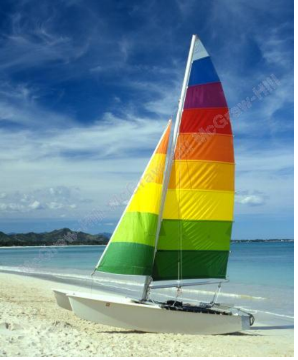


Part 2: Database Theory

ECE 656
University of Waterloo
Dr. Paul A.S. Ward

Acknowledgment: slides derived from Dr. Wojciech Golab

based on materials provided by
Silberschatz, Korth, and Sudarshan, copyright 2010
(source: www.db-book.com)

Learning Outcomes

- Relational Algebra and Relational Calculus
  - Non-procedural query languages
  - Core operators: σ, $\prod$, $\cup$, $-$, x, ρ
  - Optional operators: $\cap$, $\leftarrow$, $\bowtie$, $\bowtie_{\theta}$, $\div$
  - Null values and 3-valued logic
- Requirements for good database design
- Functional dependencies
- Normal forms

- Textbook sections (6th ed.): Chapter 6, 8.1-8.5

Domain Relational Calculus

- A non-procedural (declarative) query language
- Each query is an expression of the form:

$$\{ < x_1, x_2, \ldots, x_n > \mid P\,(x_1, x_2, \ldots, x_n)\}$$

- $x_1, x_2, \ldots, x_n$ represent domain variables
  - *i.e.,* they are constrained to values of their particular domain
- *Atoms* are
  - any tuples of values in a relation
    - *e.g.,* $< x_1, x_2, x_3, \ldots x_n > \in r$ is a atom if r is an n-ary relation and each x_i are either domain variables of domain values
  - any valid comparison between domain variables and/or values
- *P* is a predicate formula that is (1) an atom (2) valid logical connectives over a predicate formula (3) $\exists_x P(x)$ or $\forall_x P(x)$

Example Queries

- Find the *ID, name, dept_name, salary* for instructors whose salary is greater than $80,000
  - $\{< i, n, d, s > \mid \ < i, n, d, s > \in instructor \land s > 80000\}$
- As in the previous query, but output only the *ID* attribute value
  - $\{< i > \mid \exists_{n,d,s} < i, n, d, s > \in instructor \land s > 80000\}$
    - Note that ∃ is necessary to define the variables
  - For convenience, we will use "_" for any ∃ variable which is simply a placeholder in a relation. The the above would be:
    - $\{< i > \mid \exists_{s} < i, \_, \_, s > \in instructor \land s > 80000\}$
- Find the names of all instructors whose department is in the Watson building

$$\{< n > \mid \ \exists_d (< \_, n, d, \_ > \in instructor \\ \land \exists_b (< d, b, \_ > \in department \ \land \ b = \text{“Watson”}))\}$$

Example Queries

- Find the set of all courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or both

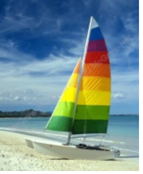
$$\{<c> \mid \exists_{s,y}(<c, \_, s, y, \_, \_, \_ > \in section \land s = \text{“Fall”} \land y = 2009) \lor \exists_{s,y}(<c, \_, s, y, \_, \_, \_ > \in section \land s = \text{“Spring”} \land y = 2010)\}$$

This case can also be written as

$$\{<c> \mid \exists_{s,y}(<c, \_, s, y, \_, \_, \_ > \in section \land ((s = \text{“Fall”} \land y = 2009) \lor (s = \text{“Spring”} \land y = 2010))\}$$

- Find the set of all courses taught in the Fall 2009 semester, and in the Spring 2010 semester

$$\{<c> \mid \exists_{s,y}((<c, \_, s, y, \_, \_, \_ > \in section \land s = \text{“Fall”} \land y = 2009) \land \exists_{s,y}((<c, \_, s, y, \_, \_, \_ > \in section] \land s = \text{“Spring”} \land y = 2010)\}$$



Safety of Expressions

- It is possible to write relational calculus expressions that generate infinite relations.
- For example, $\{ t \mid \neg\ t \in r \}$ results in an infinite relation if the domain of any attribute of relation *r* is infinite

Safety of Expressions

The expression:

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from *dom* (*P*)
   1. *i.e.*, the values appear either in *P* or in a tuple of a relation mentioned in *P*.
2. For every "there exists" sub-formula of the form $\exists_x P_1(x)$, the sub-formula is true if and only if there is a value of $x \in dom(P_1)$ such that $P_1(x)$ is true.
3. For every "for all" sub-formula of the form $\forall_x P_1(x)$, the sub-formula is true if and only if $P_1(x)$ is true for all values $x \in dom(P_1)$.

Universal Quantification

- Find all students who have taken all courses offered in the Biology department
  - $\{< i > \mid \exists_{\_} \; (< i, \_, \_, \_ > \in student \; \wedge$
    $(\forall_{\_,\, ci,\, dn} (< ci, \_, dn, \_ > \in course \wedge dn = \text{“Biology”}$
    $\Rightarrow \; \exists_{\_} \; (<i, ci, \_, \_, \_, \_> \in takes \;))\}$
  - Note that without the existential quantification on student, the above query would be unsafe if the Biology department has not offered any courses.

Relational Algebra

- A procedural query language with six core operators:
  - select: σ
  - project: $\prod$
  - union: $\cup$
  - set difference: –
  - Cartesian product: x
  - rename: ρ
  - These operators take one (σ, $\prod$, ρ) or two relations ($\cup$, –, x) as inputs and output a single relation.
  - Question: what happens if we don’t have all six operators?

Select Operation – Example

- Relation *r* :

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

- $\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

Select Operation

- Notation: $\sigma_p(r)$
- p is called the **selection predicate**
- Defined as:

$$\sigma_p(\boldsymbol{r}) = \{t \mid t \in r \textbf{ and } p(t)\}$$

Where p is a formula in propositional logic consisting of **terms** connected by : $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
Each **term** is one of:

<attribute> *op* <attribute> or <constant>

where *op* is one of: $=, \neq, >, \geq, <, \leq$

- Example of selection:

$$\sigma_{\textit{dept\_name}=\text{“Physics”}}(\textit{instructor})$$

- Question: which part of SQL corresponds to σ ?

Project Operation – Example

- Relation *r*:

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

- $\prod_{A,C}(r)$

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

Project Operation

- Notation:

$$\prod_{A_1, A_2, \ldots, A_k} (r)$$

  where A_1, A_2 are attribute names and r is a relation name.

- The result is defined as the relation of k columns obtained by erasing the columns that are not listed.
- Note: since relations are sets, duplicate rows "removed" from result.
- Example: To eliminate the *dept_name* attribute of *instructor*

$$\prod_{ID,\ name,\ salary} (instructor)$$

- Question: which part of SQL corresponds to $\prod$?

Union Operation – Example

- Relations *r, s:*

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- r ∪ s:

A	B
α	1
α	2
β	1
β	3

Union Operation

- Notation: $r \cup s$
- Defined as:

$$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

- For $r \cup s$ to be valid.
  1. *r, s* must have the *same* **arity** (same number of attributes)
  2. The attribute domains must be **compatible**

     (Example: 2nd column of *r* deals with the same type of values as does the 2nd column of *s*. Column names may be different.)
- Example: to find all courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or in both

$$\Pi_{course\_id}(\sigma_{semester=\text{“Fall”} \land year=2009}(section)) \cup$$
$$\Pi_{course\_id}(\sigma_{semester=\text{“Spring”} \land year=2010}(section))$$

Set difference of two relations

- Relations *r*, *s*:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- *r – s:*

A	B
α	1
β	1

Set Difference Operation

- Notation $r - s$
- Defined as:

$$r - s = \{t \mid t \in r \textbf{ and } t \notin s\}$$

- Set differences must be taken between **compatible** relations, just like unions.
- Example: to find all courses taught in the Fall 2009 semester, but not in the Spring 2010 semester

$$\Pi_{course\_id}(\sigma_{semester=\text{“Fall”} \ \wedge \ year=2009}(section)) \ -$$
$$\Pi_{course\_id}(\sigma_{semester=\text{“Spring”} \ \wedge \ year=2010}(section))$$

- Question: How do we do set difference in SQL?
  - (recall “except” and “intersection” are not always present.)

Cartesian-Product Operation – Example

- Relations *r, s*:

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

- *r* x *s*:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Cartesian-Product Operation

- Notation: $r \times s$
- Defined as:

$$r \times s = \{t\, q \mid t \in r \textbf{ and } q \in s\}$$

Note: *"t q"* denotes a tuple obtained by concatenating together *t* and *q*.

- Note: the definition assumes that attributes of $r(R)$ and $s(S)$ are disjoint. (That is, $R \cap S = \varnothing$.)

- If attributes of $r(R)$ and $s(S)$ are not disjoint, then renaming must be used.

Composition of Operations

- Can build expressions using multiple operations
- Example: $\sigma_{A=C}(r \times s)$

- $r \times s$

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

- $\sigma_{A=C}(r \times s)$

A	B	C	D	E
α	1	α	10	a
β	2	β	10	a
β	2	β	20	b

Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.
- Example:

$$\rho_X(E)$$

returns the expression E under the name X

- If a relational-algebra expression E has arity n, then

$$\rho_{x(A_1, A_2, \ldots, A_n)}(E)$$

returns the result of expression E under the name X, and with the attributes renamed to $A_1, A_2, \ldots, A_n$.

Example Query

- Find the largest salary in the university
  - Step 1: find instructor salaries that are less than some other instructor salary (*i.e.*, not maximum)
    - using a copy of *instructor* under a new name *d*
      - $\Pi_{instructor.salary}\,(\sigma_{\,instructor.salary\,<\,d.salary}\,(instructor \times \rho_d\,(instructor)))$
  - Step 2: find the largest salary
    - $\Pi_{salary}\,(instructor) - \Pi_{instructor.salary}\,(\sigma_{\,instructor.salary\,<\,d.salary}\,(instructor \times \rho_d\,(instructor)))$

Formal Definition of Relational Algebra

- A **basic expression** in the relational algebra consists of either a relation in the database (*e.g.*, *instructor*) or a constant relation (*e.g.*, {(1, Einstein), (2, Crick)}).
- A general **relational algebra expression** is either a basic expression or an expression constructed recursively using one of the following rules, where E_1 and E_2 denote existing relational-algebra expressions:
  - $E_1 \cup E_2$
  - $E_1 - E_2$
  - $E_1 \times E_2$
  - $\sigma_p (E_1)$, where P is a predicate on attributes in E_1
  - $\prod_S(E_1)$, where S is a list comprising a subset of the attributes in E_1
  - $\rho_{x(A1\ ,\ A2\ ,\ \ldots,\ An)}\ (E_1)$, where $x(A_1, A_2, \ldots, A_n)$ is the new name for E_1 and its attributes

Additional Operations

For convenience, additional relational operators can be defined.

- set intersection: $\cap$
- natural join: $\bowtie$
- theta join: $\bowtie_\theta$
- assignment: $\leftarrow$
- set division: $\div$
- outer join: ⟕, ⟖, ⟗
- …

Question: what happens if we are missing any of these operators?

Set-Intersection Operation

- Notation: $r \cap s$
- Defined as:

$$r \cap s = \{ t \mid t \in r \textbf{ and } t \in s \}$$

- Assume:
  - r, s have the *same arity*
  - attributes of r and s are compatible
- Core operator equivalence: $r \cap s = r - (r - s)$

Set-Intersection Operation – Example

- Relation *r, s*:

r

A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

- $r \cap s$

A	B
α	2

Assignment Operation

- The assignment operation ($\leftarrow$) provides a convenient way to express complex queries.
  - Write query as a sequential program consisting of
    - a series of assignments
    - followed by an expression whose value is displayed as a result of the query.
  - Assignment must always be made to a temporary relation variable.
- Example: find the largest salary in the university (in two lines of "code")

$$temp \leftarrow \Pi_{instructor.salary}(\sigma_{instructor.salary < d.salary}(instructor \times \rho_d(instructor)))$$

$$\Pi_{salary}(instructor) - temp$$

Core operator equivalence?

Natural-Join Operation

- Notation: $r \bowtie s$
- Let r and s be relations on schemas R and S respectively.
  Then, $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:
  - Consider each pair of tuples t_r from r and t_s from s.
  - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
    - t has the same value as t_r for attributes in R
    - t has the same value as t_s for attributes in S
- Example:

  $R = (A, B, C, D)$

  $S = (E, B, D)$

  - Result schema = (A, B, C, D, E)
  - Core operator equivalence:

$$r \bowtie s = \prod_{r.A,\ r.B,\ r.C,\ r.D,\ s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$

Natural Join Example

- Relations *r*, *s*:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

s

- $r \bowtie s$

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Natural Join and Theta Join

- Find the names of all instructors in the Comp. Sci. department together with the course titles of all the courses that the instructors teach
  - $\prod_{name,\ title} (\sigma_{dept\_name = \text{“Comp. Sci.”}} (instructor \bowtie teaches \bowtie course))$
- Natural join is associative
  - $(instructor \bowtie teaches) \bowtie course$ is equivalent to $instructor \bowtie (teaches \bowtie course)$
- The **theta join** operation $r \bowtie_\theta s$ is defined as
  - $r \bowtie_\theta s = \sigma_\theta (r \times s)$
  - Example: $instructor \bowtie_{instructor.ID = teaches.ID} teaches$

    (How is this different from *instructor* ⋈ *teaches* ?)

Natural Join and Theta Join (Cont.)

- Example schema:
  pet (<u>*p_id*</u>, *name*, *species*)
  owner (<u>*o_id*</u>, *name*, *address*)
  owns (<u>*p_id*</u>, *o_id*)
- Query: find the name and address of every dog that has an owner.
- Answer:

$$temp \leftarrow (pet \bowtie owns)$$
$$temp2 \leftarrow temp \bowtie_{temp.o\_id = owner.o\_id} owner$$
$$\Pi_{temp.name,\ address} (\sigma_{species = \text{“dog”}} (temp2))$$

Division Operator

- Given relations $r(R)$ and $s(S)$, such that $S \subset R$, $r \div s$ is the largest relation $t(R - S)$ such that
  $$t \times s \subseteq r$$
- Example:
  - let $r(ID, course\_id) = \prod_{ID,\ course\_id} (takes)$ and
    $$s(course\_id) = \prod_{course\_id} (\sigma_{\text{dept\_name="Biology"}}(course))$$
  - then $r \div s$ gives us students who have taken all courses in the Biology department
- Core operator equivalence:
  $$r \div s = \prod_{R-S} (r) - (\prod_{R-S} ((\prod_{R-S} (r) \times s) - r))$$

Division Operator (Cont.)

- Example with real data:
  - student IDs are 1, 2, 3, 4
  - course IDs are 'BIO-101', 'BIO-301', and ‘CS-101'
  - r = { (1, 'BIO-101'), (1, 'BIO-301'), (2, 'CS-301') }
  - s = { ('BIO-101'), ('BIO-301') }
  - $r \div s$ ={ (1) }

- The three-line query for $r \div s$ can be executed as follws:

  $temp1 = \prod_{R-S}(r)$ = { (1), (2) }

  $temp1 \times s$ = { (1, 'BIO-101'), (1, 'BIO-301'), (2, 'BIO-101'), (2, 'BIO-301') }

  $(temp1 \times s) - r$ = { (2, 'BIO-101'), (2, 'BIO-301') }

  $\prod_{R-S}((temp1 \times s) - r)$ = { (2) }

  $temp1 - temp2$ = { (1) }

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null.*
- Aggregate functions (not covered in this lecture) simply ignore null values (as in SQL).
- For duplicate elimination and grouping (not covered in this lecture), *null* is treated like any other value, and two *null* values are assumed to be the same (as in SQL).

Null Values

- Comparisons with null values return the special truth value: *unknown*
  - If *false* was used instead of *unknown*, then *not* $(A < 5)$ would not be equivalent to $A >= 5$
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: (*true* **and** *unknown*) = *unknown*,
    (*false* **and** *unknown*) = *false*,
    (*unknown* **and** *unknown*) = *unknown*
  - NOT*:* (**not** *unknown*) = *unknown*
  - In SQL the expression "*P* **is unknown**" evaluates to *true* if predicate *P* evaluates to *unknown*, and *false* otherwise.
- Result of select predicate is treated as *false* if it evaluates to *unknown*.

Good Database Design Requirements

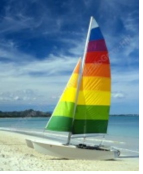
- Never specify information twice!
  - Let me repeat: Don’t specify things more than once!

- Questions:
  - Why is this a problem?
  - How do we know if something has been specified more than once?

- Note: this is not “database design” but “good database design”
  - Database design will be covered later in the course
  - The difference is akin to knowing whether or not a story is grammatically correct *vs*. whether or not it is a good story

Why is Repetition a Problem? Motivation

- Redundancy in a database leads to troublesome anomalies.
  - **Update anomalies**: a repeated value may be changed in one place but not in another place.
  - **Insertion anomalies:** in order to insert one value, it becomes necessary to insert some unrelated value.
  - **Deletion anomalies:** deleting one type of information leads to the loss of an another unrelated type of information.



Motivation (Cont.)

- Example: imagine *instructor* and *department* are merged into *inst_dept*:

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

- If a department changes its name, we may need to update many rows in *inst_dept*.
- If an instructor is inserted, we must include the instructor's department budget in the same row.
- If all ECE instructors are deleted, the ECE department no longer has any representation in the database.

Motivation (Cont.)

- We can remove redundancy by decomposing attributes and relations, but we pay a price:
  - additional processing needed to compute joins
  - additional space needed for tables and indexes
  - must enforce referential integrity constraints
- Usually, the price is right:
  - computation and memory/storage becoming less expensive
  - joins and referential integrity checks can be quite fast (*e.g.*, if tables are small or indexes are used)
  - on the other hand, dealing with anomalies may require human intervention, which is slow and costly

How do we know if something is repeated?

- Example 1: repetition within one tuple

  section(*course_id, sec_id, semester, year,*
  building, room_number)

  ('ECE356', '001', 'W13', 2013, 'E2', 'E2-1303')

- Example 2: repetition between tuples

  inst_dept(*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)

  ('22222', 'Einstein', '95000', 'Physics', 'Watson', 70000)
  ('33456', 'Gold', '87000', 'Physics', 'Watson', 70000)

Diagnosis and Remedy

- In the examples considered, there are two types of repetition:
  1. The value domains of some attributes are not **atomic**: one value may encode multiple pieces of information.

     Example: 'E2-1303' repeats information in 'E2'
  2. Attribute values in different tuples are related by **functional dependencies**: one subset of attributes **functionally determines** the values of another subset. (A type of constraint present in real data.)

     Example: *dept_name* → *building, budget*
- Remedies:
  1. Break up value domains to create atomic domains.
     1. Question: Is ECE356 atomic?
  2. Decompose relations to avoid specific types of FD's.

Relational Decomposition

- Consider the schema

  inst_dept(<u>*ID*</u>, *name*, *salary*, *dept_name*, *building*, *budget*)

  How do we know that (*dept_name*, *building*, *budget*) should be a separate relation?

  Which attributes must remain in *inst_dept* if we construct a new relation on (*dept_name*, *building*, *budget*)?

- Our goal is to produce a **lossless-join decomposition**: decomposition of relation schema R into schemas R_1 and R_2 such that for every instance $r(R)$, letting $r_1(R_1)$ and $r_2(R_2)$ denote the corresponding decomposed instances, $r = r_1 \bowtie r_2$ holds.
- Beware of **lossy decompositions**, in which relation r cannot always be reconstructed by joining r_1 and r_2.

Example: Lossy Decomposition

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
⋮				

Example: Lossless-Join Decomposition

- Decomposition of $R = (A, B, C)$ into
  $R_1 = (A, B)$ and $R_2 = (B, C)$:

A	B	C
α	1	A
β	2	B

$r(R)$

A	B
α	1
β	2

$r_1(R_1)$

B	C
1	A
2	B

$r_2(R_2)$

$r_1 \bowtie r_2$

A	B	C
α	1	A
β	2	B

- Note: $r = r_1 \bowtie r_2$ must hold for every ***possible*** relation instance r and corresponding instances r_1, r_2.

Roadmap

- We will define a theory of functional dependencies.
- We will use functional dependencies to decide whether a particular relation schema *R* is in “good” form.
- If *R* is not in “good” form, we will decompose it (very carefully) into a set of relation schemas $\{R_1, R_2, ..., R_n\}$ such that
  - each of the new relation schemas is in “good” form
  - the decomposition is a lossless-join decomposition
- We will study precise definitions of various levels of “goodness” called **normal forms**, which can be achieved by applying specific decomposition procedures.

First Normal Form (1NF)

- A value domain is **atomic** if its elements are considered to be indivisible units.
  - Examples of non-atomic domains:
    - multi-valued and composite attributes
    - identifiers such as 'ECE356', which can be broken up into parts
- A relational schema *R* is in **first normal form** if the domains of all attributes of *R* are atomic.
- Non-atomic domains are bad because:
  - they complicate storage
  - they encourage redundancy
  - they lead to information being encoded in business logic (*e.g.*, application parses 'ECE356' to obtain department name)
- **Assumption: from now on, all relations are in first normal form unless we say otherwise.**
- Question: Do we ***always*** want domains to be atomic? Why? Why not?

A Theory of Functional Dependencies

- **Functional dependencies** (FDs) are constraints on the set of **legal relations** – ones that conform to some conceptual model of the data, which itself is guided by our informal understanding of the world).
- FDs state that the value for a certain set of attributes determines (*i.e.*, constrains) uniquely the value for another set of attributes. An FD is a generalization of the notion of a **key**.

  Example: *dept_name* → *building, budget*

  Pronunciation:
  dept_name **functionally determines** building and budget

- **Note:** If we know the value of *dept_name* then we know that the values of *building* and *budget* are uniquely determined, but we may not know immediately what these values are.

Functional Dependencies (Cont.)

- Let R be a relation schema where

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relation $r(R)$, 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 r.

A	B
1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ may or may not hold.

Functional Dependencies (Cont.)

- 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
  - there is no $\alpha \subset K$ such that $\alpha \rightarrow R$.
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

  inst_dept (*<u>ID</u>, name, salary, dept_name, building, budget*)

  We expect the following functional dependencies to hold:

  $$dept\_name \rightarrow building$$

  $$ID \rightarrow building$$

  but we would not expect the following to hold:

  $$dept\_name \rightarrow salary$$

Use of Functional Dependencies

- We use functional dependencies 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.*
  - specify constraints on the set of legal relations
    - We say that *F* **holds on** *R* if all legal relations on *R* satisfy the set of functional dependencies *F.*
- 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. For example, a specific instance of *instructor* may, by chance, satisfy *name* → *ID*.

Functional Dependencies (Cont.)

- A functional dependency is **trivial** if it is satisfied by all instances of a relation
  - Example*:*
    - *ID, name* $\rightarrow$ *ID*
    - *name* $\rightarrow$ *name*
- In general, $\alpha \rightarrow \beta$ is trivial whenever $\beta \subseteq \alpha$.

Closure of a Set of Functional Dependencies

- Functional dependencies help us reason about data:
  - FDs represent constraints on legal relations
  - FDs help us identify certain forms redundancy
- Given a set F of functional dependencies, there may be certain other functional dependencies that are not in F but are logically implied by those in F.
  - For example: If F contains only $A \rightarrow B$ 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 $\boldsymbol{F^+}$.
- In general $F^+ \supseteq F$ holds.

Armstrong's Axioms

- We can find F^+, the closure of F, by repeatedly applying **Armstrong's Axioms:**
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (**reflexivity**)
  - if $\alpha \rightarrow \beta$, then $\gamma\,\alpha \rightarrow \gamma\,\beta$ (**augmentation**)
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (**transitivity**)

- These axioms (more correctly called **inference rules**) are
  - **sound**
    (*i.e.*, they generate only functional dependencies that actually hold)
  - and **complete**
    (*i.e.*, they generate all functional dependencies that hold)

Example: Applying Armstrong's Axioms

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H \}$
- Applying the axioms, we can obtain additional members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then by transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then by transitivity

Procedure for Computing F^+

- To compute the closure F^+ of a set of functional dependencies F:

input: F (a set of FDs)

$F^+ := F$

repeat

  for each functional dependency f in F^+

    apply reflexivity and augmentation rules 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

      then add the resulting functional dependency to F^+

until F^+ stops growing

output F^+

NOTE: Later on we will see an alternative procedure for this.

Additional Inference Rules

- The following rules can also be used to compute functional dependencies:
  - If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds (**union**)
  - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (**decomposition**)
  - If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds (**pseudotransitivity**)
- Note1: The above rules can be inferred from Armstrong's axioms.
- Note2: If you're asked on an exam to prove that some FD holds <u>using Armstrong's axioms</u> then use only Armstrong's axioms and <u>do not use these additional rules</u>.

Closure of Attribute Sets

- 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*

  input: α (set of attributes), *F* (set of FDs)
  result := α
  do
      for each $\beta \rightarrow \gamma$ **in** *F* **do**
        begin
          if $\beta \subseteq$ *result* **then** *result* := *result* $\cup\ \gamma$
        end
  until *result* stops growing
  output *result*

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- $(AG)^+$
  1. *result* = AG

     add B and C (since $A \rightarrow B$ and $A \rightarrow C$ and $A \subseteq AG$)
  2. *result* = $ABCG$

     add H (since $CG \rightarrow H$ and $CG \subseteq AGBC$)
  3. *result* = $ABCGH$

     add I (since $CG \rightarrow I$ and $CG \subseteq AGBCH$)
  4. *result* = $ABCGHI$

     done since *result* = R, and so *result* cannot grow any further

Uses of Attribute Set Closure

- Let R denote a relation schema, and let α denote a set of attributes.
- Use case 1: testing for a superkey:
  - To test whether α is a superkey of R, compute α^+ and check that α^+ contains all attributes of R.
- Use case 2: Testing individual functional dependencies:
  - To test whether a functional dependency $\alpha \rightarrow \beta$ holds on R, compute α^+ and check that $\beta \subseteq \alpha^+$.
  - Note: this can be much easier than computing F^+ itself!
- Use case 3: Computing the closure F^+ of a set of functional dependencies:

  For each $\gamma \subseteq R$:

  1. find the closure γ^+
  2. for each $S \subseteq \gamma^+$, output the FD $\gamma \rightarrow S$.

Example of Using Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- $(AG)^+ = ABCGHI$
- Is AG a candidate key?
  1. Is AG a super key?
     - Does $AG \rightarrow R$ hold?
       In other words, does $(AG)^+ \supseteq R$ hold?
  2. Is AG minimal (*i.e.*, no subset of AG is a superkey)?
     - Does $A \rightarrow R$ hold?
       In other words, does $(A)^+ \supseteq R$ hold?
     - Does $G \rightarrow R$ hold?
       In other words, does $(G)^+ \supseteq R$ hold?

Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others.
  - Example: $A \rightarrow C$ is redundant in $\{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - *E.g.*: on RHS: $\{A \rightarrow B,\quad B \rightarrow C,\quad A \rightarrow CD\}$ can be simplified to
      $$\{A \rightarrow B,\quad B \rightarrow C,\quad A \rightarrow D\}$$
    - *E.g.*: on LHS: $\{A \rightarrow B,\quad B \rightarrow C,\quad AC \rightarrow D\}$ can be simplified to
      $$\{\mathrm{A} \rightarrow B,\quad B \rightarrow C,\quad A \rightarrow D\}$$

- Informally speaking, a **canonical cover** of *F* is a "minimal" set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies.
  - *i.e.*, it is a transitive reduction of the FDs.

Extraneous Attributes

- Consider a set F of functional dependencies and a functional dependency $\alpha \rightarrow \beta$ in F.
  - Attribute A is **extraneous** in α if $A \in \alpha$
    and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - Attribute B is **extraneous** in β if $B \in \beta$
    and the set of functional dependencies
    $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - B)\}$ logically implies F.

- **Note 1:** The implication in the opposite direction holds trivially in each of the above cases because a potentially stronger functional dependency implies a potentially weaker one.
- **Note 2:** "X logically implies Y" means that any FD in Y can be obtained from the FDs in X using Armstrong's Axioms.
  In practice we can prove this point using attribute set closures.

Testing if an Attribute is Extraneous

- Consider a set F of functional dependencies and a functional dependency $\alpha \rightarrow \beta$ in F. We need two separate tests for extraneousness depending on whether the attribute being tested is part of α (*i.e.*, on the left) or β (*i.e.*, on the right).

- To test whether attribute $A \in \alpha$ is extraneous in α
  1. compute $(\{\alpha\} - A)^+$ using the dependencies in F
  2. A is extraneous in α if and only if $(\{\alpha\} - A)^+$ contains β
- To test whether attribute $B \in \beta$ is extraneous in β
  1. compute α^+ using only the dependencies in
     $$G = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - B)\}$$
  2. B is extraneous in β if and only if α^+ contains B
     (note: closure of α is computed here with respect to G)
- Note: Detailed examples follow in a few slides.

Extraneous Attributes: Example

- $R = (A, B, C, D)$
- Example: given $F = \{A \rightarrow C, AB \rightarrow C\}$
  - B is extraneous in $AB \rightarrow C$ because F logically implies $(F - \{AB \rightarrow C\}) \cup \{A \rightarrow C\} = \{A \rightarrow C\}$.
- Example: given $F = \{A \rightarrow C, AB \rightarrow CD\}$
  - C is extraneous in $AB \rightarrow CD$ because $(F - \{AB \rightarrow CD\}) \cup \{AB \rightarrow D\} = \{A \rightarrow C, AB \rightarrow D\}$ logically implies F.

Extraneous Attributes: Detailed Example

- $R = (A, B, C, D)$
- Example with additional detail:
  - given $F = \{A \rightarrow C, AB \rightarrow C\}$ check if B is extraneous in $AB \rightarrow C$
  - compute $F' = \{A \rightarrow C\}$ and check if $F \Leftrightarrow F'$
    - $F' \Rightarrow F$ follows trivially
    - to check if $F \Rightarrow F'$ compute attribute closures using F for the left side of each dependency in F'
      - *Case 1: $A \rightarrow C$*
        $A^+ = AC$ hence F implies $A \rightarrow C$
      - conclusion: $F \Rightarrow F'$ holds
    - thus we have shown $F \Leftrightarrow F'$ and so B is extraneous on the left side of $AB \rightarrow C$ in F

Extraneous Attributes: Detailed Example

- $R = (A, B, C, D)$
- Example with additional detail:
  - given $F = \{A \rightarrow C, AB \rightarrow CD\}$ check if C is extraneous in $AB \rightarrow CD$
  - compute $F' = \{A \rightarrow C, AB \rightarrow D\}$ and check if $F \Leftrightarrow F'$
    - $F \Rightarrow F'$ follows trivially
    - to check if $F' \Rightarrow F$ compute attribute closures using F' for the left side of each dependency in F
      - *Case 1:* $A \rightarrow C$
        $A^+ = AC$ hence F' implies $A \rightarrow C$
      - *Case 2:* $AB \rightarrow CD$
        $(AB)^+ = ABCD$ hence F' implies $AB \rightarrow CD$
      - conclusion: $F' \Rightarrow F$ holds
    - thus we have shown $F \Leftrightarrow F'$ and so C is extraneous on the right side of $AB \rightarrow CD$ in F

Canonical Cover

- Formally, a **canonical cover** for a set F of functional dependencies is a set F_c of functional dependencies such that:

    1. F logically implies all dependencies in F_c, and
    2. F_c logically implies all dependencies in F, and
    3. No functional dependency in F_c contains an extraneous attribute, and
    4. Each left side of a functional dependency in F_c is unique.

       Example: F_c cannot contain both $A \rightarrow B$ and $A \rightarrow C$ because in that case it should contain $A \rightarrow BC$ instead (union rule).

Computing a Canonical Cover

- To compute a canonical cover for F:

  input: F (set of FDs)
  $F_c := F$
  repeat
  Use the union rule to replace any pair of dependencies
  $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ in F_c with $\alpha_1 \rightarrow \beta_1 \beta_2$
  Look for a functional dependency $\alpha \rightarrow \beta$ in F_c with an
  extraneous attribute either in α or in β
  If found an extraneous attribute in α, delete it from α in $\alpha \rightarrow \beta$
  Else if found an extraneous attribute in β, delete it from β in $\alpha \rightarrow \beta$
  (do not delete attributes from both α and β in the same iteration!)
  until F_c does not change
  output F_c

- **Note:** The union rule may become applicable upon deleting an extraneous attribute, in which case it has to be re-applied at the next iteration.

Computing a Canonical Cover: Example

- $R = (A, B, C)$
  $F = \{A \rightarrow BC$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - F_c becomes $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- Is A is extraneous in $AB \rightarrow C$?
  - Check whether $\{A \rightarrow BC, B \rightarrow C\}$ logically implies $AB \rightarrow C$.
    - Yes, because $AB \rightarrow B$ holds by reflexivity and $B \rightarrow C$ is given, hence $AB \rightarrow C$ holds by transitivity.
  - F_c becomes $\{A \rightarrow BC, B \rightarrow C\}$
- Is C is extraneous in $A \rightarrow BC$?
  - Check whether $A \rightarrow BC$ is implied by $\{A \rightarrow B, B \rightarrow C\}$
    - Yes, because $A \rightarrow B$ and $B \rightarrow C$ are given and so $A \rightarrow C$ holds by transitivity, hence applying the union rule on $A \rightarrow B$ and $A \rightarrow C$ we obtain $A \rightarrow BC$.
  - F_c becomes $\{A \rightarrow B, B \rightarrow C\}$, which has no extraneous attributes.
- Result $\{A \rightarrow B, B \rightarrow C\}$ is a canonical cover for F.

Second Normal Form (2NF)

- A **non-prime attribute** of a relation schema *R* is an attribute that is not a part of any candidate key of *R*.
- A relation schema *R* is in **second normal form (2NF)** with respect to a set *F* of functional dependencies if:
  - *R* is in 1NF; and
  - no non-prime attribute is functionally determined under *F* by any proper subset (of attributes) of any candidate key of *R*
    - *i.e.,* the relation depends on the **whole** of the key
- **Note:** If every candidate key in *R* has only one attribute then *R* is automatically in 2NF.

Second Normal Form: Example

- Example 1: (many departments per instructor)

  instructor (*ID*, *name*, *salary*)
  department(*dept_name*, *building*, *budget*)
  inst_dept(*ID*, *dept_name*)

  $$F = \{ \quad ID \rightarrow name, salary \quad dept\_name \rightarrow building, budget \quad \}$$

- Schemas *instructor* and *department* are in 2NF w.r.t. *F* because they have only one candidate key each, and their candidate keys have only one attribute.
- Schema *inst_dept* is also in 2NF w.r.t. *F* because it has no non-prime attributes.

Second Normal Form: Example

- Example 2: (many departments per instructor)

  instructor (*ID*, *name*, *salary*)
  inst_dept(*ID*, *dept_name*, *building*, *budget*)

  $F = \{ \quad ID \rightarrow name, salary$
  $\quad dept\_name \rightarrow building, budget \quad \}$

- Schema *instructor* remains in 2NF w.r.t. *F*.
- Schema *inst_dept* is not in 2NF w.r.t. *F* because *dept_name* → *building* and yet *dept_name* is a proper subset of the candidate key (*ID*, *dept_name*).
  - Note that we are declaring that the department name determines the building and the budget; what if the those were also dependent on the particular instructors?

Second Normal Form: Example

- Example 3: (one department per instructor)

  inst_dept(<u>*ID*</u>, *name*, *salary*, *dept_name, building, budget*)

  $F = \{ \quad ID \rightarrow name, salary, dept\_name$
  $\quad dept\_name \rightarrow building, budget \quad \}$

- Schema *inst_dept* is in 2NF w.r.t. *F* because it has only one candidate key, and this candidate key has only one attribute.

- **Note:** the schema is in 2NF and yet it permits redundancy!

Third Normal Form (3NF)

- A relation schema R is in **third normal form (3NF)** with respect to a set F of functional dependencies if for every dependency $\alpha \rightarrow \beta$ in F^+ such that $\alpha, \beta \subseteq R$, at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (*i.e.*, $\beta \subseteq \alpha$);
  - α is a superkey for R; or
  - each attribute A in $\beta - \alpha$ is contained in a candidate key for R (**Note:** each attribute may be in a different candidate key)
  - *i.e.,* the relation depends on the **nothing but** the key
- **Theorem:** For any relation schema R and set of functional dependencies F, if R is in 3NF w.r.t. F then R is in 2NF w.r.t. F.

Testing for 3NF

- Algorithm: given a relation schema R and set F of functional dependencies
  - compute attribute closures for all subsets of attributes of R w.r.t. F
  - identify all candidate keys, examine the attribute closures and look for a dependency $\alpha \rightarrow \beta$ that violates 3NF
  - if no such dependency exists then conclude that R is in 3NF w.r.t. F
- Example: $R(A, B, C, D)$ $F = \{ A \rightarrow B, BC \rightarrow AD \}$
  - $(A)^+ = AB$ $(B)^+ = B$ $(C)^+ = C$ $(D)^+ = D$
    $(AB)^+ = AB$ $(AC)^+ = ABCD$ $(AD)^+ = ABD$
    $(BC)^+ = ABCD$ $(BD)^+ = BD$ $(CD)^+ = CD$
    $(ABC)^+ = ABCD$ $(ABD)^+ = ABD$ $(ACD)^+ = ABCD$ $(BCD)^+ = ABCD$
  - the candidate keys are AC and BC, and non-trivial dependencies occur in $(A)^+$, $(AC)^+$, $(AD)^+$, $(BC)^+$, $(ABC)^+$, $(ACD)^+$ and $(BCD)^+$
  - for each non-trivial dependency $\alpha \rightarrow \beta$ identified, either α is a superkey (*e.g.*, $AC \rightarrow BD$) or else each attribute of β is part of some candidate key (*e.g.*, $AD \rightarrow B$ where B is part of BC), hence 3NF is not violated
  - conclusion: R is in 3NF

Testing for 3NF - Simplified

- A simpler 3NF test exists in the special case when F is a set of functional dependencies over the attributes of R (and no other attributes)
- Algorithm: given a relation schema R and set F of functional dependencies
  - identify all candidate keys (*e.g.*, using attribute closures)
  - examine each functional dependency $\alpha \rightarrow \beta$ in F and check whether that dependency violates 3NF
  - if no dependency in F violates 3NF then conclude that R is in 3NF w.r.t. F
- Example: $R(A, B, C, D)$ $\quad F = \{ A \rightarrow B, BC \rightarrow AD \}$
  - C must be part of every candidate key because it appears in FDs only on the left side, but C itself is not a superkey
  - AC and BC are the only two candidate keys
  - no FD in F violates the rules for 3NF
  - conclusion: R is in 3NF

Third Normal Form: Example

- Example 4: (one department per instructor)

  inst_dept(<u>*ID*</u>, *name*, *salary*, *dept_name, building, budget*)

  $F = \{ \quad ID \rightarrow name, salary, dept\_name$
  $\quad\quad dept\_name \rightarrow building, budget \quad \}$

- Schema *inst_dept* is not in 3NF w.r.t. *F* because *dept_name* → *building* and yet *dept_name* is not a superkey for *inst_dept* and *building* is not contained in {*ID*} – the candidate key for *inst_dept.*

Third Normal Form: Example

- Example 5: (one department per instructor)

  instructor(*ID*, *name*, *salary, dept_name*)
  department(*dept_name*, *building*, *budget*)

  $F = \{ \quad ID \rightarrow name, salary, dept\_name$
  $\qquad dept\_name \rightarrow building, budget \quad \}$

- Both schemas are in 3NF w.r.t. *F* because the left side of any functional dependency (*i.e.*, the α in any $\alpha \rightarrow \beta$ in F^+) is a superkey for the relation to which the FD pertains.

Boyce-Codd Normal Form (BCNF)

- A relation schema *R* is in **Boyce-Codd Normal Form (BCNF)** with respect to a set *F* of functional dependencies if for every dependency $\alpha \rightarrow \beta$ in F^+ such that $\alpha, \beta \subseteq R$, at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (*i.e.*, $\beta \subseteq \alpha$)
  - α is a superkey for *R*

- **Theorem:** For any relation schema *R* and set of functional dependencies *F*, if *R* is in BCNF w.r.t. *F* then *R* is in 3NF w.r.t. *F*.

- **Note:** BCNF is also known as "3.5NF" – it is only slightly stronger than 3NF.

Testing for BCNF

- Algorithm: given a relation schema R and set F of functional dependencies
  - compute attribute closures for all subsets of attributes of R w.r.t. F
  - examine the attribute closures and look for a dependency $\alpha \rightarrow \beta$ that violates BCNF
  - if no such dependency exists then conclude that R is in BCNF w.r.t. F
- Example: $R(A, B, C, D)$ $F = \{ A \rightarrow B, BC \rightarrow AD \}$
  - $(A)^+ = AB$ $(B)^+ = B$ $(C)^+ = C$ $(D)^+ = D$
    $(AB)^+ = AB$ $(AC)^+ = ABCD$ $(AD)^+ = ABD$
    $(BC)^+ = ABCD$ $(BD)^+ = BD$ $(CD)^+ = CD$
    $(ABC)^+ = ABCD$ $(ABD)^+ = ABD$ $(ACD)^+ = ABCD$ $(BCD)^+ = ABCD$
  - the closure $(A)^+ = AB$ shows that $A \rightarrow B$, which is non-trivial
  - since A is not a superkey for R it follows that $A \rightarrow B$ violates BCNF
  - conclusion: R is not in BCNF

Testing for BCNF - Simplified

- A simpler BCNF test exists in the special case when F is a set of functional dependencies over the attributes of R (and no other attributes)
- Algorithm: given a relation schema R and set F of functional dependencies
  - examine each functional dependency $\alpha \rightarrow \beta$ in F and check whether that dependency violates BCNF
  - if no dependency in F violates BCNF then conclude that R is in BCNF w.r.t. F
- Example: $R(A, B, C, D)$ $F = \{ A \rightarrow B, BC \rightarrow AD \}$
  - $A \rightarrow B$ is non-trivial and violates BCNF since A is not a superkey
  - conclusion: R is not in BCNF

Boyce-Codd Normal Form: Example

- Schemas that satisfy 3NF usually also satisfy BCNF (*e.g.*, the schemas in Example 5), but not always.
- Example 6: booking tennis courts

  booking(*court_ID*, *start_time*, *end_time*, *rate_type*)

court_ID	start_time	end _time	rate_type
1	9:30	10:30	SAVER
1	12:00	13:00	STANDARD
2	10:00	11:00	PREMIUM-A
2	15:00	16:00	PREMIUM-B
2	16:00	17:00	PREMIUM-B

- It is possible to define $F \supseteq \{rate\_type \rightarrow court\_ID\}$ so that *booking* is in 3NF w.r.t. F but not in BCNF w.r.t. F.

How good is BCNF?

- There exist schemas in BCNF that do not seem to be sufficiently normalized.
- Example: a school maintains contact info for parents

$$parent\_info\ (parent\_ID,\ \underline{child\_name},\ \underline{parent\_phone})$$

$$F = \{\ child\_name,\ parent\_phone \rightarrow parent\_ID\ \}$$

(a parent may have multiple children and phone numbers)

parent_ID	*child_name*	*parent_phone*
99999	David	512-555-1234
99999	David	512-555-4321
99999	William	512-555-1234
99999	William	512-555-4321

How good is BCNF? (Cont.)

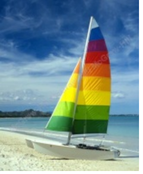
- To avoid repetition, it seems logical to decompose *parent_info* into:

parent_child

parent_ID	*child_name*
99999	David
99999	William

parent_phone

parent_ID	*parent_phone*
99999	512-555-1234
99999	512-555-4321



How good is BCNF? (Cont.)

- Since {*child_name*, *parent_phone*} is a superkey, it can be shown easily that *parent_info* is in BCNF w.r.t. *F*.
- Nevertheless, *parent_info* may suffer from insertion anomalies. For example, if we add a phone number 981-992-3443 to 99999, then we need to add two tuples:

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

- This type of redundancy is addressed in higher normal forms (*e.g.*, 4NF) and in the theory of **multivalued dependencies**.

Summary

- 1NF prohibits non-atomic domains.
  (Does not refer to functional dependencies at all.)
- 2NF prohibits functional dependencies of non-prime attributes on parts of candidate keys.
- 3NF prohibits transitive functional dependencies of non-prime attributes on candidate keys.
- BCNF prohibits all functional dependencies that might lead to redundancy.
- 4NF and higher normal forms deal with redundancy that cannot be captured at all using functional dependencies.
- For simple schemas, BCNF and 3NF are usually good enough.

Relational Decomposition

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $\quad B \rightarrow C\}$
  Candidate key = A
- R is not in BCNF or 3NF (because of $B \rightarrow C$)
- Decomposition:
  - $R_1 = (A, B)$ $\quad R_2 = (B, C)$
    - Are R_1 and R_2 in BCNF? Are they in 3NF?
    - Is this a lossless-join decomposition?
    - What happened to the functional dependencies, and how would we enforce them in an RDBMS?

The Big Picture

- **Normalization procedures** remove redundancy by decomposing a relation schema into multiple smaller schemas.

- In **lossless-join decomposition** a relation schema R is decomposed into R_1 and R_2 such that for every legal relation r of schema R the following holds:

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

- Our goal is to devise procedures that attain specific normal forms by way of lossless-join decompositions.

Dependency Preservation

- What else should a lossless-join decomposition preserve?
- Functional dependencies are both an asset and a liability:
  - They allow us to constrain the set of legal relations. In principle, such constraints can be checked and enforced by the database to prevent anomalies.
  - Checking that a relation satisfies a set *F* of functional dependencies carries a cost. We want to minimize this cost by avoiding joins and Cartesian products.
- **Goal for decomposition procedures:** ensure that all functional dependencies can be enforced in the decomposed schema by checking them against only one table at a time.

Dependency Preservation (Cont.)

- Let R be a relation schema and let F be a set of functional dependencies for R.
- Consider a lossless-join decomposition of R into $R_1, R_2, \ldots, R_n$.
- Let F_i denote the set of dependencies in F^+ that include only attributes in R_i.
- The decomposition is **dependency preserving** if and only if

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

- If the above does not hold, then checking updates for violation of functional dependencies may require computing potentially expensive joins.

Decomposing a Schema into BCNF

- Suppose that we have a relation schema *R* and a set *F* of functional dependencies such that some non-trivial dependency $\alpha \rightarrow \beta$ in *F* causes a violation of BCNF w.r.t. *F*.
- To avoid this violation, we can decompose *R* into two relations:
  - $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$
- Applying to example from last slide:
  - R = (*ID*, *name*, *salary*, *dept_name, building, budget*)
  - α = *dept_name*
  - β = *building, budget*
  - $(\alpha \cup \beta)$ = (*dept_name, building, budget*)
  - $(R - (\beta - \alpha))$ = (*ID, name, salary, dept_name*)

BCNF Decomposition Procedure

input: R (relation schema), F (set of functional dependencies)

result := $\{R\}$ /* set of relation schemas */
done := **false**
while (not *done***) do**
  if (there is a schema R_i in *result* that is not in BCNF w.r.t. F)
    then let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that
        holds on R_i such that $\alpha \cap \beta = \varnothing$ and
        α is not a superkey for R_i
      remove relation R_i from *result*
      add a relation on the attributes $R_i - \beta$ to *result*
      add a relation on the attributes $\alpha\beta$ to *result*
    else
      done := **true**
output: *result*

Note: each R_i returned is in BCNF w.r.t. F, and decomposition is lossless-join.

Easy Example of BCNF Decomposition

- $R\ (A, B, C)$
  $F = \{A \rightarrow B$
  $\quad B \rightarrow C\}$
  Candidate key = A
- R is not in BCNF ($B \rightarrow C$ but B is not a superkey)
- Decomposition:
  - $R_1\ (B, C)$
  - $R_2\ (A, B)$
- It is straightforward to verify that both R_1 and R_2 are in BCNF with respect to F, and that in this particular case the decomposition is dependency-preserving.

Harder Example of BCNF Decomposition

- $R(A, B, C, D, E)$
  $F = \{A \rightarrow B$
  $BC \rightarrow D\}$
- Applying the simplified BCNF test on R we see that $A \rightarrow B$ violates BCNF because A is not a superkey.
- Therefore we decompose R into $R_1(A, B)$ and $R_2(A, C, D, E)$
- R_1 is automatically in BCNF because it only has two attributes.
- To analyze R_2 we use the general BCNF test:
  $(A)^+ = AB$ $(C)^+ = C$ $(D)^+ = D$ $(E)^+ = E$
  $\boldsymbol{(AC)^+ = ABCD}$
  …
  We can stop computing attribute closures for R_2 since we see that $AC \rightarrow D$ holds and violates BCNF. <u>(The simplified test does not reveal this!)</u>
- Next, we decompose R_2 and look for additional BCNF violations.

Harder Example of BCNF Decomposition

- *… continued*
  $R_2 = (A, C, D, E)$
  $F = \{ A \rightarrow B$
  $\quad BC \rightarrow D \}$
- Since we found that $AC \rightarrow D$ holds, $R_2 = (A, C, D, E)$ is not in BCNF with respect to F and must be decomposed.
- We decompose R_2 into $R_{2.1}(A, C, D)$ and $R_{2.2}(A, C, E)$
- To analyze $R_{2.1}$ we use the general BCNF test:
  $(A)^+ = AB \quad (C)^+ = C \quad (D)^+ = D$
  $(AC)^+ = ABCD \quad (AD)^+ = ABD \quad (CD)^+ = CD$
  Since there are only trivial dependencies and $AC \rightarrow D$, $R_{2.1}$ is in BCNF.
- To analyze $R_{2.2}$ we use the general BCNF test:
  $(A)^+ = AB \quad (C)^+ = C \quad (E)^+ = E$
  $(AC)^+ = ABCD \quad (AE)^+ = ABE \quad (CE)^+ = CE$
  Since there are only trivial dependencies, is in BCNF.
- Output: $\{ R_1(A, B), R_{2.1}(A, C, D), R_{2.2}(A, C, E) \}$
- <u>Not dependency preserving</u> because of $BC \rightarrow D$!

3NF Decomposition Procedure

input: R (relation schema), F (set of functional dependencies)

F_c := canonical cover for F
$i := 0$
for each functional dependency $\alpha \rightarrow \beta$ in F_c **do**
    if none of the schemas R_j, $1 \leq j \leq i$ contains $\alpha\ \beta$
        then begin
            $i := i + 1$
            $R_i := \alpha\ \beta$
        end
if none of the schemas R_j, $1 \leq j \leq i$ contains a candidate key for R
    then begin
        $i := i + 1$
        R_i := any candidate key for R
    end
/* Optionally, remove redundant relations */
repeat
if any schema R_j is contained in another schema R_k
    then /* delete R_j ***/**
        $R_j := R_i$
        $i := i - 1$
return $\{R_1, R_2, ..., R_i\}$

Caveats in 3NF Decomposition

- At the beginning, be sure to compute F_c and not F^+, which is more time-consuming.
- Later on check whether R_j contains a candidate key for R. To do that, let $\alpha = R_j$, then compute α^+ using F_c, and test whether $\alpha^+ = R$.
- If none of the R_j contains a candidate key for R, then we must find a candidate key α for R and create a new relation over α. To find α:
  - first find any superkey γ for R, *e.g.*, by looking at F_c (if in doubt, start with $\gamma = R$)
  - then try to remove attributes from γ (one by one) to make it a minimal superkey
  - let α be your final γ

3NF Decomposition Example

- $R(A, B, C, D)$
  $F = \{ AB \rightarrow CD, B \rightarrow C, AC \rightarrow B, B \rightarrow D \}$
- Find the candidate keys: AB and AC.
- Apply simplified 3NF test since F only refers to attributes in R:
  - R is not in 3NF w.r.t. F because $B \rightarrow D$ holds where B is not a superkey and D is not part of any candidate key.
- Compute a canonical cover:
  - after removing extraneous attributes and applying the union rule we obtain $F_c = \{ B \rightarrow CD, AC \rightarrow B \}$
- The 3NF decomposition first creates $R_1(B, C, D)$ and $R_2(A, B, C)$.
- There are no redundant relations and R_2 contains a candidate key.
- Output $\{ R_1, R_2 \}$.
- <u>Dependency preserving</u> because every FD in F_c can be checked against one of R_1 and R_2. 3NF decomposition is always dependency-preserving!

3NF Decomposition Example 2

- $R(A, B, C, D)$
  $F = \{ A \rightarrow B, B \rightarrow C \}$
- Find the candidate keys: First, note that A and D must appear in every candidate key, since there is no functional dependency in F where A or D appear on the right. Next, note that $(AD)^+ = ABCD$. This implies that AD is the one and only candidate key.
- Apply simplified 3NF test since F only refers to attributes in R:
  - R is not in 3NF w.r.t. F because $A \rightarrow B$ holds where A is not a superkey and B is not part of any candidate key.
- It is easy to show that F itself is a canonical cover, so let $F_c = F$.
- The 3NF decomposition first creates $R_1(A, B)$ and $R_2(B, C)$
- Next, add $R_3(A, D)$ since neither R_1 nor R_2 contains a candidate key.
- None of the relations are redundant, so do not remove anything.
- Output $\{ R_1, R_2, R_3 \}$.
- <u>Dependency preserving</u> because every FD in F_c can be checked against one of R_1 and R_2. (R_3 not needed for FD checking.)

Finding a Candidate Key – Long Way

- $R(A, B, C, D)$
  F = { $A \rightarrow B$, $B \rightarrow C$ }
- First, take R and try subsets of 3 out of 4 attributes:
  $(ABC)^+ = ABC$ $(ABD)^+ = ABCD$ $(ACD)^+ = ABCD$ $(BCD)^+ = BCD$
- Thus, $ABCD$ is not a candidate key, but ABD and ACD are superkeys.
- Next, take ABD and try subsets of 2 out of 3 attributes:
  $(AB)^+ = ABC$ $(AD)^+ = ABCD$ $(BD)^+ = BCD$
- Thus, ABD is not a candidate key, but AD is a superkey.
- Next, take AD and try subsets of 1 out of 2 attributes:
  $A^+ = ABC$ $D^+ = D$
- Since neither A nor D is a superkey, AD is a candidate key.

- Note: as explained in the last slide, AD is the only candidate key for this example. In general, there can be many candidate keys.

Relationship to SQL/RDBMS

- There are no formal FDs in an RDBMS.
  1. Decompose relations into BCNF so the table structure enforces dependencies.
  2. Enforce functional dependencies using superkeys, assertions, and triggers.

  - (Assertions are powerful but expensive to test and not supported by mainstream databases, and so we do not cover them in the course.

    Syntax: **create assertion** assertion-name **check** predicate)
- Sometimes the primary key and uniqueness constraints alone logically imply all the functional dependencies. If not, you can use SQL triggers in addition.