Chapter 7: Normalized Database Design Part 1

Abu Raihan Mostofa Kamal

Professor, CSE Department
Islamic University of Technology (IUT)

February 18, 2025

[1] This is based on Textbook, its companion slide and other sources

Chapter Outline

Good Relational Database Design Aspects

The **goal** of relational database design is to generate a set of relation schemas that will meet the following 2 goals:

- allows us to store information **without unnecessary redundancy**
- but allows us to **retrieve information easily**
- Hence, we need a standard method to evaluate a design called Normal Forms

Good Relational Database Design Aspects

The **goal** of relational database design is to generate a set of relation schemas that will meet the following 2 goals:

- allows us to store information **without unnecessary redundancy**
- but allows us to **retrieve information easily**
- Hence, we need a standard method to evaluate a design called Normal Forms

Good Relational Database Design Aspects

The **goal** of relational database design is to generate a set of relation schemas that will meet the following 2 goals:

- allows us to store information **without unnecessary redundancy**
- but allows us to retrieve **retrieve information easily**
- Hence, we need a standard method to evaluate a design called Normal Forms

Good Relational Database Design Aspects

The **goal** of relational database design is to generate a set of relation schemas that will meet the following 2 goals:

- allows us to store information **without unnecessary redundancy**
- but allows us to **retrieve information easily**
- Hence, we need a standard method to evaluate a design called Normal Forms

Design Alternatives: Smaller to Larger Schema option

Consider the following 2 schemas:

1. department(dept name, building, budget)
2. instructor(ID , name, **dept name**, salary)

What is the main problem with this design?

It involves Natural Joins to retrieve necessary information which is very expensive(!!) So lets **merge them in One Schema**.

The result is:

inst_dept (ID , name, salary, dept name, building, budget)

Design Alternatives: Smaller to Larger Schema option

Consider the following 2 schemas:

1. department(dept name, building, budget)
2. instructor(ID , name, **dept name**, salary)

What is the main problem with this design?

It involves Natural Joins to retrieve necessary information which is very expensive(!!) So lets **merge them in One Schema**.

The result is:

inst_dept (ID , name, salary, dept name, building, budget)

Design Alternatives: Smaller to Larger Schema option

Consider the following 2 schemas:

1. department(<u>dept name</u>, building, budget)
2. instructor(<u>ID</u> , name, **dept name**, salary)

What is the main problem with this design?

It involves Natural Joins to retrieve necessary information which is very expensive(!!) So lets **merge them in One Schema**.

The result is:

inst_dept (ID , name, salary, dept name, building, budget)

Design Alternatives: Smaller to Larger Schema option

Consider the following 2 schemas:

1. department(dept name, building, budget)
2. instructor(ID , name, **dept name**, salary)

What is the main problem with this design?

It involves Natural Joins to retrieve necessary information which is very expensive(!!) So lets **merge them in One Schema**.

The result is:

inst_dept (ID , name, salary, dept name, building, budget)

Design Alternatives: Smaller to Larger Schema option

Consider the following 2 schemas:

1. department(dept name, building, budget)
2. instructor(ID , name, **dept name**, salary)

What is the main problem with this design?

It involves Natural Joins to retrieve necessary information which is very expensive(!!) So lets **merge them in One Schema**.

The result is:

inst_dept (ID , name, salary, dept name, building, budget)

Lets watch the data in the larger schema:

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

3 Problems:

1. **Redundancy**: Department info is repeated.
2. **Inconsistency**: Update of department info should be propagated properly.
3. **Introduces Bad Business Logic**: You can not enter data for a new department unless there is a teacher of that department.

Lets watch the data in the larger schema:

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

3 Problems:

1. **Redundancy**: Department info is repeated.
2. **Inconsistency**: Update of department info should be propagated properly.
3. **Introduces Bad Business Logic**: You can not enter data for a new department unless there is a teacher of that department.

Lets watch the data in the larger schema:

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

3 Problems:

1. Redundancy: Department info is repeated.
2. Inconsistency: Update of department info should be propagated properly.
3. Introduces Bad Business Logic: You can not enter data for a new department unless there is a teacher of that department.

Lets watch the data in the larger schema:

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

3 Problems:

1. **Redundancy**: Department info is repeated.
2. **Inconsistency**: Update of department info should be propagated properly.
3. **Introduces Bad Business Logic**: You can not enter data for a new department unless there is a teacher of that department.

Lets watch the data in the larger schema:

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

3 Problems:

1. **Redundancy**: Department info is repeated.
2. **Inconsistency**: Update of department info should be propagated properly.
3. **Introduces Bad Business Logic**: You can not enter data for a new department unless there is a teacher of that department.

Lets watch the data in the larger schema:

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

3 Problems:

1. **Redundancy**: Department info is repeated.
2. **Inconsistency**: Update of department info should be propagated properly.
3. **Introduces Bad Business Logic**: You can not enter data for a new department unless there is a teacher of that department.

Design Alternatives: Smaller Schema option

- Suppose we start with Larger Schema :
  inst_dept (ID , name, salary, dept name, building, budget)
- Now **how do we know** that it requires splitting and contains repetition?
- **Finding repetition** is easy here in particular, but it is **very hard** in real-life database where number of records is very very large (i.e. in millions).
- How do we know that **the data seen is repetition or just a co-incidence?**
  - How would we know that in our university **each department** (identified by its department name) must reside in a **single building and must have a single budget amount**?
  - May be we have 3 separate Computer Science departments residing in a single building, same budget amount is **just a co-incidence**.

Design Alternatives: Smaller Schema option

- Suppose we start with Larger Schema :
  inst_dept (ID , name, salary, dept name, building, budget)
- Now **how do we know** that it requires splitting and contains repetition?
- **Finding repetition** is easy here in particular, but it is **very hard** in real-life database where number of records is very very large (i.e. in millions).
- How do we know that **the data seen is repetition or just a co-incidence?**
  - How would we know that in our university **each department** (identified by its department name) must reside in a **single building and must have a single budget amount?**
  - May be we have 3 separate Computer Science departments residing in a single building, same budget amount is **just a co-incidence.**

Design Alternatives: Smaller Schema option

- Suppose we start with Larger Schema :
  inst_dept (ID , name, salary, dept name, building, budget)
- Now **how do we know** that it requires splitting and contains repetition?
- **Finding repetition** is easy here in particular, but it is **very hard** in real-life database where number of records is very very large (i.e. in millions).
- How do we know that **the data seen is repetition or just a co-incidence?**
  - How would we know that in our university **each department** (identified by its department name) must reside in a **single building and must have a single budget amount?**
  - May be we have 3 separate Computer Science departments residing in a single building, same budget amount is **just a co-incidence.**

Design Alternatives: Smaller Schema option

- Suppose we start with Larger Schema :
  inst_dept (ID , name, salary, dept name, building, budget)
- Now **how do we know** that it requires splitting and contains repetition?
- **Finding repetition** is easy here in particular, but it is **very hard** in real-life database where number of records is very very large (i.e. in millions).
- How do we know that **the data seen is repetition or just a co-incidence?**
  - How would we know that in our university **each department** (identified by its department name) must reside in a **single building and must have a single budget amount**?
  - May be we have 3 separate Computer Science departments residing in a single building, same budget amount is **just a co-incidence**.

Design Alternatives: Smaller Schema option

- Suppose we start with Larger Schema :
  inst_dept (ID , name, salary, dept name, building, budget)
- Now **how do we know** that it requires splitting and contains repetition?
- **Finding repetition** is easy here in particular, but it is **very hard** in real-life database where number of records is very very large (i.e. in millions).
- How do we know that **the data seen is repetition or just a co-incidence?**
  - How would we know that in our university **each department** (identified by its department name) must reside in a **single building and must have a single budget amount**?
  - May be we have 3 separate Computer Science departments residing in a single building, same budget amount is **just a co-incidence**.

Design Alternatives: Smaller Schema option

- Suppose we start with Larger Schema :
  inst_dept (ID , name, salary, dept name, building, budget)
- Now **how do we know** that it requires splitting and contains repetition?
- **Finding repetition** is easy here in particular, but it is **very hard** in real-life database where number of records is very very large (i.e. in millions).
- How do we know that **the data seen is repetition or just a co-incidence?**
  - How would we know that in our university **each department** (identified by its department name) must reside in a **single building and must have a single budget amount**?
  - May be we have 3 separate Computer Science departments residing in a single building, same budget amount is **just a co-incidence**.

Smaller Schema option (Cont.)

- We need some formal method to discover that the university requires that **every department (identified by its department name) must have only one building and one budget value.**
- We need to allow the database designer **to specify rules** such as:
  - Each dept must have only one budget.
  - Each dept must reside in one building.
- we need to write a rule that says **if there were a schema (dept name, budget), then dept name is able to serve as the primary key.**
- This form of rule is known as **Functional Dependency** as expressed:
  $dept\_name \longrightarrow$ budget

  (It will be discussed in great details very soon.)

Smaller Schema option (Cont.)

- We need some formal method to discover that the university requires that **every department (identified by its department name) must have only one building and one budget value.**
- We need to allow the database designer **to specify rules** such as:
  - Each dept must have only one budget.
  - Each dept must reside in one building.
- we need to write a rule that says **if there were a schema (dept name, budget), then dept name is able to serve as the primary key.**
- This form of rule is known as **Functional Dependency** as expressed: $dept\_name \longrightarrow$ budget

  (It will be discussed in great details very soon.)

Smaller Schema option (Cont.)

- We need some formal method to discover that the university requires that **every department (identified by its department name) must have only one building and one budget value.**
- We need to allow the database designer **to specify rules** such as:
  - Each dept must have only one budget.
  - Each dept must reside in one building.
- we need to write a rule that says **if there were a schema (dept name, budget), then dept name is able to serve as the primary key.**
- This form of rule is known as **Functional Dependency** as expressed:

  $dept\_name \longrightarrow$ budget

  (It will be discussed in great details very soon.)

Smaller Schema option (Cont.)

- We need some formal method to discover that the university requires that **every department (identified by its department name) must have only one building and one budget value.**
- We need to allow the database designer **to specify rules** such as:
  - Each dept must have only one budget.
  - Each dept must reside in one building.
- we need to write a rule that says **if there were a schema (dept name, budget), then dept name is able to serve as the primary key.**
- This form of rule is known as **Functional Dependency** as expressed:
  $dept\_name \longrightarrow$ budget

  (It will be discussed in great details very soon.)

Smaller Schema option (Cont.)

- We need some formal method to discover that the university requires that **every department (identified by its department name) must have only one building and one budget value.**
- We need to allow the database designer **to specify rules** such as:
  - Each dept must have only one budget.
  - Each dept must reside in one building.
- we need to write a rule that says **if there were a schema (dept name, budget), then dept name is able to serve as the primary key.**
- This form of rule is known as **Functional Dependency** as expressed:
  $dept\_name \longrightarrow$ budget

  (It will be discussed in great details very soon.)

Smaller Schema option (Cont.)

- We need some formal method to discover that the university requires that **every department (identified by its department name) must have only one building and one budget value.**
- We need to allow the database designer **to specify rules** such as:
  - Each dept must have only one budget.
  - Each dept must reside in one building.
- we need to write a rule that says **if there were a schema (dept name, budget), then dept name is able to serve as the primary key.**
- This form of rule is known as **Functional Dependency** as expressed:
  $dept\_name \longrightarrow \text{budget}$

  (It will be discussed in great details very soon.)

Schema Decomposition

- It is **not hard** to see that the right way to decompose inst_dept is into schemas instructor and department as in the original design. (It is easy since the scope is very small and the required rule is almost intuitive)
- Finding the **right decomposition** is much **harder** for schemas **with a large number** of attributes and several functional dependencies.
- Hence, we need some **formal methodology** for it. (Normal Forms)

Schema Decomposition

- It is **not hard** to see that the right way to decompose inst_dept is into schemas instructor and department as in the original design. (It is easy since the scope is very small and the required rule is almost intuitive)
- Finding the **right decomposition** is much **harder** for schemas **with a large number** of attributes and several functional dependencies.
- Hence, we need some **formal methodology** for it. (Normal Forms)

Schema Decomposition

- It is **not hard** to see that the right way to decompose inst_dept is into schemas instructor and department as in the original design. (It is easy since the scope is very small and the required rule is almost intuitive)
- Finding the **right decomposition** is much **harder** for schemas **with a large number** of attributes and several functional dependencies.
- Hence, we need some **formal methodology** for it. (Normal Forms)

Schema Decomposition: The Bad One

- We start with a single schema employee (ID , name, street, city, salary)
- Now lets decompose it into 2 schemas:

  employee1 (ID , name)

  employee2 (name, street, city, salary)

- Main problem with this decomposition comes from the fact that **two employees have same name** and **Name is the joining attribute**.

Schema Decomposition: The Bad One

- We start with a single schema employee (ID , name, street, city, salary)
- Now lets decompose it into 2 schemas:

employee1 (ID , name)

employee2 (name, street, city, salary)

- Main problem with this decomposition comes from the fact that **two employees have same name** and **Name is the joining attribute**.

Schema Decomposition: The Bad One

- We start with a single schema employee (ID , name, street, city, salary)
- Now lets decompose it into 2 schemas:

  employee1 (ID , name)

  employee2 (name, street, city, salary)
- Main problem with this decomposition comes from the fact that **two employees have same name** and **Name is the joining attribute**.

Schema Decomposition: The Bad One (Cont.)

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

employee

ID	name
⋮	
57766	Kim
98776	Kim
⋮	

name	street	city	salary
⋮			
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000
⋮			

natural join

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

Here observe:

- We **fail to produce the original records** by natural join.
- It results in **incorrect** records.
- Such problematic decomposition is called **lossy decompositions**.
- So, we will look for correct decomposition termed as **lossless decompositions**.

Schema Decomposition: The Bad One (Cont.)

ID	*name*	*street*	*city*	*salary*
⋮				
57766	*Kim*	*Main*	*Perryridge*	*75000*
98776	*Kim*	*North*	*Hampton*	*67000*
⋮				

employee

ID	*name*
⋮	
57766	*Kim*
98776	*Kim*
⋮	

name	*street*	*city*	*salary*
⋮			
Kim	*Main*	*Perryridge*	*75000*
Kim	*North*	*Hampton*	*67000*
⋮			

natural join

ID	*name*	*street*	*city*	*salary*
⋮				
57766	*Kim*	*Main*	*Perryridge*	*75000*
57766	*Kim*	*North*	*Hampton*	*67000*
98776	*Kim*	*Main*	*Perryridge*	*75000*
98776	*Kim*	*North*	*Hampton*	*67000*
⋮				

Here observe:

- We **fail to produce the original records** by natural join.
- It results in **incorrect** records.
- Such problematic decomposition is called **lossy decompositions**.
- So, we will look for correct decomposition termed as **lossless decompositions**.

Schema Decomposition: The Bad One (Cont.)

ID	*name*	*street*	*city*	*salary*
⋮				
57766	*Kim*	*Main*	*Perryridge*	*75000*
98776	*Kim*	*North*	*Hampton*	*67000*
⋮				

employee

ID	*name*
⋮	
57766	*Kim*
98776	*Kim*
⋮	

name	*street*	*city*	*salary*
⋮			
Kim	*Main*	*Perryridge*	*75000*
Kim	*North*	*Hampton*	*67000*
⋮			

natural join

ID	*name*	*street*	*city*	*salary*
⋮				
57766	*Kim*	*Main*	*Perryridge*	*75000*
57766	*Kim*	*North*	*Hampton*	*67000*
98776	*Kim*	*Main*	*Perryridge*	*75000*
98776	*Kim*	*North*	*Hampton*	*67000*
⋮				

Here observe:

- We **fail to produce the original records** by natural join.
- It results in **incorrect** records.
- Such problematic decomposition is called **lossy decompositions**.
- So, we will look for correct decomposition termed as **lossless decompositions**.

Schema Decomposition: The Bad One (Cont.)

ID	*name*	*street*	*city*	*salary*
⋮				
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

employee

ID	*name*
⋮	
57766	Kim
98776	Kim
⋮	

name	*street*	*city*	*salary*
⋮			
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000
⋮			

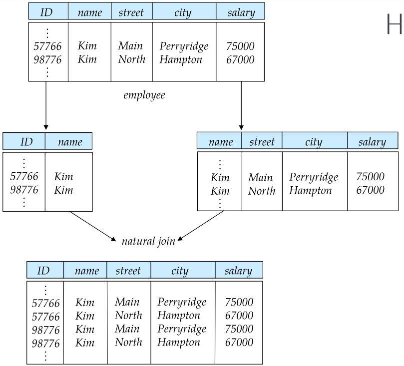
natural join

ID	*name*	*street*	*city*	*salary*
⋮				
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

Here observe:

- We **fail to produce the original records** by natural join.
- It results in **incorrect** records.
- Such problematic decomposition is called **lossy decompositions**.
- So, we will look for correct decomposition termed as **lossless decompositions**.

Schema Decomposition: The Bad One (Cont.)



Here observe:

- We **fail to produce the original records** by natural join.
- It results in **incorrect** records.
- Such problematic decomposition is called **lossy decompositions**.
- So, we will look for correct decomposition termed as **lossless decompositions**.

Lossless Decomposition

Definition

Let R be a relation schema and let R1 and R2 form a decomposition of R that is, view– ing R, R1, and R2 as sets of attributes, $R = R1 \cap R2$. We say that the decomposition is a lossless decomposition if there is no loss of information by replacing R with two relation schemas R1 and R2 . In simple language, **Decomposition is lossless if it is feasible to reconstruct relation R from decomposed relations R1 and R2 using Joins**.

In terms of relational algebra:

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

Lossless Decomposition

Definition

Let R be a relation schema and let R1 and R2 form a decomposition of R that is, view– ing R, R1, and R2 as sets of attributes, $R = R1 \cap R2$. We say that the decomposition is a lossless decomposition if there is no loss of information by replacing R with two relation schemas R1 and R2 . In simple language, **Decomposition is lossless if it is feasible to reconstruct relation R from decomposed relations R1 and R2 using Joins**.

In terms of relational algebra:

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

Lossless Decomposition (Cont.)

Lossy Decomposition

Conversely, a decomposition is lossy if when we compute the natural join of the projection results, we get a **proper superset** of the original relation. **Here, we have more tuples but less information.**

In terms of relational algebra:

$$r \subset \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$$

Lossless Decomposition (Cont.)

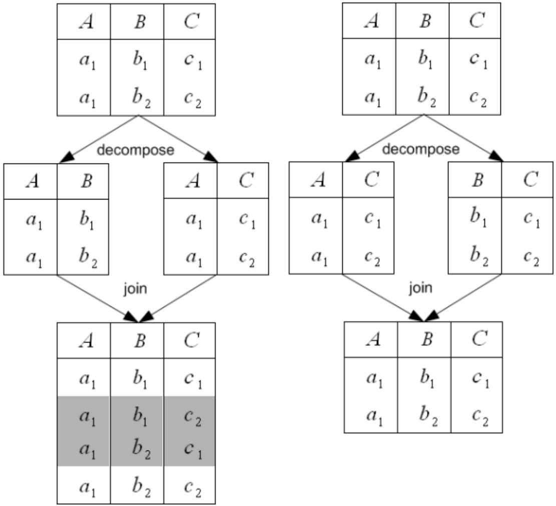
Lossy Decomposition

Conversely, a decomposition is lossy if when we compute the natural join of the projection results, we get a **proper superset** of the original relation. **Here, we have more tuples but less information.**

In terms of relational algebra:

$$r \subset \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$$

Decompositions: By a Simple Example



Normalization Theory [1]

Motivation

A general methodology for deriving a set of schemas each of which is in **good form**. Process is commonly known as **normalization**. The goal is

1. To generate a set of relation schemas that allows us to store information **without unnecessary redundancy**.
2. Yet also allows us to **retrieve information easily**.

[1] Textbook (7e) Reference 7.1.3

Normalization Theory [1]

Motivation

A general methodology for deriving a set of schemas each of which is in **good form**. Process is commonly known as **normalization**. The goal is

1. To generate a set of relation schemas that allows us to store information **without unnecessary redundancy**.
2. Yet also allows us to **retrieve information easily**.

[1] Textbook (7e) Reference 7.1.3

Decomposition Using Functional Dependencies

There are usually a variety of **constraints (rules)** on the data in the real world.

For example:

- Students and instructors are **uniquely identified** by their ID.
- Each student and instructor has **only one name**.
- Each instructor and student is (primarily) **associated with only one department**.
- Each department has **only one value for its budget**, and only one associated building.

Legal Instance

An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation; a **legal instance of a database** is one where all the relation instances are legal instances.

Decomposition Using Functional Dependencies

There are usually a variety of **constraints (rules)** on the data in the real world.
For example:

- Students and instructors are **uniquely identified** by their ID.
- Each student and instructor has **only one name**.
- Each instructor and student is (primarily) **associated with only one department**.
- Each department has **only one value for its budget**, and only one associated building.

Legal Instance

An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation; a **legal instance of a database** is one where all the relation instances are legal instances.

Decomposition Using Functional Dependencies

There are usually a variety of **constraints (rules)** on the data in the real world.
For example:

- Students and instructors are **uniquely identified** by their ID.
- Each student and instructor has **only one name**.
- Each instructor and student is (primarily) **associated with only one department**.
- Each department has **only one value for its budget**, and only one associated building.

Legal Instance

An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation; a **legal instance of a database** is one where all the relation instances are legal instances.

Decomposition Using Functional Dependencies

There are usually a variety of **constraints (rules)** on the data in the real world.
For example:

- Students and instructors are **uniquely identified** by their ID.
- Each student and instructor has **only one name**.
- Each instructor and student is (primarily) **associated with only one department**.
- Each department has **only one value for its budget**, and only one associated building.

Legal Instance

An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation; a **legal instance of a database** is one where all the relation instances are legal instances.

Decomposition Using Functional Dependencies

There are usually a variety of **constraints (rules)** on the data in the real world.

For example:

- Students and instructors are **uniquely identified** by their ID.
- Each student and instructor has **only one name**.
- Each instructor and student is (primarily) **associated with only one department**.
- Each department has **only one value for its budget**, and only one associated building.

Legal Instance

An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation; a **legal instance of a database** is one where all the relation instances are legal instances.

Decomposition Using Functional Dependencies

There are usually a variety of **constraints (rules)** on the data in the real world.

For example:

- Students and instructors are **uniquely identified** by their ID.
- Each student and instructor has **only one name**.
- Each instructor and student is (primarily) **associated with only one department**.
- Each department has **only one value for its budget**, and only one associated building.

Legal Instance

An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation; a **legal instance of a database** is one where all the relation instances are legal instances.

Decomposition Using Functional Dependencies: Notations

- Greek letters for sets of attributes (**e.g,** α, β) (When we use a lowercase Greek letter, we are referring to a set of attributes that may or may not be a schema.)
- Roman letter (**e.g** A,B,C) for set of attributes which forms a schema.
- Set of attributes is a superkey K. A superkey pertains to a specific relation schema, so we use the terminology **K is a superkey of r(R)**.
- We use a lowercase name for relations.(for example, instructor).
- **Instance of r**: a particular value at any given time.

Decomposition Using Functional Dependencies: Notations

- Greek letters for sets of attributes (**e.g,** α, β) (When we use a lowercase Greek letter, we are referring to a set of attributes that may or may not be a schema.)
- Roman letter (**e.g** A,B,C) for set of attributes which forms a schema.
- Set of attributes is a superkey K. A superkey pertains to a specific relation schema, so we use the terminology **K is a superkey of r(R)**.
- We use a lowercase name for relations.(for example, instructor).
- **Instance of r**: a particular value at any given time.

Decomposition Using Functional Dependencies: Notations

- Greek letters for sets of attributes (**e.g,** α, β) (When we use a lowercase Greek letter, we are referring to a set of attributes that may or may not be a schema.)
- Roman letter (**e.g** A,B,C) for set of attributes which forms a schema.
- Set of attributes is a superkey K. A superkey pertains to a specific relation schema, so we use the terminology **K is a superkey of r(R)**.
- We use a lowercase name for relations.(for example, instructor).
- **Instance of r**: a particular value at any given time.

Decomposition Using Functional Dependencies: Notations

- Greek letters for sets of attributes (**e.g,** α, β) (When we use a lowercase Greek letter, we are referring to a set of attributes that may or may not be a schema.)
- Roman letter (**e.g** A,B,C) for set of attributes which forms a schema.
- Set of attributes is a superkey K. A superkey pertains to a specific relation schema, so we use the terminology **K is a superkey of r(R)**.
- We use a lowercase name for relations.(for example, instructor).
- **Instance of r**: a particular value at any given time.

Decomposition Using Functional Dependencies: Notations

- Greek letters for sets of attributes (**e.g,** α, β) (When we use a lowercase Greek letter, we are referring to a set of attributes that may or may not be a schema.)
- Roman letter (**e.g** A,B,C) for set of attributes which forms a schema.
- Set of attributes is a superkey K. A superkey pertains to a specific relation schema, so we use the terminology **K is a superkey of r(R)**.
- We use a lowercase name for relations.(for example, instructor).
- **Instance of r**: a particular value at any given time.

Lossless Decomposition and Functional Dependencies (FD)

- We can use functional dependencies to show when certain decompositions are lossless.
- Let R, R1, R2 , and F be as above. R1 and R2 form a lossless decomposition of R if at-least one of the following functional dependencies is in F+ :
  1. $R1 \cap R2 \longrightarrow R1$
  2. $R1 \cap R2 \longrightarrow R2$

Meaning

The 2 conditions means that if there is any attribute is common and for 1st condition says the common attribute $R1 \cap R2$ is the **primary key** of the first Relation R1 and of course that attribute must be a **foreign key** for R2 referencing R1 (since it is common) [and vice-versa]

Lossless Decomposition and FD: Example

Lets consider the schema:

inst_dept (ID , name, salary, dept name, building, budget)

Now we split it into the instructor and department schemas:

department(dept name, building, budget)

instructor(ID , name, dept name, salary)

So,

The intersection of these two schemas, which is **dept name**. We see that **dept name** $\longrightarrow$ **dept name, building, budget** holds, thus the lossless-decomposition rule is satisfied.

Keys and Functional Dependencies

Some of the most commonly used types of real-world constraints can be represented formally as:

1. **Keys** (superkeys, candidate keys, and primary keys)
2. **Functional Dependencies(FD)** (will be discussed now)

Keys and Functional Dependencies

Some of the most commonly used types of real-world constraints can be represented formally as:

1. **Keys** (superkeys, candidate keys, and primary keys)
2. **Functional Dependencies(FD)** (will be discussed now)

Keys and Functional Dependencies

Superkey Definition: (Recall)

A superkey as a set of one or more attributes that, taken collectively, allows us to identify uniquely a tuple in the relation.

Superkey Definition: (Revisited)

Let $r(R)$ be a relation schema. A subset K of R is a superkey of $r(R)$ if, in any legal instance of $r(R)$, for all pairs t_1 and t_2 of tuples in the instance of r if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$.

Keys and Functional Dependencies

Superkey Definition: (Recall)

A superkey as a set of one or more attributes that, taken collectively, allows us to identify uniquely a tuple in the relation.

Superkey Definition: (Revisited)

Let r (R) be a relation schema. A subset K of R is a superkey of $r(R)$ if, in any legal instance of $r(R)$, for all pairs t_1 and t_2 of tuples in the instance of r if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$.

Superkey Definition: Revisited (Example)

Superkey Definition: (Revisited)

K is a superkey.... for all pairs t_1 *and* t_2 *of tuples in the instance of* r
if $t_1 \neq t_2$ *, then* $t_1[K] \neq t_2[K]$.

recordNo(t_i)	α	β	γ
1	a	b	c
2	a	b	d

In reality:
$\alpha = Name$
$\beta = Address$
$\gamma = Salary$

- Let $K = \alpha\beta$
- For $t_1 \neq t_2$ We compute:
- So, $K = \alpha\beta$ is not a Superkey.
- By similar comparison, $K = \alpha\gamma$ is a Superkey (other possibilities exist).

Superkey Definition: Revisited (Example)

Superkey Definition: (Revisited)

K is a superkey.... for all pairs t_1 and t_2 of tuples in the instance of r
if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$.

recordNo(t_i)	α	β	γ
1	a	b	c
2	a	b	d

In reality:

$\alpha = Name$

$\beta = Address$

$\gamma = Salary$

- Let $K = \alpha\beta$
- For $t_1 \neq t_2$ We compute:
  - $t_1(\alpha\beta) = ab$
  - $t_2(\alpha\beta) = ab$
- So, $K = \alpha\beta$ is not a Superkey.
- By similar comparison, $K = \alpha\gamma$ is a Superkey, (other possibilities exist)

Superkey Definition: Revisited (Example)

Superkey Definition: (Revisited)

K is a superkey.... for all pairs t_1 and t_2 of tuples in the instance of r
if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$.

recordNo(t_i)	α	β	γ
1	a	b	c
2	a	b	d

In reality:

$\alpha = Name$

$\beta = Address$

$\gamma = Salary$

- Let $K = \alpha\beta$
- For $t_1 \neq t_2$ We compute:
  - $t_1(\alpha\beta) = ab$
  - $t_2(\alpha\beta) = ab$
- So, $K = \alpha\beta$ is not a Superkey.
- By similar comparison, $K = \alpha\gamma$ is a Superkey, (other possibilities exist)

Superkey Definition: Revisited (Example)

Superkey Definition: (Revisited)

K is a superkey.... for all pairs t_1 and t_2 of tuples in the instance of r
if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$.

recordNo(t_i)	α	β	γ
1	a	b	c
2	a	b	d

In reality:

$\alpha = Name$

$\beta = Address$

$\gamma = Salary$

- Let $K = \alpha\beta$
- For $t_1 \neq t_2$ We compute:
  - $t_1(\alpha\beta) = ab$
  - $t_2(\alpha\beta) = ab$
- So, $K = \alpha\beta$ is not a Superkey.
- By similar comparison, $K = \alpha\gamma$ is a Superkey, (other possibilities exist)

Superkey Definition: Revisited (Example)

Superkey Definition: (Revisited)

K is a superkey.... for all pairs t_1 and t_2 of tuples in the instance of r
if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$.

recordNo(t_i)	α	β	γ
1	a	b	c
2	a	b	d

In reality:

$\alpha = Name$

$\beta = Address$

$\gamma = Salary$

- Let $K = \alpha\beta$
- For $t_1 \neq t_2$ We compute:
  - $t_1(\alpha\beta) = ab$
  - $t_2(\alpha\beta) = ab$
- So, $K = \alpha\beta$ is not a Superkey.
- By similar comparison, $K = \alpha\gamma$ is a Superkey, (other possibilities exist)

Superkey Definition: Revisited (Example)

Superkey Definition: (Revisited)

K is a superkey.... for all pairs t_1 and t_2 of tuples in the instance of r
if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$.

recordNo(t_i)	α	β	γ
1	a	b	c
2	a	b	d

In reality:

$\alpha = Name$

$\beta = Address$

$\gamma = Salary$

- Let $K = \alpha\beta$
- For $t_1 \neq t_2$ We compute:
  - $t_1(\alpha\beta) = ab$
  - $t_2(\alpha\beta) = ab$
- So, $K = \alpha\beta$ is not a Superkey.
- By similar comparison, $K = \alpha\gamma$ is a Superkey, (other possibilities exist)

Superkey Definition: Revisited (Example)

Superkey Definition: (Revisited)

K is a superkey.... for all pairs t_1 and t_2 of tuples in the instance of r
if $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$.

recordNo(t_i)	α	β	γ
1	a	b	c
2	a	b	d

In reality:

$\alpha = Name$

$\beta = Address$

$\gamma = Salary$

- Let $K = \alpha\beta$
- For $t_1 \neq t_2$ We compute:
  - $t_1(\alpha\beta) = ab$
  - $t_2(\alpha\beta) = ab$
- So, $K = \alpha\beta$ is not a Superkey.
- By similar comparison, $K = \alpha\gamma$ is a Superkey, (other possibilities exist)

Functional Dependency: Definition Revisited

Functional Dependency

Consider a relation schema r (R), and let $\alpha \subseteq R$ and $\beta \subseteq R$.

- Given an instance of r(R), we say that the instance satisfies the functional dependency $\alpha \rightarrow \beta$ if for all pairs of tuples t 1 and t 2 in the instance such that : $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$.
  α determines β or β is determined by α
- We say that the functional dependency $\alpha \rightarrow \beta$ holds on schema r(R) if, in every legal instance of r (R) it satisfies the functional dependency. In other words, this is not a co-incidence rather the mapping is a result of some required rules.

- The first point is the basic definition.
- The second point is to ensure that "functional dependency $\alpha \rightarrow \beta$ holds" means this property is **valid over all data at all time for that relation**.

Functional Dependency: Definition Revisited

Functional Dependency

Consider a relation schema r (R), and let $\alpha \subseteq R$ and $\beta \subseteq R$.

- Given an instance of r(R), we say that the instance **satisfies** the **functional dependency** $\alpha \rightarrow \beta$ if for all pairs of tuples t 1 and t 2 in the instance such that : $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$.
  α **determines** β **or** β **is determined by** α
- We say that the functional dependency $\alpha \rightarrow \beta$ holds on schema r(R) if, **in every legal instance of r (R) it satisfies the functional dependency**. In other words, this is not a co-incidence rather the mapping is a result of some required rules.

- The first point is the basic definition.
- The second point is to ensure that "functional dependency $\alpha \rightarrow \beta$ holds" means this property is **valid over all data at all time for that relation.**

Functional Dependency: Definition Revisited

Functional Dependency

Consider a relation schema r (R), and let $\alpha \subseteq R$ and $\beta \subseteq R$.

- Given an instance of r(R), we say that the instance **satisfies** the **functional dependency** $\alpha \rightarrow \beta$ if for all pairs of tuples t 1 and t 2 in the instance such that : $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$.
  α **determines** β **or** β **is determined by** α
- We say that the functional dependency $\alpha \rightarrow \beta$ holds on schema r(R) if, **in every legal instance of r (R) it satisfies the functional dependency**. In other words, this is not a co-incidence rather the mapping is a result of some required rules.

- The first point is the basic definition.
- The second point is to ensure that functional dependency $\alpha \rightarrow \beta$ holds means this property is **valid over all data at all time for that relation.**

Functional Dependency: Definition Revisited

Functional Dependency

Consider a relation schema r (R), and let $\alpha \subseteq R$ and $\beta \subseteq R$.

- Given an instance of r(R), we say that the instance **satisfies** the **functional dependency** $\alpha \rightarrow \beta$ if for all pairs of tuples t 1 and t 2 in the instance such that : $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$.
  α **determines** β **or** β **is determined by** α
- We say that the functional dependency $\alpha \rightarrow \beta$ holds on schema r(R) if, **in every legal instance of r (R) it satisfies the functional dependency**. In other words, this is not a co-incidence rather the mapping is a result of some required rules.

- The first point is the basic definition.
- The second point is to ensure that functional dependency $\alpha \rightarrow \beta$ holds means this property is **valid over all data at all time for that relation**.

Functional Dependency: Definition Revisited

Functional Dependency

Consider a relation schema r (R), and let $\alpha \subseteq R$ and $\beta \subseteq R$.

- Given an instance of r(R), we say that the instance **satisfies** the **functional dependency** $\alpha \rightarrow \beta$ if for all pairs of tuples t 1 and t 2 in the instance such that : $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$.
  α **determines** β **or** β **is determined by** α
- We say that the functional dependency $\alpha \rightarrow \beta$ holds on schema r(R) if, **in every legal instance of r (R) it satisfies the functional dependency**. In other words, this is not a co-incidence rather the mapping is a result of some required rules.

- The first point is the basic definition.
- The second point is to ensure that functional dependency $\alpha \rightarrow \beta$ holds means this property is **valid over all data at all time for that relation**.

Functional Dependency: Definition Revisited

Functional Dependency

Consider a relation schema r (R), and let $\alpha \subseteq R$ and $\beta \subseteq R$.

- Given an instance of r(R), we say that the instance **satisfies** the **functional dependency** $\alpha \rightarrow \beta$ if for all pairs of tuples t 1 and t 2 in the instance such that : $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$.
  α **determines** β **or** β **is determined by** α
- We say that the functional dependency $\alpha \rightarrow \beta$ holds on schema r(R) if, **in every legal instance of r (R) it satisfies the functional dependency**. In other words, this is not a co-incidence rather the mapping is a result of some required rules.

- The first point is the basic definition.
- The second point is to ensure that functional dependency $\alpha \rightarrow \beta$ holds means this property is **valid over all data at all time for that relation**.

Functional Dependency: Example

Functional Dependency Definition. (Recall)

...**functional dependency** $\alpha \rightarrow \beta$ if for all pairs of tuples t 1 and t 2 in the instance such that :
$t_1[\alpha] = t_2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$

recordNo(t_i)	α	β	γ
1	a	b	d
2	a	s	d
3	m	r	w
4	q	s	d

- Here, $t_2(\beta) = s$ and $t_4(\beta) = s$; $t_2(\beta) = t_4(\beta)$
- And, $t_2(\gamma) = d$ and $t_4(\gamma) = d$; $t_2(\gamma) = t_4(\gamma)$
- So,Functional Dependency $\beta \longrightarrow \gamma$ holds.

Functional Dependency: Example

Functional Dependency Definition. (Recall)

...**functional dependency** $\alpha \rightarrow \beta$ if for all pairs of tuples t 1 and t 2 in the instance such that : $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$

recordNo(t_i)	α	β	γ
1	a	b	d
2	a	s	d
3	m	r	w
4	q	s	d

- Here, $t_2(\beta) = s$ and $t_4(\beta) = s$; $t_2(\beta) = t_4(\beta)$
- And, $t_2(\gamma) = d$ and $t_4(\gamma) = d$; $t_2(\gamma) = t_4(\gamma)$
- So,Functional Dependency $\beta \longrightarrow \gamma$ holds.

Functional Dependency: Example

Functional Dependency Definition. (Recall)

...**functional dependency** $\alpha \rightarrow \beta$ if for all pairs of tuples t 1 and t 2 in the instance such that : $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$

recordNo(t_i)	α	β	γ
1	a	b	d
2	a	s	d
3	m	r	w
4	q	s	d

- Here, $t_2(\beta) = s$ and $t_4(\beta) = s$; $t_2(\beta) = t_4(\beta)$
- And, $t_2(\gamma) = d$ and $t_4(\gamma) = d$; $t_2(\gamma) = t_4(\gamma)$
- So,Functional Dependency $\beta \longrightarrow \gamma$ holds.

Functional Dependency: Example

Functional Dependency Definition. (Recall)

...**functional dependency** $\alpha \rightarrow \beta$ if for all pairs of tuples t 1 and t 2 in the instance such that : $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$

recordNo(t_i)	α	β	γ
1	a	b	d
2	a	s	d
3	m	r	w
4	q	s	d

- Here, $t_2(\beta) = s$ and $t_4(\beta) = s$; $t_2(\beta) = t_4(\beta)$
- And, $t_2(\gamma) = d$ and $t_4(\gamma) = d$; $t_2(\gamma) = t_4(\gamma)$
- So,Functional Dependency $\beta \longrightarrow \gamma$ holds.

Functional Dependency: Example

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

- (remember the constraint) Each department has **only one value for its budget**, and **only one associated building**.
- Here, $dept\_name \longrightarrow budget$ and $dept\_name \longrightarrow building$ hold.

Functional Dependency: Example

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

- (remember the constraint) Each department has **only one value for its budget**, and **only one associated building**.
- Here, $dept\_name \longrightarrow budget$ and $dept\_name \longrightarrow building$ hold.

Types of Functional Dependency

There are in general 4 types of FD:

1. Trivial (✔)
2. Non–Trivial (✔)
3. Muti–valued (needed for 4NF only) (✘)
4. Transitive (✔)

Trivial Functional Dependency

- They are called **trivial** as because they are **satisfied by all relations, always valid**.
- For example, $A \longrightarrow A$ is satisfied by all relations involving attribute A.
- In the same way, $AB \longrightarrow A$ is satisfied by all relations involving attribute A.
- In general, a functional dependency of the form $\alpha \longrightarrow \beta$ is trivial if $\beta \subseteq \alpha$
- Example: $StudentID \longrightarrow StuedentID$
- Example: $StudentID, Dept \longrightarrow StuedentID$

Trivial Functional Dependency

- They are called **trivial** as because they are **satisfied by all relations, always valid**.
- For example, $A \longrightarrow A$ is satisfied by all relations involving attribute A.
- In the same way, $AB \longrightarrow A$ is satisfied by all relations involving attribute A.
- In general, a functional dependency of the form $\alpha \longrightarrow \beta$ is trivial if $\beta \subseteq \alpha$
- Example: $StudentID \longrightarrow StuedentID$
- Example: $StudentID, Dept \longrightarrow StuedentID$

Trivial Functional Dependency

- They are called **trivial** as because they are **satisfied by all relations, always valid**.
- For example, $A \longrightarrow A$ is satisfied by all relations involving attribute A.
- In the same way, $AB \longrightarrow A$ is satisfied by all relations involving attribute A.
- In general, a functional dependency of the form $\alpha \longrightarrow \beta$ is trivial if $\beta \subseteq \alpha$
- Example: $StudentID \longrightarrow StuedentID$
- Example: $StudentID, Dept \longrightarrow StuedentID$

Trivial Functional Dependency

- They are called **trivial** as because they are **satisfied by all relations, always valid**.
- For example, $A \longrightarrow A$ is satisfied by all relations involving attribute A.
- In the same way, $AB \longrightarrow A$ is satisfied by all relations involving attribute A.
- In general, a functional dependency of the form $\alpha \longrightarrow \beta$ is trivial if $\beta \subseteq \alpha$
- Example: $StudentID \longrightarrow StuedentID$
- Example: $StudentID, Dept \longrightarrow StuedentID$

Trivial Functional Dependency

- They are called **trivial** as because they are **satisfied by all relations, always valid**.
- For example, $A \longrightarrow A$ is satisfied by all relations involving attribute A.
- In the same way, $AB \longrightarrow A$ is satisfied by all relations involving attribute A.
- In general, a functional dependency of the form $\alpha \longrightarrow \beta$ is trivial if $\beta \subseteq \alpha$
- Example: $StudentID \longrightarrow StuedentID$
- Example: $StudentID, Dept \longrightarrow StuedentID$

Trivial Functional Dependency

- They are called **trivial** as because they are **satisfied by all relations, always valid**.
- For example, $A \longrightarrow A$ is satisfied by all relations involving attribute A.
- In the same way, $AB \longrightarrow A$ is satisfied by all relations involving attribute A.
- In general, a functional dependency of the form $\alpha \longrightarrow \beta$ is trivial if $\beta \subseteq \alpha$
- Example: $StudentID \longrightarrow StuedentID$
- Example: $StudentID, Dept \longrightarrow StuedentID$

Non-Trivial Functional Dependency

- For trivial functional dependency **no need to check, it is always true**.
- If $X \longrightarrow Y$ and $X \cap Y = \phi$ (ie. no common attribute)
- Example: $StudentID \longrightarrow CGA$
- We **can not say instantly** if it holds there, we need to observe the rules or constraints supporting the dependency or not.

Non-Trivial Functional Dependency

- For trivial functional dependency **no need to check, it is always true**.
- If $X \longrightarrow Y$ and $X \cap Y = \phi$ (ie. no common attribute)
- Example: $StudentID \longrightarrow CGA$
- We **can not say instantly** if it holds there, we need to observe the rules or constraints supporting the dependency or not.

Non-Trivial Functional Dependency

- For trivial functional dependency **no need to check, it is always true**.
- If $X \longrightarrow Y$ and $X \cap Y = \phi$ (ie. no common attribute)
- Example: $StudentID \longrightarrow CGA$
- We **can not say instantly** if it holds there, we need to observe the rules or constraints supporting the dependency or not.

Non-Trivial Functional Dependency

- For trivial functional dependency **no need to check, it is always true**.
- If $X \longrightarrow Y$ and $X \cap Y = \phi$ (ie. no common attribute)
- Example: $StudentID \longrightarrow CGA$
- We **can not say instantly** if it holds there, we need to observe the rules or constraints supporting the dependency or not.

Semi-Trivial Functional Dependency

- For trivial functional dependency **no need to check, it is always true**.
- If $X \longrightarrow Y$ and $Y \subsetneq X$ (ie. Y is **not a subset** of X and $Y \cap X \neq \phi$)
- Example: $StudentID, Name \longrightarrow Name, CGPA$
- We **can not say instantly here also like non-trivial** if it holds there, we need check it.

Semi-Trivial Functional Dependency

- For trivial functional dependency **no need to check, it is always true**.
- If $X \longrightarrow Y$ and $Y \subsetneq X$ (ie. Y is **not a subset** of X and $Y \cap X \neq \phi$)
- Example: $StudentID, Name \longrightarrow Name, CGPA$
- We **can not say instantly here also like non-trivial** if it holds there, we need check it.

Semi-Trivial Functional Dependency

- For trivial functional dependency **no need to check, it is always true**.
- If $X \longrightarrow Y$ and $Y \subsetneq X$ (ie. Y is **not a subset** of X and $Y \cap X \neq \phi$)
- Example: $StudentID, Name \longrightarrow Name, CGPA$
- We **can not say instantly here also like non-trivial** if it holds there, we need check it.

Semi-Trivial Functional Dependency

- For trivial functional dependency **no need to check, it is always true**.
- If $X \longrightarrow Y$ and $Y \subsetneq X$ (ie. Y is **not a subset** of X and $Y \cap X \neq \phi$)
- Example: $StudentID, Name \longrightarrow Name, CGPA$
- We **can not say instantly here also like non-trivial** if it holds there, we need check it.

Multi-valued & Transitive Functional Dependency

- Not covered here (Only used in 4NF).
- Given a relation R if there exist FD: $\alpha \longrightarrow \beta$ and $\beta \longrightarrow \gamma$ then the relation R holds transitive FD:$\alpha \longrightarrow \gamma$

Closure of the set F: F+ [1]

- Given that a set of functional dependencies F holds on a relation r(R), it may be possible to infer that **certain other functional dependencies must also hold**.
- For instance if $A \longrightarrow B$ and $B \longrightarrow C$ then by transitivity rule (will be detailed later) we can **infer** $A \longrightarrow C$
- When testing for normal forms, it is not sufficient to consider the given set of functional dependencies; rather, we need to **consider all functional dependencies** that hold on the schema.

[1] Textbook (7e) Reference 7.4.1

Closure of the set F: F+ [1]

- Given that a set of functional dependencies F holds on a relation r(R), it may be possible to infer that **certain other functional dependencies must also hold**.
- For instance if $A \longrightarrow B$ and $B \longrightarrow C$ then by transitivity rule (will be detailed later) we can **infer** $A \longrightarrow C$
- When testing for normal forms, it is not sufficient to consider the given set of functional dependencies; rather, we need to **consider all functional dependencies** that hold on the schema.

[1] Textbook (7e) Reference 7.4.1

Closure of the set F: F+ [1]

- Given that a set of functional dependencies F holds on a relation r(R), it may be possible to infer that **certain other functional dependencies must also hold**.
- For instance if $A \longrightarrow B$ and $B \longrightarrow C$ then by transitivity rule (will be detailed later) we can **infer** $A \longrightarrow C$
- When testing for normal forms, it is not sufficient to consider the given set of functional dependencies; rather, we need to **consider all functional dependencies** that hold on the schema.

[1] Textbook (7e) Reference 7.4.1

Closure of the set F: F+ [1]

Definition

Let F be a set of functional dependencies. The closure of F , denoted by F+ , is the set of all functional dependencies **logically implied by F** . Given F , we can compute F+ directly from the formal definition of functional dependency.

If F were large, this process would be *lengthy and difficult.* So, we use some rules of inference (Called Armstrong's Axioms to speed up the process).

Armstrongs Axioms (Inference Rules)

Motivation

- Given F , we can compute F+ **directly from** the formal definition of functional dependency.
- But for **larger F**, this manual process would be **tiresome and inefficient**. So, we can use of some set of rules to simplify the process: called **Armstrong's Axioms** named after William W. Armstrong who proposed it in 1974.
- Armstrongs Axioms are **used to infer all the functional dependencies** on a relational database given F.

Armstrongs Axioms (Inference Rules)

Motivation

- Given F , we can compute F+ **directly from** the formal definition of functional dependency.
- But for **larger F**, this manual process would be **tiresome and inefficient**. So, we can use of some set of rules to simplify the process: called **Armstrong's Axioms** named after William W. Armstrong who proposed it in 1974.
- Armstrongs Axioms are **used to infer all the functional dependencies** on a relational database given F.

Armstrongs Axioms (Inference Rules)

Motivation

- Given F , we can compute F+ **directly from** the formal definition of functional dependency.
- But for **larger F**, this manual process would be **tiresome and inefficient**. So, we can use of some set of rules to simplify the process: called **Armstrong's Axioms** named after William W. Armstrong who proposed it in 1974.
- Armstrongs Axioms are **used to infer all the functional dependencies** on a relational database given F.

Armstrongs Axioms (Cont.)

Primary Rules:

1. **Reflexivity rule.** If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \longrightarrow \beta$ holds. (i.e it is trivial dependency)
2. **Augmentation rule.** If $\alpha \longrightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \longrightarrow \gamma\beta$ holds.
3. **Transitivity rule.** If $\alpha \longrightarrow \beta$ holds and $\beta \longrightarrow \gamma$ holds, then $\alpha \longrightarrow \gamma$ holds. (*this rule is very commonly used*)

Completeness and soundness of the Rules

Armstrongs axioms are **sound**, *because* they do not generate any incorrect functional dependencies. They are **complete**, *because*, for a given set F of functional dependencies, they allow us to generate all F+ (no additional FD can be derived).

Armstrongs Axioms: Additional Rules

Additional or Secondary Rules:

Motivation

Although Primary Rules are both sound and complete, some additional rule will **ease** the process. They are called **Secondary or Additional Rules**. (*Just like* NAND and NOR gates are universal gates but still we have AND, OR gates) It is possible to use Armstrongs axioms to prove that these rules are sound.

Armstrongs Axioms: Additional Rules (Cont.)

1. **Union rule.** If $\alpha \longrightarrow \beta$ holds and $\alpha \longrightarrow \gamma$ holds, then $\alpha \longrightarrow \beta\gamma$ holds.
2. **Decomposition rule.** If $\alpha \longrightarrow \beta\gamma$ holds, then $\alpha \longrightarrow \beta$ holds and $\alpha \longrightarrow \gamma$ holds. (The decomposition rule is only applicable for the dependent part (i.e Right Hand Side))
3. **Pseudotransitivity rule.** If $\alpha \longrightarrow \beta$ holds and $\gamma\beta \longrightarrow \delta$ holds, then $\alpha\gamma \longrightarrow \delta$ holds.
4. **Composition rule.** If $\alpha \longrightarrow \beta$ and $\gamma \longrightarrow \delta$ hold then $\alpha\gamma \longrightarrow \beta\delta$

Note: **Composition rule** is a **generalization** of the **Union rule**.

Armstrong's Axioms: A Table Data Example

Objective: To have an **intuitive idea** of these rules in **regard to real-life data**.

SID	Dept	Budget	Est.	Hall	CID	Credit	Grade
1	CSE	120	1999	North	CSE101	3	A
2	EEE	110	1995	South	EEE101	2	B
2	EEE	110	1995	South	EEE102	4	A
1	CSE	120	1999	North	CSE102	1.5	C
3	EEE	110	1995	North	EEE102	1.5	D

Table: Grades

Students Classroom Task

Given the above data, students will verify all primary and secondary rules of Armstrong's Axioms.

Armstrong's Axioms: A Table Data Example

Objective: To have an **intuitive idea** of these rules in **regard to real-life data**.

SID	Dept	Budget	Est.	Hall	CID	Credit	Grade
1	CSE	120	1999	North	CSE101	3	A
2	EEE	110	1995	South	EEE101	2	B
2	EEE	110	1995	South	EEE102	4	A
1	CSE	120	1999	North	CSE102	1.5	C
3	EEE	110	1995	North	EEE102	1.5	D

Table: Grades

Students Classroom Task

Given the above data, students will verify all primary and secondary rules of Armstrong's Axioms.

Armstrong's Axioms: Example Solution

SID	Dept	Budget	Est.	Hall	CID	Credit	Grade
1	CSE	120	1999	North	CSE101	3	A
2	EEE	110	1995	South	EEE101	2	B
2	EEE	110	1995	South	EEE102	4	A
1	CSE	120	1999	North	CSE102	1.5	C
3	EEE	110	1995	North	EEE102	1.5	D

- **Reflexivity rule.** If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \longrightarrow \beta$ holds. (**Trivial**)

This is always true, for instance
Lets consider α=**SID,Dept** and β=**Dept** then, **1,CSE**$\longrightarrow$ **CSE** holds (always will hold for each value).

Armstrong's Axioms: Example Solution

SID	Dept	Budget	Est.	Hall	CID	Credit	Grade
1	CSE	120	1999	North	CSE101	3	A
2	EEE	110	1995	South	EEE101	2	B
2	EEE	110	1995	South	EEE102	4	A
1	CSE	120	1999	North	CSE102	1.5	C
3	EEE	110	1995	North	EEE102	1.5	D

- **Augmentation rule.** If $\alpha \longrightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \longrightarrow \gamma\beta$ holds.

Lets consider α=**Dept** and β=**Budget** and γ=SID
Here we observe that, **Dept** $\longrightarrow$ **Budget** holds
So, Dept,SID $\longrightarrow$ Budget,SID

EEE$\longrightarrow$110
implies EEE,2$\longrightarrow$110,2

Armstrong's Axioms: Example Solution

SID	Dept	Budget	Est.	Hall	CID	Credit	Grade
1	CSE	120	1999	North	CSE101	3	A
2	EEE	110	1995	South	EEE101	2	B
2	EEE	110	1995	South	EEE102	4	A
1	CSE	120	1999	North	CSE102	1.5	C
3	EEE	110	1995	North	EEE102	1.5	D

- **Transitivity rule.** If $\alpha \longrightarrow \beta$ holds and $\beta \longrightarrow \gamma$ holds, then $\alpha \longrightarrow \gamma$ holds. (*this rule is very commonly used*)

Lets consider α=**SID** and β=**Dept** and γ=**Budget**

SID$\longrightarrow$Dept and **Dept$\longrightarrow$Budget** hold

So, SID$\longrightarrow$Budget

Example, **1$\longrightarrow$CSE** and **CSE$\longrightarrow$120**,

Thus, 1$\longrightarrow$120

Armstrong's Axioms: Example Solution

SID	Dept	Budget	Est.	Hall	CID	Credit	Grade
1	CSE	120	1999	North	CSE101	3	A
2	EEE	110	1995	South	EEE101	2	B
2	EEE	110	1995	South	EEE102	4	A
1	CSE	120	1999	North	CSE102	1.5	C
3	EEE	110	1995	North	EEE102	1.5	D

- **Union rule.** If $\alpha \longrightarrow \beta$ holds and $\alpha \longrightarrow \gamma$ holds, then $\alpha \longrightarrow \beta\gamma$ holds.

Lets consider α=**SID** and β=**Dept,Hall**
Here, **SID⟶Dept** and **SID⟶Hall** hold
Implies: SID⟶Dept,Hall

Example, **1⟶CSE** and **1⟶North**
So, 1⟶CSE,North

Note: **Decomposition Rule** is just the reverse, so is here omitted

Armstrong's Axioms: Example Solution

SID	Dept	Budget	Est.	Hall	CID	Credit	Grade
1	CSE	120	1999	North	CSE101	3	A
2	EEE	110	1995	South	EEE101	2	B
2	EEE	110	1995	South	EEE102	4	A
1	CSE	120	1999	North	CSE102	1.5	C
3	EEE	110	1995	North	EEE102	1.5	D

- **Pseudotransitivity rule.** If $\alpha \longrightarrow \beta$ holds and $\gamma\beta \longrightarrow \delta$ holds, then $\alpha\gamma \longrightarrow \delta$ holds.

Lets consider α=**SID** and β=**Dept**, γ=**Budget** and δ=**Est**
So, **SID⟶Dept** and **Dept,Buget⟶Est** hold
Implies: SID,Buget⟶Est

Example, **2⟶EEE** and
EEE,110⟶1995
Thus, 2,110⟶1995

Armstrong's Axioms: Example Solution

SID	Dept	Budget	Est.	Hall	CID	Credit	Grade
1	CSE	120	1999	North	CSE101	3	A
2	EEE	110	1995	South	EEE101	2	B
2	EEE	110	1995	South	EEE102	4	A
1	CSE	120	1999	North	CSE102	1.5	C
3	EEE	110	1995	North	EEE102	1.5	D

- **Composition rule.** If $\alpha \longrightarrow \beta$ and $\gamma \longrightarrow \delta$ hold then $\alpha\gamma \longrightarrow \beta\delta$

Lets consider α=**SID** and β=**Dept**, γ=**CID** and δ=**Credit**
So, $SID \longrightarrow Dept$ and $CID \longrightarrow Credit$ hold
Implies: $SID, CID \longrightarrow Dept, Credit$

Example, $1 \longrightarrow CSE$ and
$CSE102 \longrightarrow 1.5$
Thus, $1, CSE102 \longrightarrow Dept, 1.5$

(**Note:** Converse may not be true, as given **SID,CID⟶Dept,Credit** it is not possible to determine if $CID \longrightarrow Dept$ or $SID \longrightarrow Dept$ hold. There are 2 possible combinations in the Independent Side (RHS))

Armstrong's Axioms: Example2

Objective: Instead of looking at the physical data (as in the previous example) we can readily use the given FDs to deduce further FD.

Example

Suppose we are given a relation R with attribute A,B,C,D,E,F and FDs are:

$A \longrightarrow BC \qquad B \longrightarrow E \qquad CD \longrightarrow EF$

In reality, A=Emp No, B=dept No., C= Manager Emp No., D=Project No., E=Dept Name, F= pct of time spent by that manager for that project. *(Example adopted from C. J. Date's Book)*

Our task is to verify if FD: $AD \longrightarrow F$ holds or not.

Armstrong's Axioms: Example2 (Cont.)

Example

Suppose we are given a relation R with attribute A,B,C,D,E,F and FDs are:

$A \longrightarrow BC \qquad B \longrightarrow E \qquad CD \longrightarrow EF$

Need to verify if $AD \longrightarrow F$ holds.

Solution:

1. $A \longrightarrow BC$ (given)
2. $A \longrightarrow C$ (decomposition)
3. $AD \longrightarrow CD$ (2: augmentation)
4. $CD \longrightarrow EF$ (given)
5. $AD \longrightarrow EF$ (3,4 transitivity)
6. $AD \longrightarrow F$ (5: decomposition)

Armstrong's Axioms: Example2 (Cont.)

Example

Suppose we are given a relation R with attribute A,B,C,D,E,F and FDs are:
$A \longrightarrow BC \quad B \longrightarrow E \quad CD \longrightarrow EF$
Need to verify if $AD \longrightarrow F$ holds.

Solution:

1. $A \longrightarrow BC$ (given)
2. $A \longrightarrow C$ (decomposition)
3. $AD \longrightarrow CD$ (2: augmentation)
4. $CD \longrightarrow EF$ (given)
5. $AD \longrightarrow EF$ (3,4 transitivity)
6. $AD \longrightarrow F$ (5: decomposition)

Armstrong's Axioms: Example2 (Cont.)

Example

Suppose we are given a relation R with attribute A,B,C,D,E,F and FDs are:
$A \longrightarrow BC \qquad B \longrightarrow E \qquad CD \longrightarrow EF$
Need to verify if $AD \longrightarrow F$ holds.

Solution:

1. $A \longrightarrow BC$ (given)
2. $A \longrightarrow C$ (decomposition)
3. $AD \longrightarrow CD$ (2: augmentation)
4. $CD \longrightarrow EF$ (given)
5. $AD \longrightarrow EF$ (3,4 transitivity)
6. $AD \longrightarrow F$ (5: decomposition)

Armstrong's Axioms: Example2 (Cont.)

Example

Suppose we are given a relation R with attribute A,B,C,D,E,F and FDs are:
$A \longrightarrow BC \qquad B \longrightarrow E \qquad CD \longrightarrow EF$
Need to verify if $AD \longrightarrow F$ holds.

Solution:

1. $A \longrightarrow BC$ (given)
2. $A \longrightarrow C$ (decomposition)
3. $AD \longrightarrow CD$ (2: augmentation)
4. $CD \longrightarrow EF$ (given)
5. $AD \longrightarrow EF$ (3,4 transitivity)
6. $AD \longrightarrow F$ (5: decomposition)

Armstrong's Axioms: Example2 (Cont.)

Example

Suppose we are given a relation R with attribute A,B,C,D,E,F and FDs are:
$A \longrightarrow BC \qquad B \longrightarrow E \qquad CD \longrightarrow EF$
Need to verify if $AD \longrightarrow F$ holds.

Solution:

1. $A \longrightarrow BC$ (given)
2. $A \longrightarrow C$ (decomposition)
3. $AD \longrightarrow CD$ (2: augmentation)
4. $CD \longrightarrow EF$ (given)
5. $AD \longrightarrow EF$ (3,4 transitivity)
6. $AD \longrightarrow F$ (5: decomposition)

Armstrong's Axioms: Example2 (Cont.)

Example

Suppose we are given a relation R with attribute A,B,C,D,E,F and FDs are:

$A \longrightarrow BC \qquad B \longrightarrow E \qquad CD \longrightarrow EF$

Need to verify if $AD \longrightarrow F$ holds.

Solution:

1. $A \longrightarrow BC$ (given)
2. $A \longrightarrow C$ (decomposition)
3. $AD \longrightarrow CD$ (2: augmentation)
4. $CD \longrightarrow EF$ (given)
5. $AD \longrightarrow EF$ (3,4 transitivity)
6. $AD \longrightarrow F$ (5: decomposition)