# CSC343
# Guide

Pierre-William Lessard

# Contents

# Chapter 1

# Relations

## 1.1 The Relational Model

**Definition 1.1: Relation**

A set of tuples that have a certain number of attributes, or columns.
Each tuple in the relation has the same number of attributes, and each attribute has a unique name.

A relation is defined by schemas which indicate the structure of a relation.
Instances of relations include tuples which all follow a relation's schema.

Relations are traditionally thought of as tables. Where the rows are denoted as **tuples** and the columns as **attributes**. The **arity** of a relation is its number of attributes, and its **cardinality** is the number of tuples within a relation instance.

In Relational Databases, a relation is a set of tuples, meaning:

- There are no duplicate tuples
- The order of tuples does not matter

**Definition 1.1.1: Key**

A key for a relation, $R$, is defined as a set of attributes $a_1, a_2, \ldots, a_n$ such that:

- $\forall$ tuples $t_1, t_2 \in R.\ \exists i \in [1, n] \cap \mathbb{N}.\ t_1[a_i] \neq t_2[a_2]$
  This means that no two tuples may have all the attributes denoted in the key, agree.
- There is no subset of $a_1, a_2, \ldots, a_n$ with this same property

This means that keys uniquely identify tuples in a relation's instance, as there must not exist another tuple with the same values for the attributes indicated by the key.

Keys are denoted in the schema by underlining the attributes of the key. A **superkey** is any set of attributes which meet the first property defined above, yet not necessarily the second.

## 1.2 Relational Algebra

Relational algebra is a type of algebra where the operands are instances of relations. It is a formal system for manipulating and querying data stored in relational databases. It is based on a set of fundamental operations that allow us to combine data from multiple tables in a declarative way. Some of the most common operations in relational algebra include selection, projection, union, intersection, and difference. These operations allow us to specify the data we want to retrieve from a database, as well as the relationships between different data sets. Relational algebra is an important theoretical foundation for the design and implementation of database systems.

### 1.2.1 Projection

Projection refers to the operation that extracts certain columns from a relation (table). For example, if you have a relation with columns "name", "age", "gender", and "country", and you only want to see the "name" and "age" columns, you would use the projection operation to create a new relation that only contains those columns. This operation is useful for simplifying and organizing the information in a relation, and for focusing on the specific data that you are interested in.

**Definition 1.2.1: Projection**

Projection is denoted with the $\pi$ symbol. Let $R$ be an arbitray relation with some subset of attributes, $a_1, a_2, \ldots, a_n$. Projecting $a_1, a_2, \ldots, a_n$ onto $R$ is notated as:

$$\pi_{a_1, a_2, \ldots, a_n} R$$

The result is a new relation with all the same tuples as $R$, but only including attributes $a_1, a_2, \ldots, a_n$

### 1.2.2 Selection

Selection is an operation that selects a subset of tuples from a relation. It is typically denoted by the symbol $\sigma$, and it is typically used in combination with other operations such as projection and soon to be learned join. Selection is a fundamental operation in relational algebra and is used to specify the desired subset of a relation in order to answer a query. For example, consider a relation R with attributes A, B, and C, and suppose we want to select all tuples from R where the value of attribute A is greater than 5. We can use the select operator to return a relation where this is true.

**Definition 1.2.2: Selection**

Selection is denoted with the $\sigma$ symbol. Let $R$ be an arbitray relation with some subset of attributes, $a_1, a_2, \ldots, a_n$. Let $L(a_1, a_2, \ldots, a_n)$ be a logical statement incorporating a subset attributes of $R$. Selecting tuples which render $L(a_1, a_2, \ldots, a_n)$ true in $R$ is notated as:

$$\sigma_{L(a_1, a_2, \ldots, a_n)} R$$

The result is a new relation with all the same tuples as $R$, but only including tuples where $L(a_1, a_2, \ldots, a_n)$ rendered true.

Overall, the selection operation is a powerful tool for filtering and selecting subsets of data from a relation, based on specified conditions. It is often used in combination with other operations in relational algebra, such as projection and join, to manipulate and query relational data.

### 1.2.3 Cartesian Product

The Cartesian product is a binary operation that combines every row of one table with every row of another table. The resulting table, known as the Cartesian product, has a number of rows equal to the product of the number of rows in the two input tables. This operation is also known as a simple join.

**Definition 1.2.3: Cartesian Product (Join)**

Cartesian Product is denoted with the $\times$ symbol. Let $R_1$ and $R_2$ be arbitray relations with some set of attributes $A_1$ and $A_2$ respectively. Joining tuples $R_1$ and $R_2$ is notated as:

$$R_1 \times R_2$$

The result is a new relation which we will denote $R_\times$. The following defines the new relation:

$$\forall \text{ tuples } t_1 \in R_1. \forall \text{ tuples } t_2 \in R_2. \exists \text{ a tuple } t_\times \in R_\times. t_1|t_2 = t_\times$$

Where $t_1|t_2$ is the tuple such that the attributes of either tuple is concatenated.

This means that each tuple in the first relation is paired with every tuple in the next relation. Therefore, the new table includes all possible combinations of rows from the two input tables. Meaning, given two relations $R_1$ and $R_2$ with cardinalities $n_1$ and $n_2$, the cardinality of $R_1 \times R_2$ is $n_1 \cdot n_2$. The arities of the resulting relation is the sum of the two relations' original arities.

**Example 1.2.1**

Let's say we have two relations, Students and Courses, with the following schemas:

- Students(ID, Name, Age, Gender)
- Courses(Code, Name, Credits)

The cartesian product of these two relations would be a new relation, StudentCourses, with the schema: StudentCourses(ID, Name, Age, Gender, Code, CourseName, Credits)

The relation would have a row for every possible combination of a student and a course, with the student's ID, name, age, and gender paired with the course's code, name, and credits.
Let us say we have an instance of relations Students and Courses, denoted by $S$ and $C$:

| ID | Name | Age | Gender |
|---|---|---|---|
| 1 | Alice | 18 | F |
| 2 | Bob | 19 | M |
| ... | ... | ... | ... |

| Code | CourseName | Credits |
|---|---|---|
| CS101 | Introduction to Computer Science | 4 |
| MATH201 | Calculus I | 4 |
| ENGL101 | Composition | 3 |
| ... | ... | ... |

The result of $S \times C$ would be the following:

| ID | Name | Age | Gender | Code | CourseName | Credits |
|---|---|---|---|---|---|---|
| 1 | Alice | 18 | F | CS101 | Introduction to Computer Science | 4 |
| 1 | Alice | 18 | F | MATH201 | Calculus I | 4 |
| 1 | Alice | 18 | F | ENGL101 | Composition | 3 |
| 2 | Bob | 19 | M | CS101 | Introduction to Computer Science | 4 |
| 2 | Bob | 19 | M | MATH201 | Calculus I | 4 |
| 2 | Bob | 19 | M | ENGL101 | Composition | 3 |
| ... | ... | ... | ... | ... | ... | ... |

This table shows every possible combination of a student and a course, with each row representing a student taking a specific course. Note that the original relation schemas are repeated in the new schema to indicate that each attribute belongs to a specific relation.

**Note:-**

There exists another join operation, named the "Theta join". This operation is denoted by $R_1 \bowtie_{L(A)} R_2$ where $L(A)$ is some logical statement asserting something about the attributes of the relations being joined. This is join is simply shorthand for $\sigma_{L(A)}(R_1 \times R_2)$.

### 1.2.4 Natural Join

A natural join is a way of combining two relations into a single relation. The resulting relation includes only pairs of tuples who values of similar attributes agree. Natural joins are useful when combining two relations which agree on some attribute.

**Definition 1.2.4: Natural Join**

Natural Join is denoted with the $\bowtie$ symbol. Let $R_1$ and $R_2$ be arbitray relations with some set of attributes $A_1$ and $A_2$ respectively. Assume their exists some attribute $a$ that is in both $A_1$ and $A_2$. Natural joining tuples $R_1$ and $R_2$ is notated as:

$$R_1 \bowtie R_2$$

The result is a new relation. Let $A$ be the set of attributes that agree between $A_1$ and $A_2$, meaning $A = A_1 \cap A_2$. The following defines the new relation in terms of select ($\sigma$) and cartesian product ($\times$):

$$R_1 \bowtie R_2 = \sigma_{\forall a \in A. R_1[a]=R_2[a]}(R_1 \times R_2)$$

Meaning the natural join of two relations is simply their cartesian product, where the pairs selected agree on the shared attributes. In the final relation, the shared attributes are express as one, since we already know they agree.

This property of natural join makes it very useful when combining two relations who are defined as agreeing on some attribute (which we will explain in the next chapter on integrity constraints). The following example illustrates this on two relations, $R(A, B, C)$ and $S(C, D, E)$. Note that these relations agree on the attribute 'C'.

**Example 1.2.2**

R

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

$\bowtie$

S

| C | D | E |
|---|---|---|
| 3 | 7 | 8 |
| 6 | 9 | 10 |

$=$

R $\bowtie$ S

| A | B | C | D | E |
|---|---|---|---|---|
| 1 | 2 | 3 | 7 | 8 |
| 4 | 5 | 6 | 9 | 10 |

This example shows two tables R and S, which are joined using the natural join operation. The resulting table contains all combinations of rows from R and S where the values in the C column match.

Important properties of Natural Join are as follows:

- Natural join is associative, *i.e.* for all arbitrary relations $R$,$S$,$T$: $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
- Natural join is commutative, *i.e.* $R \bowtie S = S \bowtie R$
- Natural join can produce unintended results when there are attributes of the same name which are not meant to be related.
- When no attributes match between the two relations, natural join acts the same as cartesian product

### 1.2.5 Assignment and Rename

When operating with many relations and results of operations it is helpful to assign new names to said results. By assigning names to results we are able to recall previous operations with limited redundancy. The assignment operations is used in this case. The operation uses the := operator.

**Definition 1.2.5: Assignment**

For any given relational expression, $E$, with resulting attributes $n$ resulting attributes, an assignment is as follows:

$$R(A_1, \ldots, A_n) := E$$

Where $R$ is the name of the new assigned relation, and $A_1, \ldots, A_n$ are the new (or original) names of the attributes.

In multi-step operations of relational algebra, the assignment operation saves space and time rewriting redundant operations. Renaming attributes is important when joining two instances of the same relation. An important thing to note is that the name of the new assigned relation $R$, must not be one of the relations in the schema. Assignment is **not** equivalent or used to update relations.

It is also possible to assign names to an instance of a relation in an operation. This is different from what the assignment operations does, which is save a result for later use. The rename operator, using the greek symbol $\rho$, rho, is meant to alleviate ambiguity when relational algebra references an attribute that may come from multiple results.

**Example 1.2.3**

Let $R$ be a relation with schema $R(A, B)$.
For this example $R$ will have two tuples $(1, 1), (0, 0)$. Observe the following operation:

$$\sigma_{A=B}(R \times R)$$

If $A$ references the attribute from the left hand side of the product, and $B$ references the right, then the resulting relation is empty. However, if both variables reference the left side then the resulting relation has the maximum cardinality of 4.
This example is simple yet demonstrates the potential ambiguity that may be present during a more complicated combination of operations. For this reason we have the renaming operator

**Definition 1.2.6: Rename**

To rename an instance of a relation, inside of an expression, the $\rho$ symbol can be used with a new name referenced in the subscript. Any following usage of an attribute preceded by the new name and a dot will reference the attributes of the renamed instance. Note this is only for the expression where the relation is renamed. Renaming any relation $R$, with name $T$, is denoted as

$$\rho_T R$$

Specifying attributes, is possible through following $T$ with new attribute names enclosed in parenthesis.

**Example 1.2.4**

Following the previous example, the ambiguity can be removed by specifying a name for each relation.

$$\sigma_{R_1.A=R_2.B}(\rho_{R_1} R \times \rho_{R_2} R) \tag{1.1}$$

$$\sigma_{R_1.A=R_1.B}(\rho_{R_1} R \times \rho_{R_2} R) \tag{1.2}$$

The operation in (1.1) now definitely produces the empty relation, $\{\}$, and the operation in (1.2) produced the every possible combination from $R \times R$

### 1.2.6 Set Operations on Relations

The final important operation on relations directly follows from relations being defined as set adjacent. This means that the main set operations are possible and very applicable to relations.

Set operations such as union, intersection, and difference can be used in relational algebra. Set operations allow you to combine two or more relations to form a new relation, based on the criteria specified in the set operation. A caveat is that the two relations being operated on must agree on the number, name, and order of attributes. This ensures the rules of set relations directly apply.

**Definition 1.2.7: Relational Algebra Set Operations**

Given two relations $R_1$ and $R_2$, where the attributes of $R_1$ and $R_2$ agree in order, name, and number (the schemas are the same), then the following set operations from set theory are applicable:

- Union: $\cup$
- Intersect: $\cap$
- Difference: $-$

**Example 1.2.5**

Suppose we have two relations, R and S, with the following attributes and tuples:

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

| A | B | C |
|---|---|---|
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | 9 |

The operation, $R \cup S$, results in the following table:

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | 9 |

As is obvious from the example, the relations must have equal attributes. Otherwise, a set operation is not applicable. The same can be seen for the similar example of the intersect set operation:

**Example 1.2.6**

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 5 | 6 |

$\cap$

| A | B |
|---|---|
| 3 | 4 |
| 5 | 7 |
| 9 | 8 |

$=$

| A | B |
|---|---|
| 3 | 4 |

In this example, we have two relations, A and B, and the resulting table after performing the intersect operation contains only the tuple (3, 4), since it is the only tuple that appears in both of the original tables.

### 1.2.7 Advanced Relational Algebra Queries

There are many complicated Relational Algebra Queries, but there are two patterns that show up in most difficult query problems. These are the "Extremal" and "Every/All" queries. These problems seem impossible at first but usually have unintuitive ways to answer them. The premise to finding them revolves around finding the negation of the statement and computing the difference. To understand why these problems are unintuitive it is important to understand which queries are impossible.

#### Impossible Queries

Impossible queries often include counting occurrences of some tuple. This is not possible with relational algebra when the counting is undefined. This means it is not possible to find a value corresponding to the maximum amount of tuples with said value. It is also not possible to compare the occurrences between values.

**Note:-**

It is possible, however, to query all values which occur a certain number of times. This is done by joining the relation the desired number of times, asserting that all the keys of each tuple differ, then asserting that the specific value is the same. The only tuples which would pass this would be ones which have the same value, yet are unique tuples. Be careful of pseudo-duplicates, these can be removed by asserting the key increases.

Another impossible query is directly finding something that is true "for all" other relations. This is impossible since our select operation would need to check against every other tuple, requiring an undefined number of joins. This can be queried by instead finding the negation. Negating a "for all" gives us a "there exist" statement. This means, if we can find find the answer from the negation, we can then find all tuples where this is not true, by simply using the set difference operation.

#### Extremal Queries

An extremal query is a type of "for all" query and requires finding a relation where one of the properties of the tuples is that they include the maximum or minimum of some attribute.
The reason this is a "for all" query is that we must select all tuples that have some attribute which is greater than or less than all other tuples. Doing this in a single select, without negating, is impossible as we must compare tuples to an undefined number of other tuples, requiring an undefined number of joins.
Instead we can find all tuples that are not the extremal. This is possible to find, since a property of non-extremal values is that there exists another tuple that must have a more extreme value. Therefore to find all tuples with extremal values, we can find all values without extremal values and then compute the difference of this relation with the total tuples.

**Example 1.2.7**

Let $R(A, B, C)$ be a relation. Suppose we are trying to query for all tuples that have the minimum value for, attribute, $A$. Using the method described above this can be found by first finding the negation and then of the query and computing the difference:

$$R - (\pi_{r_1} \sigma_{r_1.A > r_2.A} (\rho_{r_1} R \times \rho_{r_2} R))$$

The expression in the parenthesis computes all tuples where there was some other tuples that had a value of $A$ less than it. By computing the difference we are left with tuples that did not qualify as having any tuple with a value of $A$ less than it.

## 1.3 Integrity Constraints

Integrity constraints are important in constraining more custom rules onto a relation. They sometimes refer to some reference between relations. This can be simple as an instance's attribute's values needing to be a subset of some other relations. This can become much more complicated through using relational algebra.

### 1.3.1 Referential Integrity Constraints

A referential integrity constraint is a rule that is used to ensure that relationships between tables in a database are maintained. It is a way of ensuring that data in one table references data in another table and that the data in the other table is valid and consistent. For example, if a table contains a foreign key that references a primary key in another table, a referential integrity constraint can be used to ensure that the value in the foreign key column is always a valid primary key value in the other table. This helps to maintain the integrity of the data and ensures that the relationships between the tables are consistent. More rigorously we can define it as follows:

**Definition 1.3.1: Referential Integrity Constraint**

Let $R_1$ and $R_2$ be two unique relations. Let $X$ and $Y$ be arbitrary sets of attributes of equal length in $R_1$ and $R_2$ respectively. The referential integrity constraint between these two relations, denoted $R_1[X] \subseteq R_2[Y]$ implies:

$$\forall \text{ values } v_1 \in R_1[X].\exists \text{ a value } v_2 \in R_2[Y].v_1 = v_2$$

Meaning the set of values in $R_1[X]$ are a subset of the values in $R_2[Y]$

Referential Integrity Constraints are important in defining the relationship between two relations. When $Y$ is a key we call the referential integrity constraint a **Foreign Key Constraint**.

**Note:-**

Referential Integrity Constraints can also be expressed using set operations and a equals assertion:

$$R_1[X] \subseteq R_2[Y] \leftrightarrow R_1[X] - R_2[Y] = \emptyset$$

### 1.3.2 Broad Integrity Constraints

Integrity constraints do not need to always refer to one attribute being a subset to another. They can be much more complex by simply asserting something as always being true. This allows for specifying things such as the symmetrically of relations. An example being a relation, Friend, with schema: Friend(A,B), specifying that "A considers B a friend". Using an integrity constraint we could ensure that if A and B are friends then B and A are friends.

Often, constraints upon a schema are specified in the form of an expression equalling the empty set. This asserts that no tuple, or combination of tuples, in these relations render the assertion false.

**Example 1.3.1**

Let Friend be a relation with schema Friend(A,B). We define this relation by implying that A considers B a friend. If we want to ensure that every friendship is mutual we would have to assert that if (A,B) are in Friend, then (B,A) is also in Friend. The following integrity constraint asserts this:

$$\pi_{B,A}Friend - Friend = \emptyset$$

This asserts (B,A) exists in the relation as if not it would appear in the final result.

# Chapter 2

# Design Theory

Design Theory is simply just different possible ideas on making schemas more digestible and efficient. The way this is achieved is primarily through systematically changing schemas such that they fit different defined forms. Many of these forms are defined as normal forms. Where normal form just means it is some sort of agreed upon standard. The process of changing a schema such that it fits some 'normal form' is called **normalization**. This design theory considers what is "better."

## 2.1 Functional Dependency Theory

Often in schemas we observe patterns or instances where an attribute is dependant on another. These patterns often show themselves in redundancy. Redundant attributes still include information but it is repeated. Observe the following example of redundancy:

**Example 2.1.1**

Let Person(ID, Name, Age, Gender, Company, Company Address) be a relation which tracks information on someone. Note the following instance of this relation:

| ID | Name | Age | Gender | Company | Company Address |
|---|---|---|---|---|---|
| 1 | John | 32 | Male | Acme Inc. | 123 Main St |
| 2 | Mary | 28 | Female | XYZ Corp. | 456 Park Ave |
| 3 | Steven | 44 | Male | Acme Inc. | 123 Main St |

Observe the company address attribute, it is clearly dependant on the company. This means that we are storing redundant information within the table.

There are many ways to fix redundant information and that will be addressed in a following section. However, it is obvious that the information in Company Address is dependant on the information in Company. This can be note using a functional dependency.

**Definition 2.1.1: Functional Dependency (FD)**

Let $R$ be a relation with subsets of arbitrary subsets of attributes $X$ and $Y$, such that $X \cap Y = \emptyset$. The statement, $Y$ is functionally dependant on $X$ is represented by the following notation:

$$X \to Y$$

This asserts that if there are two tuples which agree on the values of $X$ then they must agree on the values of $Y$. This can be represented with the following statement. Let $t_1$ and $t_2$ be arbitrary tuples of $R$.

$$(\forall x \in X . t_1[x] = t_2[x]) \implies (\forall y \in Y . t_1[Y] = t_2[Y])$$

**Example 2.1.2**

Following the previous example and our definition of Functional Dependancy, we can express the relationship between the Company and Company Address attributes with the following functional dependency:

$$\text{Company} \rightarrow \text{CompanyAddress}$$

The normalization of schemas and relations generally follow from manipulating or interpreting information from the functional dependencies; therefore FDs are an important concept to understand. There are several important properties of functional dependencies that should not be ignored. These properties describe equivalent sets of functional dependencies. A simple concept is that functional dependencies can be split by their right hand side but not their left. This follows from the definition.

**Note:-**

Functional dependencies are closely related to keys in that all the other attributes in a relation are dependant on the key. This follows from there only being allowed one tuple with a certain key. This allows us to apply the functional dependency definition.

### 2.1.1 Principles of Functional Dependencies

The principles of FDs allow use to break them down and manipulate them in order to achieve some normal form described in the future.

**Definition 2.1.2: The Six Properties of Functional Dependencies**

Let $R$ be a relation with the following arbitrary sets of attributes: $W$, $X$, $Y$, and $Z$.

1. Reflexivity: If X is a set of attributes, then X $\rightarrow$ X.
2. Augmentation: If X $\rightarrow$ Y, then XZ $\rightarrow$ YZ.
3. Transitivity: If X $\rightarrow$ Y and Y $\rightarrow$ Z, then X $\rightarrow$ Z.
4. Union: If X $\rightarrow$ Y and X $\rightarrow$ Z, then X $\rightarrow$ YZ.
5. Decomposition: If X $\rightarrow$ YZ, then X $\rightarrow$ Y and X $\rightarrow$ Z.
6. Pseudotransitivity: If X $\rightarrow$ Y and WY $\rightarrow$ Z, then WX $\rightarrow$ Z.

**Example 2.1.3** ( Functional Dependency Principles )

Suppose we have the following relation with attributes A, B, C, D, E, and F:

$$R = A, B, C, D, E, F$$
$$F = (a, b, c, d, e, f) \mid a, b, c, d, e, f \in R$$

We can use the principles of functional dependencies to determine the functional dependencies in this relation. For example, suppose we know that A $\rightarrow$ B and B $\rightarrow$ C. Using the transitivity principle, we can conclude that A $\rightarrow$ C.
Next suppose we know that A $\rightarrow$ B and C $\rightarrow$ D, we can use the union principle to conclude that A $\rightarrow$ BD.

### 2.1.2 Attribute Closure

The closure of an attribute or set of attributes indicates the largest set of attributes dependant on said attributes or set of attributes. In the future, closure attributes will be used to determine the minimal set of attributes that

are required to determine the values of all other attributes in a relation.
Closure attributes are useful in relational algebra because it allows us to reason about the relationships between different attributes in a relation, and to make inferences about what can be derived from a given attribute.

**Definition 2.1.3: Closure**

Let $R$ be a relation with an arbirtary set of attributes $X$. The closure of $X$ is denoted as $X^+$. The following expresses the closure of $X$:

$$X^+ = Y$$

$Y$ is the largest set of attributes, such that $X \rightarrow Y$ is true.

**Example 2.1.4**

For example, let $R(A, B, C)$ be a relation. If we have the attributes A and B, where $A \rightarrow B$, and we know that we can derive attribute C from A and B, (e.g. $AB \rightarrow C$), then using the property of transitivity we can say that the closure of attribute A includes attribute C, as follows:

$$A^+ = A, B, C$$

This is because $A$ refers to the set of all attributes that can be derived from the original attribute using a series of properties. Notice how $A$ is present in the closure of A. This follows from the properties of reflexivity.

### 2.1.3 Projection of Functional Dependencies

Sometimes we decompose relations. This is often used to combat redundancy or to normalize. In these cases it is important to be able to maintain the relevancy of functional dependencies on these new relations. Projecting functional dependencies do not necessarily maintain the functional dependency in the entire schema. This depends on how we normalize, which is problem that will be addressed in the future. What projecting functional dependencies does do is indicate what functional dependencies must hold in the new relation.
The projection algorithm is as follows, where S is a set of FDs and L is a set of attributes:

**Algorithm 1:** Project

**Input:** (S,L) S is a set of FDs and L is a set of attributes
**Output:** T, the new of FDs applicable to the decompose relation L

```
1 T ← {};
2 foreach subset X of L do
3 |   Compute X+;
4 |   foreach attribute A in X+ do
5 |   |   if A is in L then
6 |   |   |   add "X → A" to T;
7 |   end
8 end
9 return T;
```

Speed ups to this algorithm is as follows:

- Functional Dependencies that follow from the law of reflexivity are trivial. This means there is no need to add $X \rightarrow A$ if $A \subseteq X$. This is trivial as it is always is inferable by reflexivity
- Attribute sets such as $\emptyset$ and $L$ itself are also trivial and do not yield anything remarkable.
- By augmentation, if we find $X^+ = L$ then, on line 2, we can ignore any superset of $X$. This is a big time saver.

### 2.1.4 Minimal Basis

From the previous sections it is obvious that there are many equivalent sets of functional dependencies given some FDs. Often we want to find specific equivalent sets of FDs. A common equivalent set is called the minimal basis of a set of FDs.

**Definition 2.1.4: Minimal Basis**

Given some set of functional dependencies, $S$. The minimal basis is an equivalent set of functional dependencies where:

- There are no redundant Functional Dependencies
- There are no functional dependencies with unnecessary attributes on the left hand side

Redundant functional dependencies are any dependencies that can be inferred from the law of transitivity

**Example 2.1.5**

For example, suppose we have the following set of functional dependencies: $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$ Here, the functional dependencies $A \rightarrow C$ is implied by $A \rightarrow B, B \rightarrow C$ by the law of transitivity. So the set of functional dependencies $\{A \rightarrow B, B \rightarrow C\}$ is a minimal basis for the original set.

The process of finding a minimal basis for some arbitrary set of functional dependencies, $S$, is as follows:

1. Split the right hand side of each functional dependency
2. For each functional dependency, $X \rightarrow Y$, where $|X| \geqslant 2$, attempt to remove attributes from $X$ to get a functional dependency which still follows from $S$. This can be done by finding all the closures of subsets of $X$. If any of the attributes of $X$ not included in the subset is present in the closure, we know this attribute is extraneous and can be removed.
3. For each remaining functional dependency, remove any that are inferrable from transitivity, without removing that inferrability. The caveat comes from when a functional dependency is inferrable via transitivity yet is a part of a loop, meaning that it's transitivity is inferred from itself. Meaning removing it would no longer be the same set of FDs.

**Note:-**

There are often multiple possible results and after removing redundant functional dependencies those FDs must not be considered in subsequent closures.

## 2.2 Normalization

Normalization in relational algebra is the process of organizing a relation into smaller, more manageable relations that are in a certain normal form. This is important because it helps to reduce redundancy and improve the overall design of the database.
In a relational database, normalization is the process of organizing data into smaller, more manageable relations that are in a certain normal form. This helps to ensure that the data is stored in an efficient and organized manner, and reduces the risk of inconsistencies and other problems.
Without normalization, a relational database can become cluttered and difficult to manage, and may contain redundant data that can lead to inconsistencies and other problems. Normalization helps to prevent these issues by organizing the data into smaller, more manageable relations that are in a certain normal form.
Overall, normalization is an important concept in relational databases because it helps to improve the design and organization of the database, and makes it easier to manage and maintain the data.

### 2.2.1 Boyce-Codd Normal Form

The first form and method of normalization we will look at is called the Boyce-Codd Normal Form. Boyce-Codd normal form describes a fix to the redundancy. Redundancy is negative in relational databases as we require extra space and clutter our tuples. An example of redundancy was shown at the beginning of this chapter.

**Definition 2.2.1: Boyce-Codd Normal Form (BCNF)**

A relation being in **Boyce-Codd Normal Form** is equivalent to every nontrivial functional dependency's left hand side being a superkey for the relation. Formally this expressed as the following:
Let $R$ be an arbitrary relation, with a set of functional dependencies, $S$. For any functional dependency, $f$, let left hand side be denoted by $f[LHS]$. $R$ being in **Boyce-Codd Normal Form** is equivalent to:

$$\forall f \in S.f[LHS] = R$$

Boyce-Codd Normal Form (BCNF) ensures that there is no redundancy in a relational database because it requires that every determinant in the relation is a candidate key. In other words, every non-trivial functional dependency in the relation must be a function of a candidate key.

This means that any data that is dependent on a candidate key can be uniquely determined by that key, which helps to ensure the integrity of the data and prevents certain types of redundancy. For example, if a relation is in BCNF, then we can be sure that there is no redundant data in the relation, and that any data that is dependent on a candidate key can be uniquely determined by that key.

Unless a relation is already in Boyce-Codd Normal Form, we must decompose said relation to ensure BCNF. The algorithm for decomposition is as follows:

**Algorithm 2:** BCNF_Decomposition

**Input:** (R,F) R is a relation and F is R's functional dependencies
**Output:** The new set of relations and functional dependencies following BCNF

**if** *an FD* $X \rightarrow Y in F$ *violates BCNF* **then**
  Compute $X^+$;
  $R_1 \leftarrow X^+$ ;
  $R_2 \leftarrow R - (X^+ - X)$ ;
  Project the FD's from $F$ onto $R_1$ and $R_2$ to get $F_1$ and $F_2$ ;
  **return** BCNF_Decomposition($R_1$, $F_1$) ∪ BCNF_Decomposition($R_2$, $F_2$);
**else**
  **return** $\{(R, F)\}$
**end**

Overall, BCNF is a strong normal form that helps to ensure that there is no redundancy in a relational

database. It requires that every non-trivial functional dependency in the relation is a function of a candidate key, which helps to prevent certain types of redundancy and ensures the integrity of the data.

### 2.2.2 Lossless Join and Dependency Preservation

An important point about decomposition is that we may want to be able to preserve the properties of the original relation. There are two main points of decomposition. The **Lossless Join** and **Dependency Preservation**. Ensuring these will mean that we will be able to reconstruct the original schema by natural joining the decomposed relations. Lossless Join ensures that no new tuple will be included when recomposed. Dependency Preservation ensures that the functional dependencies will still be applicable after recomposing.

#### Lossless Join

**Definition 2.2.2: Lossless Join**

Let $R$ be a relation which was decomposed into $r_1, \ldots, r_n$. The decomposition has a lossless join when:

$$R = r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

This means that we can ensure we will have all the tuples from the original composition, without any less or more.

**Note:-**

A lossy join is what usually occurs when we decompose and do not ensure a lossless join. In this case $R \subset r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$. This means the decomposition has introduced spurious tuples.

It is also important to note that you are not really losing anything specific in the join during a lossy join, instead you are gaining extra tuples in your resulting join that should not be there. On second thought, maybe you are losing your sanity? ¯_(ツ)_/¯

The BCNF property from early does not guarantee a lossless join. Which means the algorithm we gave earlier does not necessarily return a usable relation. Therefore, BCNF is usually just used as a property to ensure there is no redundancy. There is a future test to ensure a lossless join.

#### Dependancy Preservation

Dependency Preservation ensures that all the original dependencies described in the schema, pre decomposition, are maintained. There are some decompositions which do not preserve dependencies in these case it is possible to create an instance that satisfies all the FDs in the final schema, yet violate on the originals FDs. Note the following example of not maintaining functional dependencies.

**Example 2.2.1**

Let $R$ be a relation that has the functional dependency $A \rightarrow BD$. Observe the following decomposition:

$$R = \{A, B, C, D\}$$
$$R1 = \{A, B\}$$
$$R2 = \{C, D\}$$

In this case, the functional dependency $A \rightarrow B$ is preserved in $R1$, but $A \rightarrow D$ is lost in $R2$. This is because $R2$ does not contain the attribute $A$, and therefore cannot enforce the functional dependency.

In general, excessive decomposition of a relation can lead to the loss of functional dependencies and can result in a loss of data integrity. It is important to carefully design the schema of a database to ensure that functional dependencies are preserved.

### 2.2.3 Keys in Terms of Functional Dependencies

The final important point before understanding further normalization, specifically 3NF, is to learn keys in the context of functional dependencies.

**Definition 2.2.3: Key (In Terms of Functional Dependencies)**

Let $R$ be a relation. Let $A$ be an arbitary subset of $R$'s attributes. $A$ being a key is equivalent to saying that:

$$(A^+ = R) \wedge (\nexists \text{ a subset of } A, \text{ denoted } B \ s.t. B^+ = R)$$

The process to finding keys is not very simple and usually requires analyzing and understanding the relationship between all the functional dependencies. However, there is a simple method to understand whether an attribute belongs in every key, may belong in a key or does not belong in any keys. The method is as follows:

**Belongs in Every Key:** This occurs when an attribute is not in any functional dependency or only on the left hand side. This means that for the cover to include the whole relation we must include these attributes, and hence they are part of every key.

**Does not Belong in Any Key:** This occurs when an attribute is only on the right hand side of dependencies. This follows from requiring the cover of the key to include every attribute. Suppose, for sake of contradiction, we have a key that includes an attribute only in the right hand side of functional dependencies. Since this key is a cover for $R$, the parent of the attribute must be included in the cover. As it is the parent of the attribute, we can remove the attribute from the key and the cover will still be the entire relation. Therefore this key is not minimal and not a key. Hence by contradiction we have shown that attributes only in the right hand side of functional dependencies will not be in any key.

**May Belong in a Key** This is for any attribute that does not fit any of the terms above. This means it is any attribute that is both in the left and right hand sides of functional dependencies.

Afterwards the process simply requires trying every possible combination of the attributes which may belong in a key and those which belong in every key. First start with the fewest possible number of attributes to add. Whenever a key is found, disregard any superset of the key.

### 2.2.4 3rd Normal Form

3rd normal form ensures that the decomposition has a **lossless join** and **dependency preservation**. This is useful in that our new decomposition reduces redundancy and provides clarity to how the schema relates, without introducing spurious tuples or allowing dependencies to be broken. However, redundancy can still be present in a relation that is in 3rd normal form.

**Definition 2.2.4: 3rd Normal Form (3NF)**

Let $R$ be a relation with functional dependencies, $S$. We say an attribute is **prime** if it is a member of any key.
$R$ is in 3rd Normal Form is equivalent to, for all functional dependencies, $f \in S$, at least one of the following is true:

- $f[LHS]^+ = R$
- $f[RHS]$ is prime

Where $f[LHS]$ and $f[RHS]$ are the left and right hand side of $f$, respectively.

**Note:-**

Note how we treat $f[RHS]$ as a singular attribute. This means we consider every attribute in the right hand side, if there are multiple attributes. This is because, by the 5th property of functional dependencies we have that we decomposed FDs are equivalent.

Consequently of the definition of 3rd Normal Form, if we have a functional dependency, $X \to Y$, where $X$ is not a superkey and $Y$ is not prime, then that FD is said to 'violate' 3NF. The process of synthesizing a relation in 3rd normal form, from some functional dependencies, $F$, and a relation $L$ is as follows:

1. Construct a minimal basis $M$ for $F$. (See the section on minimal basis for details)
2. For each FD $X \to Y$ in $M$, define a new relation with schema $X \cup Y$
3. If no relation exists that is a superkey for $L$, add a relation with a schema that is some key. (This step is necessary when there is an attribute not present in any FD)

The algorithm can be formalized into pseudo-code as:

**Algorithm 3:** 3NF_Synthesis

**Input:** $(L, F)$ $L$ is a relation and $F$ is $L$'s functional dependencies
**Output:** The new set of relations and functional dependencies following 3NF

```
1 M → a minimal basis for F;
2 S → {} ;                                   // Set of Relations to be returned
3 foreach FD X → Y in M do
4 |  S → S ∪ (X ∪ Y) ;                      // Make a relation containing the attributes of X and Y
5 end
6 if no relation is a superkey for L then
7 |  K → a key of L;
8 |  S → S ∪ (L);
9 return S
```

**Note:-**

CSC343 notes define BCNF and 3NF as "good", whatever that means ¯_(ツ)_/¯

## 2.3 Testing Relational Properties

This section includes techniques on validating relational properties, mainly redundancy, lossless join, and functional preservation. This is important to consider since not all normalization forms from the previous section guarantee these properties to hold. For example, a lossless join is guaranteed by Boyce-Codd Normal Decomposition but not Boyce-Codd Normal Form itself, so it is important to validate this.

### 2.3.1 Validating Lossless Join

The definition of Lossless Join makes it difficult to ensure that the join will be lossless for every possible set of tuples. To ensure this we can instead use a test called the **Chase Test**. First we should understand the intuition behind the Chase Test.

**Proposition 2.3.1** Chase Test Intuition

Suppose that we joined a decomposition. A tuple $t$ appears in the resulted join. This means $t$ is the join of projections of some tuples originally from $R$. One tuple for each of the relations in the decomposition. What we must ensure to test there is a lossless join is to ensure that $t$ is always a join of the same tuple. This could be done by using the functional dependencies.

The Chase Test is described as follows:

1. First create a table, where each column is a different attribute and each row is a different relation from the decomposition. Indicate the cells where the column's attribute is present in that row's relation.
2. For any functional dependency, if there are two rows whose columns are indicated for every attribute in the right hand side, and one of these rows indicates a value from the right hand side, then these two rows must agree on that value from the right hand side when joined. Therefore we indicate the empty cell corresponding to attribute in the right hand side. Repeat this step until no longer possible, do not forget to consider newly indicated cells.
3. If there is a row where every column is indicated, the decomposition is lossless, otherwise it is lossy.

This explanation is unfortunately abstract and hard to conceptualize so an example is very helpful.

**Example 2.3.1**

Let $R(A, B, C)$ be a relation with functional dependencies, $F = \{(A \rightarrow B), (B \rightarrow C)\}$.
$R$ has been decomposed into the following relations, $R_1(A, B)$ and $R_2(B, C)$.
To find out if the decomposition is lossless we can use the Chase Test. To do this we complete the first step which initializes the grid:

| - | $A$ | $B$ | $C$ |
|---|---|---|---|
| $R_1(A, B)$ | × | × | |
| $R_2(B, C)$ | | × | × |

Since we have the functional dependency, $(B \rightarrow C)$, and both relations have $B$ indicated, we can assume that when they are joined they will agree on $C$. This means we will now indicate $C$ in the first row:

| - | $A$ | $B$ | $C$ |
|---|---|---|---|
| $R_1(A, B)$ | × | × | × |
| $R_2(B, C)$ | | × | × |

Since there is a row where every column is indicated, the first row, we can assume that there is a lossless join, and our decomposition of $R$ is lossless.

### 2.3.2 Checking Redundancy

Redundancy is unfortunately ill-defined, meaning it is difficult to fully understand how to check for it. Thankfully, relations that fit Boyce-Codd Normal Form are defined as not being redundant. This means that if every relation in a decomposition is in Boyce-Codd Normal Form, it must not be redundant.

**Note:-**

Most relations constructed by the 3NF synthesis algorithm follow Boyce-Codd Normal Form. There may be a proof that it is always the case, but it does seem possible that loops would create issues, such as in cases where a relation is discarded as it is the subset of another.

### 2.3.3 Validating Functional Dependency Preservation

A simple way to validate that functional dependencies are preserved between the original relation and the decomposed relation is by first projecting all the functional dependencies of the original relation on the new relation. The obvious way to check preservation is to compare the the FDs of the new and original schema and see if there are any missing in the new relation. This can leave room for errors as it is not always easy to track if relations are the same. A more methodical method is to calculate the minimal basis of each set of FDs, if they are the same the functional dependencies are preserved.