Principles of Database Systems (CS307)

Lecture 10: Normalization - Part 1

Zhong-Qiu Wang

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

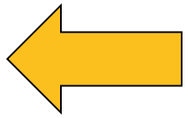
Normalization: A First Look

Design Alternatives

- In designing a database schema, we must ensure that we avoid two major pitfalls
  - **Redundancy**: a bad design may result in repeated information
    - E.g., store course identifier and title of a course for each course offering
      - Only store course identifier is sufficient
    - Redundant representation of information may lead to data inconsistency among the various copies of information
      - E.g., update is not performed on all the copies
  - **Incompleteness**: a bad design may make certain aspects of the enterprise difficult or impossible to model
    - E.g., only have entity for course offering, but without entity for courses
      - Impossible to model new courses that are not offered yet

Design Alternatives

- Avoiding bad designs is not enough
  - There may be many good designs from which we must choose
- For example, a customer who buys a product
  - The sale activity is a relationship between the customer and the product?
  - The sale activity is a relationship among the customer, the product, and the sale itself?
    - i.e., the sale can be considered as an entity
- Database design can be difficult
  - When #entities and #relationships are large
- Do we have any guidelines on how to get a good design?
  - Normal Forms (范式)!



Normalization （规范化）

- In practice, we usually just satisfy 1NF, 2NF and 3NF

	UNF (1970)	1NF (1970)	2NF (1971)	3NF (1971)	EKNF (1982)	BCNF (1974)	4NF (1977)	ETNF (2012)	5NF (1979)	DKNF (1981)	6NF (2003)
Primary key (no duplicate tuples)[4]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Atomic columns (cells cannot have tables as values)[5]	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency either does not begin with a proper subset of a candidate key or ends with a prime attribute (no partial functional dependencies of non-prime attributes on candidate keys)[5]	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency either begins with a superkey or ends with a prime attribute (no transitive functional dependencies of non-prime attributes on candidate keys)[5]	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency either begins with a superkey or ends with an elementary prime attribute	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	N/A
Every non-trivial functional dependency begins with a superkey	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	N/A
Every non-trivial multivalued dependency begins with a superkey	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	N/A
Every join dependency has a superkey component[8]	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	N/A
Every join dependency has only superkey components	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	N/A
Every constraint is a consequence of domain constraints and key constraints	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Every join dependency is trivial	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

https://en.wikipedia.org/wiki/Database_normalization

First Normal Form (1NF，第一范式)

- A relational schema *R* is in 1NF if the domains of all attributes of R are atomic
  - Domain is atomic if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS307 that can be broken up into parts
      - However, in practice, we can also consider it atomic
  - Non-atomic values complicate storage and encourage redundant (repeated) storage of data

First Normal Form (1NF)

- Example: Non-atomic attribute

station_id	name	location
1	Luohu(罗湖)	114.11833 , 22.53111
2	Guomao(国贸)	114.11889 , 22.54
3	Laojie(老街)	114.11639 , 22.54444
4	Grand Theater(大剧院)	114.10333 , 22.54472
5	Science Museum(科学馆)	114.08972 , 22.54333
6	Huaqiang Rd(华强路)	114.07889 , 22.54306
7	Gangxia(岗厦)	114.06306 , 22.53778
8	Convention and Exhibition Center Station(会展中心)	114.05472 , 22.5375
9	Shopping Park(购物公园)	114.05472 , 22.53444
10	Xiangmihu(香蜜湖)	114.034 , 22.5417

First Normal Form (1NF)

- Fix it by splitting the names into two columns

station_id	english_name	chinese_name	longitude	latitude
1	Luohu	罗湖	114.11833	22.53111
2	Guomao	国贸	114.11889	22.54
3	Laojie	老街	114.11639	22.54444
4	Grand Theater	大剧院	114.10333	22.54472
5	Science Museum	科学馆	114.08972	22.54333
6	Huaqiang Rd	华强路	114.07889	22.54306
7	Gangxia	岗厦	114.06306	22.53778
8	Convention and Exhibition Cent…	会展中心	114.05472	22.5375
9	Shopping Park	购物公园	114.05472	22.53444
10	Xiangmihu	香蜜湖	114.034	22.5417

First Normal Form (1NF)

- Another example: Starring
  - Problems: 1) Redundant names; 2) difficulties in updating/deleting a specific person; 3) extra cost in splitting names; 4) difficulties in making statistics

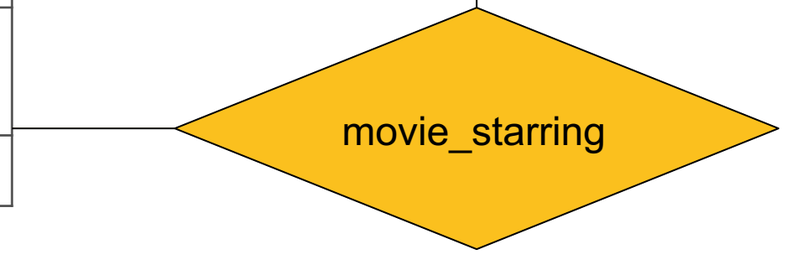
Movie ID	Movie Title	Country	Year	Director	Starring
0	Citizen Kane	US	1941	welles, o.	Orson Welles, Joseph Cotten
1	La règle du jeu	FR	1939	Renoir, J.	Roland Toutain, Nora Grégor, Marcel Dalio, Jean Renoir
2	North By Northwest	US	1959	HITCHCOCK, A.	Cary Grant, Eva Marie Saint, James Mason
3	Singin' in the Rain	US	1952	Donen/Kelly	Gene Kelly, Debbie Reynolds, Donald O'Connor
4	Rear Window	US	1954	Alfred Hitchcock	James Stewart, Grace Kelly

First Normal Form (1NF)

- Fix it by treating the column as a multi-valued attribute
  - *movie_starring* table has two foreign keys, *movid_id* and *star_id*

Movie ID	Movie Title	Country	Year	Director
0	Citizen Kane	US	1941	welles, o.
1	La règle du jeu	FR	1939	Renoir, J.
2	North By Northwest	US	1959	HITCHCOCK, A.
3	Singin' in the Rain	US	1952	Donen/Kelly
4	Rear Window	US	1954	Alfred Hitchcock

Star ID	Firstname	Lastname	Born	Died
1				
2				
3				



Second Normal Form (2NF，第二范式)

- A relation satisfying 2NF must:
  - be in 1NF
  - not have any non-prime attribute that is dependent on any proper subset of any candidate key of the relation
    - A non-prime attribute of a relation is an attribute that is not a part of any candidate key of the relation
  - 不 包含只依赖于 主键中部分属性 的非主属性
    - “非主属性”是指 不属于 任何候选键的属性

Second Normal Form (2NF)

- Example: consider this table with the composite primary key (*station_id*, *line_id*)

station_id	english_name	chinese_name	district	line_id	line_color	operator
1	Luohu	罗湖	Luohu	1	Green	Shenzhen Metro Corporation
2	Guomao	国贸	Luohu	1	Green	Shenzhen Metro Corporation
3	Laojie	老街	Luohu	1	Green	Shenzhen Metro Corporation
4	Grand Theater	大剧院	Luohu	1	Green	Shenzhen Metro Corporation
4	Grand Theater	大剧院	Luohu	11	Purple	Shenzhen Metro Corporation
4	Grand Theater	大剧院	Luohu	2	Orange	Shenzhen Metro Corporation
3	Laojie	老街	Luohu	3	DeepSkyBlue	Shenzhen Metro No.3 Line

  - The columns *line_color* and *operator* are not related to *station_id*
    - They are only related to *line_id*, which is only part of (a subset of) the primary key
  - Similarly, *english_name*, *chinese_name*, and *district* are not related to *line_id*
    - They are only related to *station_id*, which is only part of (a subset of) the primary key
  - 非主属性 *line_color*, *operator*, *english_name*, *chinese_name*, *district* 只依赖于主键中的部份属性

Second Normal Form (2NF)

- Example: Consider this table with the composite primary key (*station_id, line_id*)

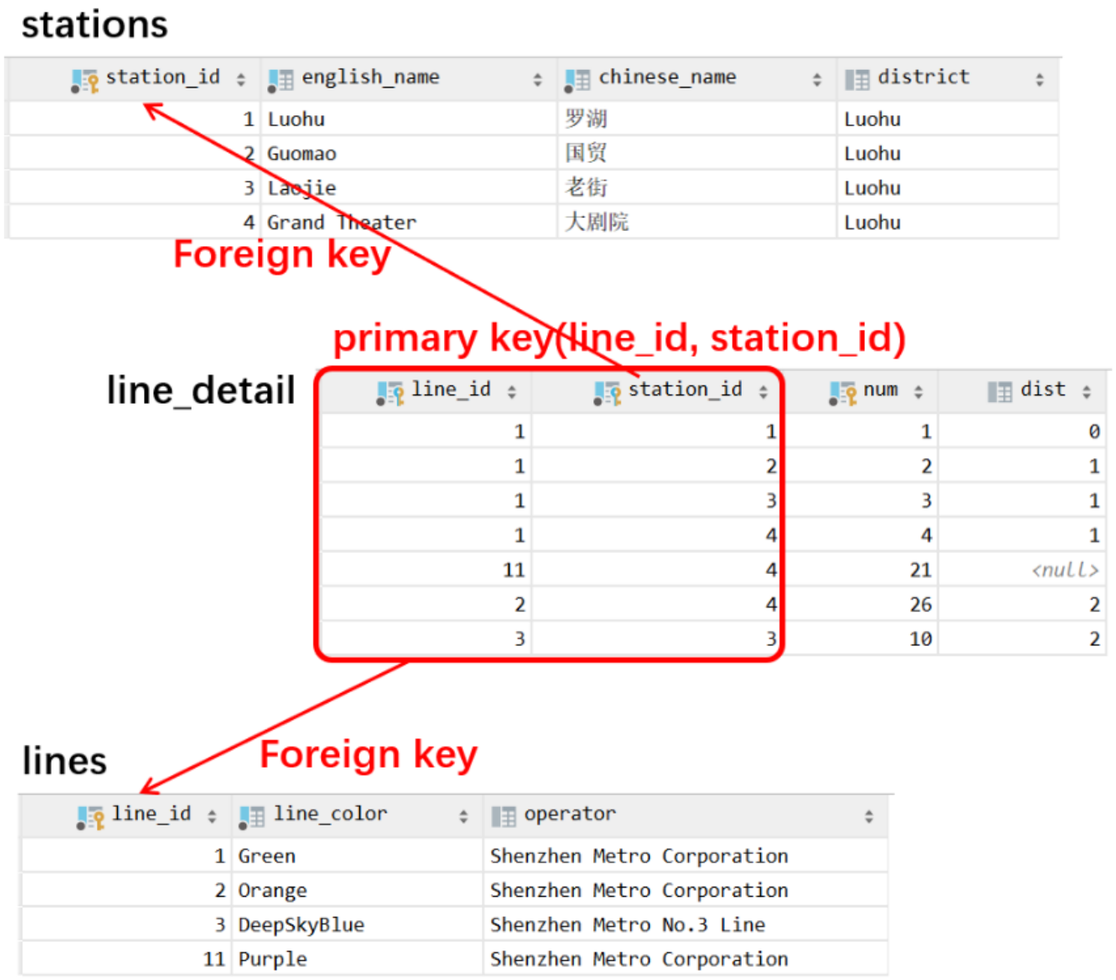
station_id	english_name	chinese_name	district	line_id	line_color	operator
1	Luohu	罗湖	Luohu	1	Green	Shenzhen Metro Corporation
2	Guomao	国贸	Luohu	1	Green	Shenzhen Metro Corporation
3	Laojie	老街	Luohu	1	Green	Shenzhen Metro Corporation
4	Grand Theater	大剧院	Luohu	1	Green	Shenzhen Metro Corporation
4	Grand Theater	大剧院	Luohu	11	Purple	Shenzhen Metro Corporation
4	Grand Theater	大剧院	Luohu	2	Orange	Shenzhen Metro Corporation
3	Laojie	老街	Luohu	3	DeepSkyBlue	Shenzhen Metro No.3 Line

  - Problem when not meeting 2NF: Insertion and deletion anomaly
    - We CANNOT insert a new station with no lines assigned yet (unless using NULLs)
    - If we delete a line, all stations associated with this line will be deleted as well

Second Normal Form (2NF)

- Fix it by
  - Splitting the two unrelated parts into two different tables of entities
  - And create a relationship set (if it is the many-to-many relationship between the two entities)

- By the way…
  - A relation with a single-attribute primary key is automatically in 2NF once it meets 1NF



Third Normal Form (3NF，第三范式)

- A relation satisfying 3NF must:
  - be in 2NF
  - all the attributes in a table are determined only by the candidate keys of that relation, not by any non-prime attributes
  - 所有属性 只依赖于 主键-候选键，不依赖于任意非主属性

Third Normal Form (3NF)

- Example: Consider this table which describes the bus lines and their stops
  - Primary key (*bus_line*)

bus_line	station_id	chinese_name	english_name	district
B796	21	鲤鱼门	Liyumen	Nanshan
M343	21	鲤鱼门	Liyumen	Nanshan
M349	21	鲤鱼门	Liyumen	Nanshan
M250	26	坪洲	Pingzhou	Bao'an
374	61	安托山	Antuo Hill	Futian
B733	61	安托山	Antuo Hill	Futian
B828	120	临海	Linhai	Nanshan

    - *station_id* depends on the primary key (*bus_line*)
    - However, the columns *chinese_name*, *english_name*, and *district* depend on *station_id*, which is not the primary key.
      - They only have "indirect/transitive" dependence （非直接/传递依赖） on the primary key
    - Problem: Data redundancy

Third Normal Form (3NF)

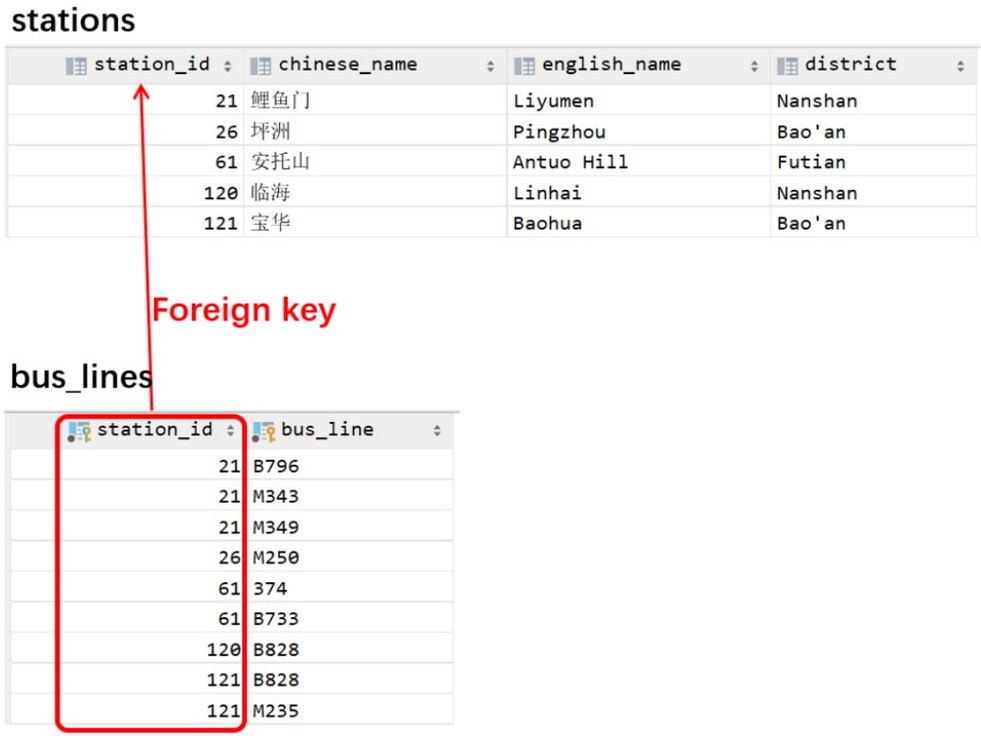
- Example: Consider this table which describes the bus lines and their stops
  - Primary key (*bus_line*)

bus_line	station_id	chinese_name	english_name	district
B796	21	鲤鱼门	Liyumen	Nanshan
M343	21	鲤鱼门	Liyumen	Nanshan
M349	21	鲤鱼门	Liyumen	Nanshan
M250	26	坪洲	Pingzhou	Bao'an
374	61	安托山	Antuo Hill	Futian
B733	61	安托山	Antuo Hill	Futian
B828	120	临海	Linhai	Nanshan

  - Problem when not meeting 3NF:
    - Data redundancy: as you can see in the table, the attributes for a station have been stored multiple times
    - Insertion and deletion anomaly: inserting a new bus line with no station becomes impossible without NULLs; deleting a station/bus line may also delete corresponding bus lines/stations.

Third Normal Form (3NF)

- Fix it by:
  - Create a new table with *station_id* as the primary key
    - i.e., the column which *chinese_name*, *english_name*, and *district* depend on
  - Move all columns which depend on the new primary key into the new table
    - … and, only leave the primary key of the new table (*station_id*) in the original table
  - (*In practice, if necessary) Add a foreign-key constraint
    - Not related to relational database modeling, only in implementations



Normalization

- In practice, we usually just satisfy 1NF, 2NF and 3NF

	UNF (1970)	1NF (1970)	2NF (1971)	3NF (1971)	EKNF (1982)	BCNF (1974)	4NF (1977)	ETNF (2012)	5NF (1979)	DKNF (1981)	6NF (2003)
Primary key (no duplicate tuples)[4]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Atomic columns (cells cannot have tables as values)[5]	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency either does not begin with a proper subset of a candidate key or ends with a prime attribute (no partial functional dependencies of non-prime attributes on candidate keys)[5]	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency either begins with a superkey or ends with a prime attribute (no transitive functional dependencies of non-prime attributes on candidate keys)[5]	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency either begins with a superkey or ends with an elementary prime attribute	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	N/A
Every non-trivial functional dependency begins with a superkey	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	N/A
Every non-trivial multivalued dependency begins with a superkey	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	N/A
Every join dependency has a superkey component[8]	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	N/A
Every join dependency has only superkey components	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	N/A
Every constraint is a consequence of domain constraints and key constraints	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Every join dependency is trivial	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

https://en.wikipedia.org/wiki/Database_normalization

Normalization

Every non key attribute must
provide a fact about the key,
the whole key,
and nothing but the key.

2NF

3NF

William Kent (1936 – 2005)

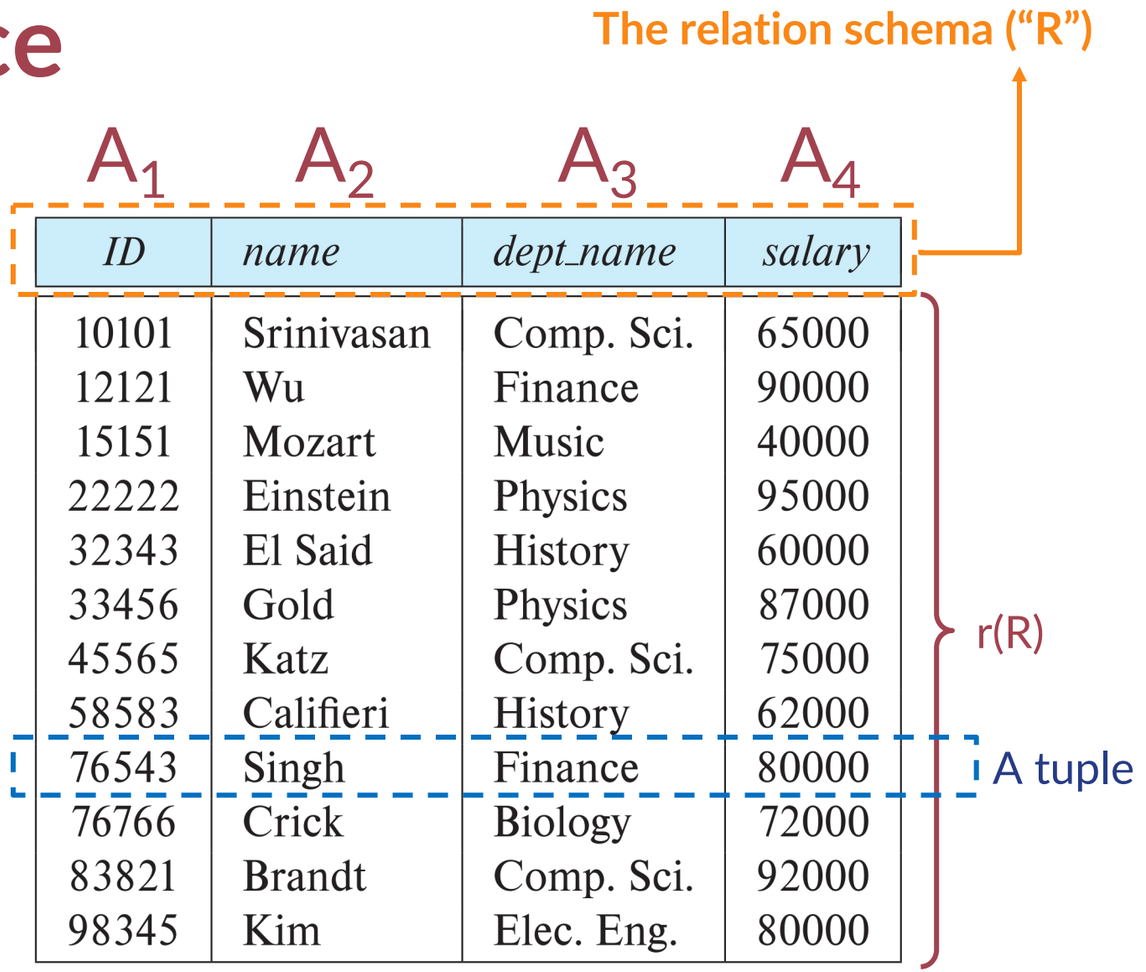
William Kent. "A Simple Guide to Five Normal Forms in Relational Database Theory", Communications of the ACM 26 (2), Feb. 1983, pp. 120–125.

(Recall) Prerequisites of Decomposition & Functional Dependency

Relation Schema and Instance

- A_1, A_2, …, A_n are attributes
- $R = (A_1, A_2, …, A_n)$ is a **relation schema**
  - Example on the right side:

    instructor = (*ID*, *name*, *dept_name*, *salary*)
- $r(R)$ denotes a relation instance r defined over schema R
  - Or to say, the entire table on the right side
- An element t of relation r is called a tuple
  - … and is represented by a row in a table



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

Relation Schema and Instance

- An analogy to programming languages:
  - Relation - Variables
  - Relation schema – Variable types
  - Relation instance – Value(s) stored in the variable

Database Schema

- Database schema is the logical structure of the database
  - It contains a set of relation schemas
  - … and a set of integrity constraints
- Database instance is a snapshot of the data in the database at a given instant in time

Keys

- Let K $\subseteq$ *R*
  - K is a **superkey** of *R* if values for K are sufficient to identify a unique tuple of each possible relation *r*(*R*)
    - E.g., {*ID*} and {*ID*, *name*} are both superkeys of instructor
    - If K is a superkey, any superset K' of K where K' $\subseteq$ *R* is a superkey as well
  - Superkey K is a **candidate key** if K is minimal, i.e., no proper subset of K is a superkey
    - E.g., {ID} is a candidate key for *instructor*
  - One of the candidate keys is selected to be the **primary key**
    - We mark the primary key with an underline:

      instructor = (*ID*, *name*, *dept_name*, *salary*)

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

instructor

Decomposition & Functional Dependency

Features of Good Relational Designs

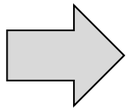
- Suppose we combine *instructor* and *department* into *in_dep*, which represents the natural join on the relations *instructor* and *department*
  - There is repetition of information (e.g., building and budget)
    - Could lead to inconsistency
  - Need to use nulls (if we add a new department with no instructors)
    - In many cases, null values are troublesome, as we saw in our study of SQL

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

instructor

dept_name	*building*	*budget*
Physics	Watson	70000
Finance	Painter	120000
History	Painter	50000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Biology	Watson	90000
Comp. Sci.	Taylor	100000
History	Painter	50000
Comp. Sci.	Taylor	100000
Music	Packard	80000
Physics	Watson	70000
Finance	Painter	120000

department



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

in_dep

Decomposition

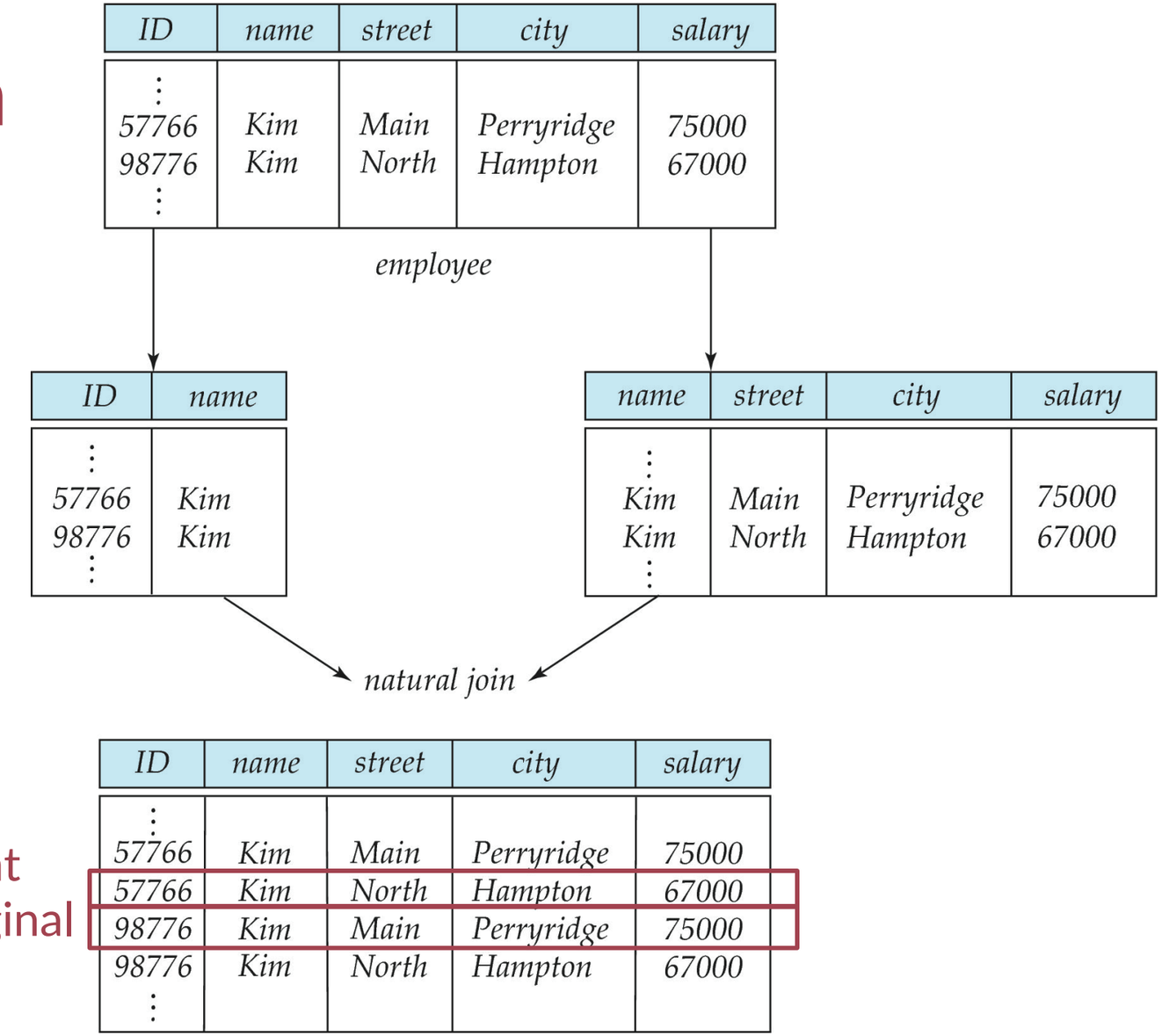
- Avoid the repetition-of-information problem
  - Decompose *in_dep* into two schemas: *instructor* and *department*

- However, not all decompositions are good
  - E.g., decompose *employee*(ID, name, street, city, salary) into:
    - employee1(ID, name)
    - employee2(name, street, city, salary)

  The problem arises when we have two employees with the same name

A Lossy Decomposition

- (Continue) we cannot reconstruct the original employee relation with the join operation
  - Unable to represent certain important facts about the university employee
  - We call it a **lossy decomposition**



Two “ghost” records that do NOT exist in the original table

Lossless Decomposition

- Let R be a relation schema and let R_1 and R_2 form a decomposition of R
  - That is, $R = R_1 \cup R_2$
  - The decomposition is a **lossless decomposition** if there is no loss of information by replacing R with the two relation schemas $R = R_1 \cup R_2$

- Formally, $\prod_{R_1}(r) \bowtie \prod_{R_2}(r) = r$
  - … and a decomposition is lossy if $r \subset \prod_{R_1}(r) \bowtie \prod_{R_2}(r)$

(!) proper subset

- Or to say, the two SQL queries on the right side generate identical results:

```
select * -- 1
from (select R1 from r)
    natural join
     (select R2 from r);

select * from R; -- 2
```

Normalization Theory

- Decide whether a particular relation *R* is in "good" form
- In the case that a relation *R* is not in "good" form, decompose it into a set of relations $\{R_1, R_2, ..., R_n\}$ such that
  - Each relation is in good form
  - The decomposition is a lossless decomposition

- Our theory is based on:
  - Functional dependencies
  - * Multivalued dependencies (self study)
- Generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily

Functional Dependencies

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

- For example, some of the constraints that are expected to hold in a university database are:
  - 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
    - E.g., an instructor can have multiple departments, a student can have double major
    - But we assume that they can only have one
  - Each department has only one value for its budget, and only one associated building

Functional Dependencies

- 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
- Constraints on the set of legal relations
  - Require that the value for a certain set of attributes determines uniquely the value for another set of attributes
- A functional dependency is a generalization of the notion of a key
- Functional dependencies allow us to express constraints that we cannot express with superkeys

Definition of Functional Dependencies

- Let R be a relation schema, and $\alpha \subseteq R$ *and* $\beta \subseteq R$,

  the functional dependency

  $$\alpha \rightarrow \beta$$

  holds on R if and only if for any legal relations $r(R)$, whenever any two tuples t_1 and t_2 of r agree on the attributes α, they also agree on the attributes β.

  That is,

  $$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- Example: Consider $r(A, B)$ with the following instance of r,
  - On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold

A	B
1	4
1	5
3	7

Closure (闭包) of a Set of Functional Dependencies

- Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F:
  - If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$

- The set of all functional dependencies logically implied by F is the **closure** of F
  - We denote the closure of F by F^+
  - F^+ is a superset of F

Keys and Functional Dependencies

- Let's see how can we (re)define the concept of "**keys**" under the language of functional dependencies

Keys and Functional Dependencies

- K is a superkey for relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if
  - $K \rightarrow R$, and
  - for **no** $\alpha \subset K$, $\alpha \rightarrow R$

(!) proper subset, again

Keys and Functional Dependencies

- Functional dependencies allow us to express constraints that cannot be expressed using superkeys

- E.g. Consider the schema: *inst_dept*(*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)
  - We expect these functional dependencies to hold:

    dept_name*→ *building*, *ID* → *building

  … but would not expect the following to hold:

  dept_name* → *salary

Remember the constraints we had earlier

- For example, some of the constraints that are expected to hold in a university database are:
  - 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
    - E.g., an instructor can have multiple departments, a student can have double major
    - But we assume that they can only have one
  - Each department has only one value for its budget, and only one associated building

Use of Functional Dependencies

- We use functional dependencies to
  - To test relations to see if they are legal under a given set of functional dependencies
    - If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* satisfies *F*
    - For each functional dependency $\alpha \rightarrow \beta$, for all pairs of tuples t1 and t2 in the instance, $t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$
  - To specify constraints on the set of legal relations
    - If all legal relations on *R* satisfy the set of functional dependencies *F*, we say that *F* holds on *R*

Use of Functional Dependencies

- Example: List some functional dependencies that the table satisfies

A	B	C	D
a1	b1	c1	d1
a1	b2	c1	d2
a2	b2	c2	d2
a2	b3	c2	d3
a3	b3	c2	d4

Use of Functional Dependencies

- Example: List some functional dependencies that the table satisfies
  - $A \rightarrow C$ (but C $\rightarrow$ A is not satisfied)
  - $D \rightarrow B$

Can you find more?

A	B	C	D
a1	b1	c1	d1
a1	b2	c1	d2
a2	b2	c2	d2
a2	b3	c2	d3
a3	b3	c2	d4

Use of Functional Dependencies

- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.

- Example: we see that *room_number* → *capacity* is satisfied.
  - However, in real world, two classrooms in different buildings can have the same room number but with different room capacity
  - We prefer {*building*, *room_number*} → *capacity*

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Trivial Functional Dependencies

- A functional dependency is **trivial** if it is satisfied by all relations

- Example:
  - *ID*, *name* → *ID*
  - *name* → *name*

- In general,
  - $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Lossless Decomposition

- We can use functional dependencies to show when certain decomposition are lossless
  - For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \prod_{R1}(r) \bowtie \prod_{R2}(r)$$

  - A decomposition of R into R_1 and R_2 is a lossless decomposition if at least one of the following dependencies is in F^+:
    - $R_1 \cap R_2 \rightarrow R_1$
    - $R_1 \cap R_2 \rightarrow R_2$

In other words, if $R_1 \cap R_2$ forms a superkey for either R_1 or R_2, the decomposition of R is a lossless decomposition

Lossless Decomposition

- Example:
  - *in_dep (ID, name, salary, dept_name, building, budget)*
  - … and the decomposed schemas, *instructor* and *department*:
    - *instructor(ID, name, dept_name, salary)*
    - *department(dept_name, building, budget)*

$instructor \cap department = dept\_name$

$dept\_name \rightarrow dept\_name, building, budget$

(… which means the decomposition is lossless)

Lossless Decomposition

- Another example:

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Lossless decomposition:

    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
- $R_1 = (A, B), \quad R_2 = (A, C)$
  - Lossless decomposition:

    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
- *Note:*
  - $B \rightarrow BC$

    is a shorthand notation for
  - $B \rightarrow \{B, C\}$

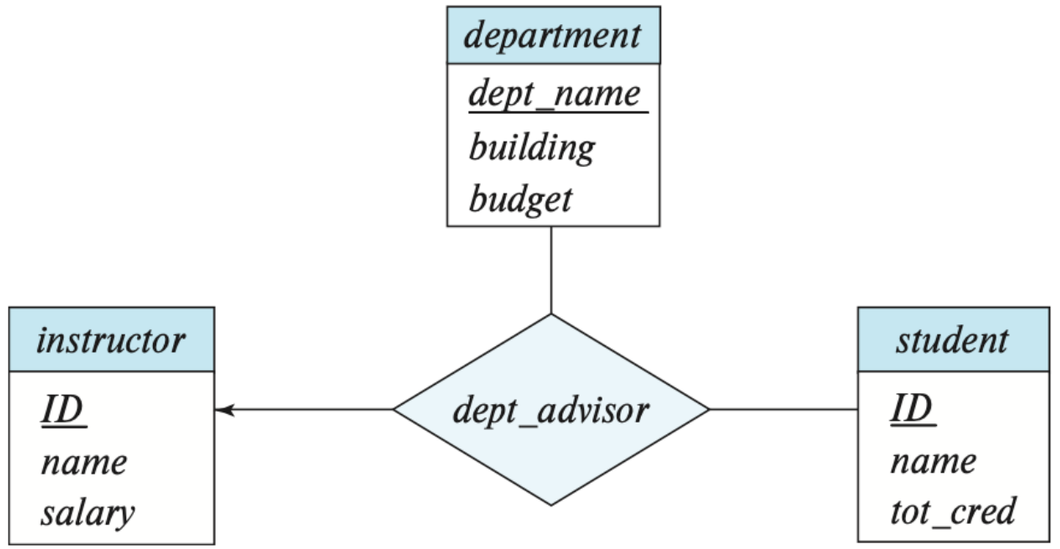
Dependency Preservation

- Testing functional dependency constraints each time the database is updated can be costly
  - It is useful to design the database in a way that constraints can be tested efficiently
  - After decomposition, some functional dependency may involve multiple relation schemas and cannot be tested efficiently (as join is required)

- If a functional dependency in the original relation *R* does not exist in any of the decomposed relations, we say it is not dependency-preserving
  - In the dependency preservation, every dependency must be satisfied in at least one decomposed table

Dependency Preservation

- Consider a new E-R design for relationships between students, instructors, and departments
  - An instructor can only be associated with one department
  - A student can have multiple advisors but not more than one from a given department
    - Think about double-major students



Dependency Preservation

- *instructor*, *department*, and *student* schemas unchanged
- Consider a schema
  - *dept_advisor(s_ID, i_ID, dept_name)*
  - … with function dependencies: (1) *i_ID* → *dept_name* (2) *s_ID*, *dept_name* → *i_ID*
  - *(1) follows "an instructor can act as an advisor for only one department"*
  - *(2) follows "a student may have at most one advisor for a given department"*

In this design, we are forced to repeat the department name once for each time an instructor participates in a *dept_advisor* relationship.

Dependency Preservation

- *instructor*, *department*, and *student* schemas unchanged
- Consider a schema
  - *dept_advisor(s_ID, i_ID, dept_name)*
  - … with function dependencies: (1) $i\_ID \rightarrow dept\_name$ (2) $s\_ID, dept\_name \rightarrow i\_ID$
  - *(1) follows "an instructor can act as an advisor for only one department"*
  - *(2) follows "a student may have at most one advisor for a given department"*

In this design, we are forced to repeat the department name once for each time an instructor participates in a *dept_advisor* relationship.

- To fix this problem, we need to decompose *dept_advisor*
  - However, any decomposition will not include all the attributes in

    $s\_ID, dept\_name \rightarrow i\_ID$
  - Thus, the decomposition will NOT be dependency-preserving
  - Redundancy reduction and dependency preservation cannot be achieved at the same time

Dependency Preservation

- Problem when not meeting dependency preservation
  - Every time the database wants to check the integrity of the functional dependency

    $$s\_ID, dept\_name \rightarrow i\_ID,$$

    the decomposed tables must be joined

  - … where the computational cost could be very high with join operations

BCNF and 3NF

Normal Forms: Revisited

- Boyce-Codd Normal Form (BCNF)
- 3NF
- Higher-order normal forms

Boyce-Codd Normal Form

- A relation schema *R* is in **BCNF** with respect to a set *F* of functional dependencies if for all functional dependencies in F^+ of the form

$$\alpha \rightarrow \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for R

- * A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF

Boyce-Codd Normal Form

- Example schema that is **not** in BCNF:
  - *in_dep (ID, name, salary, dept_name, building, budget)*

  Because,

  $$dept\_name \rightarrow building, budget$$

  - holds in *in_dep*, however, *dept_name* is not a superkey
    - … where {ID, dept_name} is

  - When decompose *in_dept* into *instructor* and *department*
    - *instructor* is in BCNF
      - *instructor (ID, name, dept_name, salary)*
      - No functional dependencies without having ID on the left, except trivial ones
    - *department* is in BCNF
      - *department (dept_name, building, budget)*

Decomposing a Schema into BCNF

- Let *R* be a schema *R* that is not in BCNF
- Let $\alpha \rightarrow \beta$ be the functional dependency that causes a violation of BCNF
  - We decompose *R* into:
    - $(\alpha \cup \beta)$
    - $(R - (\beta - \alpha))$
  - Keep decomposing until all relation schemas are in BCNF
  - Notice that $(\alpha \cup \beta) \cap (R - (\beta - \alpha)) = \alpha$, and $\alpha \rightarrow \beta$, so lossless decomposition

- Example: *in_dep (<u>ID</u>, name, salary, <u>dept_name</u>, building, budget)*
  - α = *dept_name*, β = *building, budget*
  - Thus, *in_dep* is replaced by:
    - $(\alpha \cup \beta)$ = (*dept_name, building, budget*)
    - $(R - (\beta - \alpha))$ = (*ID, name, dept_name, salary*)

Decomposing a Schema into BCNF

- Another example:
  - $R = (A, B, C)$
    $F = \{A \rightarrow B, B \rightarrow C\}$
  - $R_1 = (A, B)$, $R_2 = (B, C)$
    - Lossless-join decomposition:

      $$R_1 \cap R_2 = \{B\} \quad \text{and } B \rightarrow BC$$

    - Dependency preserving
  - $R_1 = (A, B)$, $R_2 = (A, C)$
    - Lossless-join decomposition:

      $$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

    - Not dependency preserving
      (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

BCNF and Dependency Preservation

- It is not always possible to achieve both BCNF and dependency preservation
- Consider the schema (that we have visited before)
  - *dept_advisor(s_ID, i_ID, dept_name)*
    - with function dependencies: (1) *i_ID* → *dept_name* (2) *s_ID*, *dept_name* → *i_ID*
    - *(1) follows "an instructor can act as an advisor for only one department"*
    - *(2) follows "a student may have at most one advisor for a given department"*
    - Superkeys {*s_ID*, *i_ID*}, {*s_ID*, *dept_name*}
  - *dept_advisor* is not in BCNF since for *i_ID* → *dept_name*, i_ID is not a superkey
    - Although (2) satisfies the conditions of BCNF

BCNF and Dependency Preservation

- It is not always possible to achieve both BCNF and dependency preservation
- Consider the schema (that we have visited before)
  - *dept_advisor(s_ID, i_ID, dept_name)*
    - with function dependencies: (1) $i\_ID \rightarrow dept\_name$ (2) $s\_ID, dept\_name \rightarrow i\_ID$
    - Superkeys {*s_ID*, *i_ID*}, {*s_ID*, *dept_name*}
  - *dept_advisor* is not in BCNF since for $i\_ID \rightarrow dept\_name$, i_ID is not a superkey
- To fix this problem, we need to decompose *dept_advisor* into
  - *(s_ID, i_ID)* and (*i_ID*, *dept_name)*, both in BCNF
  - However, the decomposition will **not** include all the attributes in (2)
  - Thus, the decomosition will NOT be dependency-preserving
    - Can check (1) without any join
    - But checking (2) requires computing the join of the decomposed relations

Third Normal Form (3NF)

- Relax the constraints of BCNF to ensure dependency preservation
- A relation schema R is in **third normal form (3NF)** if for all

$$\alpha \rightarrow \beta \text{ in } F^+$$

at least one of the following holds:

  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for R
  - Each attribute A in $\beta - \alpha$ is contained in a candidate key for R
- Notes
  - Each attribute A may be in a different candidate key
  - If a relation is in BCNF, it is in 3NF (... since in BCNF, one of the first two conditions above must hold)
  - The third condition above is a <u>minimal relaxation of BCNF</u> to ensure dependency preservation
    - It can ensure that every schema has a dependency-preserving decomposition into 3NF
    - Seems unintuitive. It will become more clear later when we talk about decomposition to 3NF

3NF Example

- Consider the schema (that we have visited before)
  - *dept_advisor(s_ID, i_ID, department_name)*
  - … with function dependencies: (1) *i_ID* $\rightarrow$ *dept_name* (2) *s_ID*, *dept_name* $\rightarrow$ *i_ID*

  - We have two candidate keys: {*s_ID*, *dept_name*} and {*s_ID*, *i_ID*}

- *dept_advisor* is not in BCNF, but it can be in 3NF
  - {s_ID, dept_name} is a superkey
  - *i_ID* $\rightarrow$ *dept_name* and *i_ID* is NOT a superkey (which violates BCNF), but:
    - α is *i_ID*, β is *dept_name*
    - {*dept_name*} – {*i_ID*} = {*dept_name*}
    - *dept_name* is contained in a candidate key (-> {*s_ID*, *dept_name*})

Comparison of BCNF and 3NF

- Advantages to 3NF over BCNF
  - It is always possible to obtain a 3NF design without sacrificing *losslessness* or *dependency preservation*
- Disadvantages to 3NF
  - We may have to use nulls to represent some of the possible meaningful relationships among data items
  - There is a problem of potential repetition of information

Goals of Normalization

- Let *R* be a relation scheme with a set *F* of functional dependencies
  - Decide whether a relation scheme *R* is in “good” form.
  - In the case that a relation scheme R is not in “good” form, need to decompose it into a set of relation scheme $\{R_1, R_2, ..., R_n\}$ such that:
    - Each relation scheme is in good form
    - The decomposition is a lossless decomposition
    - *Preferably*, the decomposition should be dependency preserving

Principles of Database Systems (CS307)

Lecture 11: Normalization - Part 2

Zhong-Qiu Wang

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

Functional Dependency Theory

Closures, Attribute Closures, Canonical Cover, and Dependency Preservation

It is strongly recommended to read Section 7.4 of the reference textbook “Database System Concepts, 7th Edition” for more details about the functional dependency theory

Functional Dependency Theory Roadmap

- We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies

Computing Closure F^+

- We then develop algorithms to generate lossless decompositions into BCNF and 3NF

Decomposition into BCNF and 3NF

- Furthermore, we develop algorithms to test if a decomposition is dependency-preserving

Testing whether decomposition is dependency-preserving

(Recall) Closure of a Set of Functional Dependencies

- Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F:
  - For example, if $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$

- The set of all functional dependencies logically implied by F is the **closure** of F
  - We denote the closure of F by F^+
  - F^+ is a superset of F

Computing Closure with Armstrong's Axioms

- We can compute F^+, the closure of F, by repeatedly applying **Armstrong's Axioms**:
  - Reflexive rule: if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$
  - Augmentation rule: if $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$
  - Transitivity rule: if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$

- Greek letters (α, β, γ) represent *sets of attributes*
- "$\gamma\alpha$" means "γ U α"

- These rules are sound and complete
  - *Sound* （可靠） : Generate only functional dependencies that actually hold
  - *Complete* （完备） : Generate all functional dependencies that hold

Computing Closure with Armstrong's Axioms

- We can compute F^+, the closure of F, by repeatedly applying **Armstrong's Axioms**:
  - Reflexive rule: if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$
  - Augmentation rule: if $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$
  - Transitivity rule: if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$

- Greek letters (α, β, γ) represent *sets of attributes*
- "$\gamma\alpha$" means "γ U α"

- These rules are sound and complete
  - *Sound* （可靠）: Generate only functional dependencies that actually hold
  - *Complete* （完备）: Generate all functional dependencies that hold

However, it is difficult and tiresome to use them for deriving F^+

Computing Closure with Armstrong's Axioms

- Additional rules:
  - Union rule: If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\,\gamma$ holds
  - Decomposition rule: If $\alpha \rightarrow \beta\,\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds
  - Pseudotransitivity rule: If $\alpha \rightarrow \beta$ holds and $\gamma\,\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds
    - From $\alpha \rightarrow \beta$, we have $\gamma\alpha \rightarrow \gamma\beta$, together with $\gamma\,\beta \rightarrow \delta$, we have $\gamma\alpha \rightarrow \delta$

The above rules can be inferred from Armstrong's axioms

Procedure for Computing F^+

$F^+ = F$
apply the reflexivity rule /* Generates all trivial dependencies */
repeat
    for each functional dependency f in F^+
        apply the augmentation rule on f
        add the resulting functional dependencies to F^+
    for each pair of functional dependencies f_1 and f_2 in F^+
        if f_1 and f_2 can be combined using transitivity
            add the resulting functional dependency to F^+
until F^+ does not change any further

- Problem: The target F^+ can be very lengthy
  - For $\alpha \rightarrow \beta$, there may be 2^n possible values for α and 2^n for β
  - We will introduce other ways of computing F^+ later

Closure of Attribute Sets （属性集闭包）

- We say that an attribute *B* is functionally determined by α if $\alpha \rightarrow B$

- Given a set of attributes α, define the **closure** of α under *F* (denoted by α^+) as the set of attributes that are functionally determined by α under *F*

Algorithm to compute α^+, the closure of α under *F*

```
result := α;        // by reflexibility rule, α → result
repeat
        for each functional dependency β → γ in F do
            begin
                if β ⊆ result then result := result ∪ γ;
            end
until (result does not change)
```

// based on $\alpha \rightarrow$ *result*, $\beta \rightarrow \gamma$ and $\beta \subseteq$ *result*
// *result* $\cup \beta \rightarrow$ *result* $\cup \gamma$, by augmentation rule
// *result* $\rightarrow$ *result* $\cup \gamma$
// $\alpha \rightarrow$ *result* $\cup \gamma$

Example of Attribute Set Closure

- ***Given***:

  $R = (A, B, C, G, H, I)$

  $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$

- What is $(AG)^+$?
  1. $result = AG$
  2. $result = ABG$ $(A \rightarrow B)$
  3. $result = ABCG$ $(A \rightarrow C)$
  4. $result = ABCGH$ $(CG \rightarrow H$ and $CG \subseteq ABCG)$
  5. $result = ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq ABCGH)$

Example of Attribute Set Closure

- ***Given***:

  $R = (A, B, C, G, H, I)$

  $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$

- What is $(AG)^+$?
  1. *result* = AG
  2. *result* = ABG $(A \rightarrow B)$
  3. *result* = $ABCG$ $(A \rightarrow C)$
  4. *result* = $ABCGH$ $(CG \rightarrow H$ and $CG \subseteq ABCG)$
  5. *result* = $ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq ABCGH)$

Further Questions:

Is *AG* a candidate key?

1. Is AG a super key?
   1. Does $AG \rightarrow R$? == Is $R \supseteq (AG)^+$
2. Is any subset of AG a superkey?
   1. Does $A \rightarrow R$? == Is $R \supseteq (A)^+$
   2. Does $G \rightarrow R$? == Is $R \supseteq (G)^+$
   3. In general: check for each subset of size *n-1*

Use of Attribute Closures

There are several uses of the attribute closure algorithm

- Testing for superkey
  - To test if α is a superkey, we compute α^+ and check if α^+ contains all attributes of R
- Testing functional dependencies
  - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$
  - That is, we compute α^+ by using attribute closure, and then check if it contains β
  - It is a simple and cheap test, and very useful
- Computing closure of F
  - For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$

Canonical Cover（正则 /最小覆盖）

- Suppose that we have a set of functional dependencies *F* on a relation schema
  - Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies
  - … that is, all the functional dependencies in *F* are satisfied in the new database state
  - If an update violates any functional dependencies in the set *F*, the system must roll back the update

Canonical Cover（正则/最小覆盖）

- We can reduce the effort spent in checking for violations by testing *a simplified set* of functional dependencies that has the same closure as the given set
  - Any database satisfying the simplified set of functional dependencies also satisfies the original set, and vice versa, since the two sets have the same closure
  - This *simplified set* is termed the **canonical cover**

To define **canonical cover**, we must first define **extraneous attributes**

- An attribute of a functional dependency in F is extraneous if we can remove it without changing F^+

Extraneous Attributes (Stronger Constraint vs. Weaker Constraint)

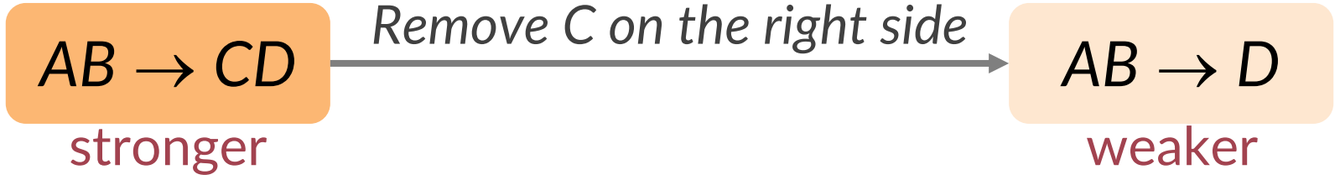
- Removing an attribute from the left side of a functional dependency could make it a stronger constraint

$A \rightarrow C$ is stronger than $AB \rightarrow C$ because $A \rightarrow C$ logically implies $AB \rightarrow C$

- However, depending on what our set F of functional dependencies happens to be, we may be able to remove B from $AB \rightarrow C$ safely
  - E.g., $F = \{AB \rightarrow C, A \rightarrow D, D \rightarrow C\}$

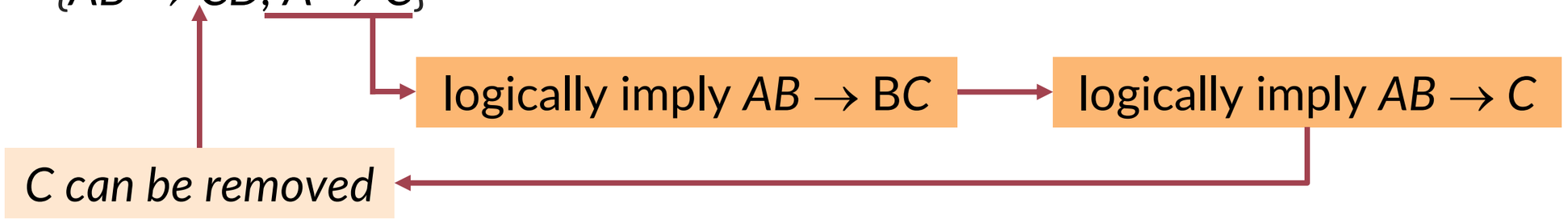
Extraneous Attributes (Stronger Constraint vs. Weaker Constraint)

- In reverse, removing an attribute from the right side of a functional dependency could make it a weaker constraint



$AB \rightarrow CD$ is stronger than $AB \rightarrow D$ because $AB \rightarrow D$ cannot logically imply $AB \rightarrow C$

- However, depending on what our set F of functional dependencies happens to be, we may be able to remove C from $AB \rightarrow CD$ safely
  - E.g., $F = \{AB \rightarrow CD, A \rightarrow C\}$



Extraneous Attributes

- An attribute of a functional dependency in F is **extraneous** if we can remove it without changing F^+

- Consider a set F of functional dependencies and the functional dependency

$\alpha \rightarrow \beta$ in F

Remove from the Left Side

- Attribute A is **extraneous** in α if
  - $A \in \alpha$, and
  - F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$

Remove from the Right Side

- Attribute A is **extraneous** in β if
  - $A \in \beta$, and
  - The set of functional dependency $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ logically implies F

Testing if an Attribute is Extraneous

- Let *R* be a relation schema; *F* be a set of functional dependencies held on *R*
- Consider an attribute in the functional dependency $\alpha \rightarrow \beta$

To test if attribute $A \in \alpha$ is extraneous in α

- Let $\gamma = \alpha - \{A\}$, check if $\gamma \rightarrow \beta$ can be inferred from *F*
  - Compute γ^+ using the dependencies in F
  - If γ^+ includes all attributes in β, *A* is extraneous in α

To test if attribute $A \in \beta$ is extraneous in β

- Consider the set:
  $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$
- Check that α^+ contains *A*
  - if it does, *A* is extraneous in β

Examples of Extraneous Attributes

- Let $F = \{AB \rightarrow CD, A \rightarrow E, E \rightarrow C\}$

- To check if C is extraneous in $AB \rightarrow CD$
  - Compute the attribute closure of AB under $F' = \{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$
  - The closure is $ABCDE$, which includes CD
  - This implies that C is extraneous

To test if attribute $A \in \beta$ is extraneous in β

- Consider the set:
  $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$
- Check that α^+ contains A
  - if it does, A is extraneous in β

Canonical Cover

- A **canonical cover** for F is a set of dependencies F_c such that
  - F logically implies all dependencies in F_c , and
  - F_c logically implies all dependencies in F, and

  - No functional dependency in F_c contains an extraneous attribute, and

  - Each left side of functional dependency in F_c is unique.

    ... that is, there are no two dependencies in F_c
    - $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ such that
    - $\alpha_1 = \alpha_2$

Canonical Cover

- To compute a canonical cover for *F*:

$F_c = F$
repeat
    Use the union rule to replace any dependencies in F_c of the form
        $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$.
    Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an extraneous
        attribute either in α or in β.
        /* Note: the test for extraneous attributes is done using F_c, not F */
    If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$ in F_c.
until (F_c does not change)

Example: Computing a Canonical Cover

Given:

$R = (A, B, C)$

$F = \{A \rightarrow BC$
$B \rightarrow C$
$A \rightarrow B$
$AB \rightarrow C\}$

- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
  - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
    - Yes: in fact, $B \rightarrow C$ is already present!
  - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
  - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
    - Yes: using transitivity on $A \rightarrow B$ *and* $B \rightarrow C$

The resulting canonical cover:

$\{A \rightarrow B, B \rightarrow C\}$

Dependency Preservation

- Let F_i be the set of dependencies in F^+ that include only attributes in R_i
  - A decomposition is **dependency preserving**, if

$$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$$

  - Using the above definition, testing for dependency preservation take exponential time
    - A faster algorithm is introduced in Section 7.4.4 of the reference book

- If a decomposition is NOT dependency-preserving, checking updates for violation of functional dependencies may require computing joins, which is expensive

Functional Dependency Theory

Algorithms for BCNF and 3NF Decompositions Using Functional Dependencies

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
  - compute α^+ (the attribute closure of α), and
  - verify that it includes all attributes of *R*
    - ... that is, it is a superkey of *R*

- **Simplified Test**: To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F* for violation of BCNF, rather than checking all dependencies in F^+
  - If none of the dependencies in *F* causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF, either
  - * However, simplified test using only F is incorrect when testing a relation in a decomposition of R

BCNF Decomposition Algorithm

```
result := {R};
done := false;
while (not done) do
    if (there is a schema R_i in result that is not in BCNF)
      then begin
                let α → β be a nontrivial functional dependency that holds
                on R_i such that α+ does not contain R_i and α ∩ β = ∅;
                result := (result − R_i) ∪ (R_i − β) ∪ ( α, β);
            end
      else done := true;
```

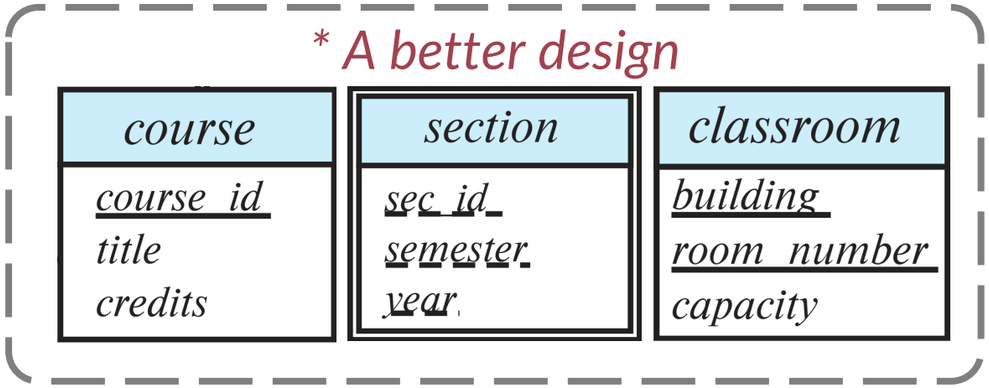
*Note: each R_i is in BCNF, and decomposition is lossless-join

Example of BCNF Decomposition

- **Schema**
  - *class* (*course_id*, *title*, *dept_name*, *credits*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)

- **Candidate key**
  - {*course_id*, *sec_id*, *semester*, *year*}

- **Functional dependencies**
  - *course_id* → {*title*, *dept_name*, *credits*}
  - {*building*, *room_number*} → *capacity*
  - {*course_id*, *sec_id*, *semester*, *year*} → {*building*, *room_number*, *time_slot_id*}



Example of BCNF Decomposition

- BCNF Decomposition (round 1)
  - *course_id*→ *title*, *dept_name*, *credits* holds
    - but *course_id* is not a superkey
  - We replace *class* by:
    - *course*(*course_id*, *title*, *dept_name*, *credits*)
    - *class-1* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)

    Here, *course* is in BCNF

Example of BCNF Decomposition

- BCNF Decomposition (round 2)
  - *building*, *room_number*→*capacity* holds on *class-1*
    - but {*building*, *room_number*} is not a superkey for *class-1*
  - We replace *class-1* by:
    - *classroom* (*building*, *room_number*, *capacity*)
    - *section* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)

    classroom and *section* are in BCNF

Third Normal Form (3NF)

- There are some situations where
  - BCNF is not dependency preserving, and
  - Efficient checking for FD violation on updates is important

- Solution: define a weaker normal form, called Third Normal Form (3NF)
  - Allows some redundancy, but functional dependencies can be checked on individual relations without computing a join.
  - There is always a lossless-join, dependency-preserving decomposition into 3NF

3NF Decomposition Algorithm

- The algorithm ensures
  - Each relation schema R_i is in 3NF
  - Decomposition is dependency preserving and lossless-join

let F_c be a canonical cover for F;
$i := 0$;
for each functional dependency $\alpha \rightarrow \beta$ in F_c
    $i := i + 1$;
    $R_i := \alpha\beta$;
if none of the schemas $R_j, j = 1, 2, \ldots, i$ contains a candidate key for R
  then
    $i := i + 1$;
    $R_i :=$ any candidate key for R;
/* Optionally, remove redundant relations */
repeat
    if any schema R_j is contained in another schema R_k
      then
        /* Delete R_j */
        $R_j := R_i$;
        $i := i - 1$;
until no more R_js can be deleted
return $(R_1, R_2, \ldots, R_i)$

Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
  - The decomposition is lossless
  - The dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
  - The decomposition is lossless
  - It may not be possible to preserve dependencies

An alternative method of generating a BCNF design: First use the 3NF algorithm. Then, for any schema in the 3NF design that is not in BCNF, decompose using the BCNF algorithm. If the result is not dependency-preserving, revert to the 3NF design.

Other Normal Forms

How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a relation *inst_info*(*ID*, *child_name*, *phone*)
  - … where an instructor may have more than one phone and can have multiple children
    - Actually, we would better use two relations: (*ID*, *child_name*) and (*ID*, *phone*)

An instance of *inst_info*:

(99999, David, 512-555-1234)
(99999, David, 512-555-4321)
(99999, William, 512-555-1234)
(99999, William, 512-555-4321)

How good is BCNF?

- inst_info is in BCNF
  - *Primary key is {ID, child_name, phone}*
  - *ID → child_name ? No*
  - *ID → phone? No*
  - No non-trivial functional dependencies exist, except trivial ones

- However,
  - Insertion anomalies
    - If we add a phone number 981-992-3443 to the instructor 99999, we need to add two tuples:

      (99999, David, 981-992-3443)
      (99999, William, 981-992-3443)

    - If we only add one of the two tuples above, it will imply that only David or William corresponds to 981-992-3443, which is not the functional dependency we need to keep

Fourth Normal Form (4NF)

- It is better to decompose *inst_info* into *inst_child* and *inst_phone*:

ID	*child_name*
99999	David
99999	William

ID	*phone*
99999	512-555-1234
99999	512-555-4321

  - This suggests a need for higher normal forms, such as Fourth Normal Form (4NF) that resolves such kind of multivalued dependencies

Wait, Where are 1NF and 2NF?

- 1NF is about attribute domains but not decompositions
  - … and hence not quite related to dependencies we have learned in this section

Wait, Where are 1NF and 2NF?

- 2NF: Partial dependency
  - A functional dependency $\alpha \rightarrow \beta$ is called a partial dependency if there is a proper subset γ of α such that $\gamma \rightarrow \beta$
    - We say that β is partially dependent on α
  - A relation schema *R* is in second normal form (2NF) if each attribute *A* in *R* meets one of the following criteria:
    - It appears in a candidate key
    - It is not partially dependent on a candidate key

Wait, Where are 1NF and 2NF?

- 2NF: Partial dependency
  - You can try to prove that a relation meeting 3NF also satisfies 2NF
    - Exercise 7.19 in “Database System Concepts, 7th Edition”
  - In practice, we usually choose to satisfy 3NF or BCNF

Summary for Database Design

Overall Database Design Process

- We have assumed schema *R* is given
  - *R* could have been generated when converting E-R diagram to a set of tables
  - *R* could have been a single relation containing all attributes that are of interest (called universal relation)
  - Normalization breaks *R* into smaller relations
  - *R* could have been the result of some ad-hoc design of relations, which we then test/convert to normal form

E-R Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
  - However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity
  - Example: an *employee* entity with attributes *department_name* and *building*
    - … but with functional dependency: *department_name* $\rightarrow$ *building*
    - Good design would have made department an entity

Denormalization for Performance

- We may want to use non-normalized schemas for better performance
- For example, displaying *prereqs* along with *course_id*, and *title* requires join of *course* with *prereq*
  - Alternative 1: Use denormalized relation containing attributes of *course* as well as *prereq* with all above attributes
    - faster lookup
    - extra space and extra execution time for updates
    - extra coding work for programmer and possibility of error in extra code
  - Alternative 2: use a materialized view defined on *course* ⋈ *prereq*
    - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

Other Design Issues

- Some aspects of database design are not caught by normalization
  - Examples of bad database design, to be avoided: Instead of *earnings* (*company_id*, *year*, *amount*), use
    - *earnings_2004*, *earnings_2005*, *earnings_2006*, etc., all on the schema (*company_id*, *earnings*).
      - Above are in BCNF, but make querying across years difficult and needs new table each year
    - *company_year* (*company_id*, *earnings_2004*, *earnings_2005*, *earnings_2006*)
      - Also in BCNF, but also makes querying across years difficult and requires new attribute each year
      - It is an example of a **crosstab**, where values for one attribute become column names
      - Such crosstabs are widely used in spreadsheets and data analysis tools