completeness) $F \models f \implies F \vdash f$

Definition 4.1.5 (Closure) of α with respect to F α^+ is the set of all attributes $y \in \mathcal{R}$ such that $\alpha \rightarrow y$ can be derived from F using Armstrong's axioms:

$$\alpha^+ = \{y \in \mathcal{R} \mid F \vdash \alpha \rightarrow y\} \quad F \vdash \alpha \rightarrow \beta \Leftrightarrow \beta \subseteq \alpha^+$$

4.1.3 Other rules

- **Union of FDs** $\alpha \rightarrow \beta \wedge \alpha \rightarrow \gamma \implies \alpha \rightarrow \beta\gamma$
- **Decomposition** $\alpha \rightarrow \beta\gamma \implies \alpha \rightarrow \beta \wedge \alpha \rightarrow \gamma$
- **Pseudo Transitivity** $\alpha \rightarrow \beta \wedge \beta\gamma \rightarrow \theta \implies \alpha\gamma \rightarrow \theta$

4.1.4 Closure Algorithm

For the definition of the closure, see above.

Finding a closure α^+ using an algorithm²:

Algorithm 1 Finding Closure

```

1 procedure FINDCLOSURE( $F$ )
2    $\alpha^+ \leftarrow \alpha$ 
3   repeat
4      $\alpha_{old}^+ \leftarrow \alpha^+$ 
5     for each FD  $\beta \rightarrow \gamma \in F$  do
6       if  $\beta \subseteq \alpha^+$  then
7          $\alpha^+ \leftarrow \alpha^+ \cup \gamma$ 
8   until  $\alpha^+ = \alpha_{old}^+$ 
9   return  $\alpha^+$ 
  
```

We can use that to check the following:

- $F \vdash \alpha \rightarrow \gamma$: Calculate α^+ and check $\gamma \in \alpha^+$
- $F \vdash G$: For each $\alpha \rightarrow \gamma \in G$, check $F \vdash \alpha \rightarrow \gamma$
- F equivalent to G : Check $F \vdash G$ and $G \vdash F$
- $K \subseteq \mathcal{R}$ superkey for F : Check $F \vdash K \rightarrow \mathcal{R}$:

Minimal Cover

Definition 4.1.6

A minimal cover (or minimum basis) of F is a set of FDs G that has the following properties:

- $G \equiv F$
- All FDs in G have form $X \rightarrow A$, with A being a single attribute
- It is not possible to make G “smaller”, i.e. if deleting a FD, $G - \{X \rightarrow A\} \not\equiv G$, or $G - \{XA \rightarrow B\} + \{X \rightarrow B\} \not\equiv G$, for any FD $XA \rightarrow B \in G$

Computing the minimum basis:

1. G set of FDs obtained from F by decomposing the right hand sides of each FD to single attribute
2. Remove trivial FDs
3. Remove redundant attributes from LHS of FDs in G (by, for each $X \rightarrow Y$ taking each attribute $x \in X$ and, if $X - \{x\} \rightarrow Y$ implies $X \rightarrow Y$, replacing $X \rightarrow Y$ with $X - \{x\} \rightarrow Y$)
4. Remove all redundant FDs

4.1.5 Using Functional Dependencies

Functional dependencies help with decomposing relations into several relations that each contain just one concept, since relations combining several concepts are bad.

²Yes, I did name the algorithm that. Sorry about that, the pun was too good to skip

Glossary

bag A bag, or multiset, is a set, in which, (as is usual with sets) order does not matter, but duplicates are allowed.

declarative Describes *what* to achieve, but not how.

imperative Describes how to achieve something.

record .

schema A set of relations.

SQL Structured Query Language.

subquery Also Nested Query, a separate query that can be used in the FROM or WHERE clause of another query. See 3.3.2.1.