

Update Anomalies and Lossless Join

SWEN 304
Trimester 2, 2017

Lecturer: Dr Hui Ma
Engineering and Computer Science



Slides by: Pavle Morgan & Hui Ma

Database Design Quality

- Logical database design aims at a layout of relational tables such that:
 - most common queries can be processed efficiently
 - data redundancies and processing difficulties with database are minimised

- We will now focus on the second objective:
 - find semantic properties of well-designed databases:
absence of *data redundancies*, *update anomalies*,
data inconsistencies
 - develop automatic tools to achieve these properties

Database Design Quality

- Database **normalisation**: obtain database schema avoiding redundancies and processing difficulties
- Database **denormalisation**: join normalized relation schemata for the sake of better query processing

Database Design Quality

- Update anomalies
- lossless join decomposition
- Functional dependencies
- Normal forms: define to which extent we should normalize
- synthesis algorithm (3NF decomposition) and BCNP decomposition algorithm: the formal normalization methods that show how to normalize
- *Readings from the textbook:*
 - *Chapter 15*
 - *Chapter 16*

Outline

- Universal Relation Schema
- Data redundancy via efficient query processing
- Data redundancy and redundancy-causing dependencies
- Processing difficulties: consistency validation, update anomalies
- Lossless join decomposition

Universal Relation Schema

- In the theory of the relational data model, there exists an assumption about the existence of a **universal relation schema** (URS), denoted (U, C)
- Universal relation schema contains all attributes and all constraints of the UoD
- A **URS** is a possible database schema of a UoD database
- There are many consequences:
 - universal relation as an instance over URS ,
 - **unique role** of attributes,
 - after decomposing a URS , each relation schema has a different set of attributes, so the relation schema names can be replaced by attribute sets,
 - **sound theory**,...

URS Example: Employee

- Table Employee(e_no, e_name, salary, child) with following instance

e_no	e_name	salary	child
003	Homer	2000	Bart
003	Homer	2000	Lisa
007	Marge	3000	Bart
007	Marge	3000	Lisa

Trade-off: Data Redundancy but Efficient Query Processing

- Redundancy-causing data dependency:
different rows with same entry in e_no-column always have same entries in e_name-column and in salary-column, respectively
- If employee has two children, then e_name and salary need to be stored repeatedly for that employee
- Query: List the e_name of all employees who have a child named Bart:

$$\pi_{e_name}(\sigma_{child='Bart'}(Employee))$$

- Query can be processed efficiently (no join needed)

Trade-off: No Data Redundancy but no Efficient Query Processing

- Relations Info(e_no, e_name, salary) and Parent(e_no, child)
- Due to data dependency Employee is the lossless join of Info and Parent

Info

e_no	e_name	Salary
003	Homer	2000
007	Marge	3000

Parent

e_no	Child
003	Bart
003	Lisa
007	Bart
007	Lisa

- Data redundancies eliminated
- Query: List the e_name of all employees who have a child named Bart:

$$\pi_{e_name}(\sigma_{child='Bart'}(Info * Parent))$$

- Query processing requires join

Redundancy-causing Dependencies

- Employee(e_no, e_name, salary, child) with:

e_no	e_name	salary	child
003	Homer	2000	Bart
003	Homer	2000	Lisa

- e_no functionally determines e_name (we write $e_no \rightarrow e_name$):
 - For the same entry in e_no-column there is always the same entry in e_name-column
- e_no does not functionally determine child:
 - First and second tuple have same e_no-entry, but different child-entries

Redundancy-causing Dependencies

- What does data redundancy mean:
 - We obtain duplicates for some projections, such as {e_no, e_name} or {e_no, salary}

e_no	e_name
003	Homer
003	Homer

e_no	salary
003	2000
003	2000

- We can infer data entries from other data entries and data dependencies:

e_no	e_name	salary	child
003	Homer	2000	Bart
003	?	?	Lisa

Processing Difficulties: Consistency Validation

- Consistency of key constraints is simple to check (database practice):
 - Just check uniqueness of entries in key columns
- Info(e_no, e_name, salary) and Parent(e_no,child) only exhibit keys

Info

e_no	e_name	salary
003	Homer	2000
007	Marge	3000

Parent

e_no	child
003	Bart
003	Lisa
007	Bart
007	Lisa

Processing Difficulties: Update Anomalies

- Data consistency prohibitively expensive to check for *non-key constraints*, e.g. any two rows with same e_no-entry must have same e_name-entry
- Relation Employee(e_no, e_name, salary, child) with following instance

e_no	e_name	salary	child
003	Homer	2000	Bart
003	Homer	2000	Lisa
007	Marge	3000	Bart
007	Marge	3000	Lisa

- Updates for redundant data must be processed for all its occurrences
- Data redundancy may cause anomalies when updating required

Processing Difficulties: Update Anomalies

- insertion of (003, Homer, 2500, Maggie) into Person-table: satisfies key {e_no, child}, but violates $e_no \rightarrow salary$

e_no	e_name	salary	child
003	Homer	2000	Bart
003	Homer	2000	Lisa
003	Homer	2500	Maggie
007	Marge	3000	Bart
007	Marge	3000	Lisa

- update of (003, Homer, 2000, Lisa) to (003, Homer, 2500, Lisa) in Person-table: satisfies key {e_no, child}, but violates $e_no \rightarrow salary$

e_no	e_name	salary	child
003	Homer	2000	Bart
003	Homer	2500	Lisa
007	Marge	3000	Bart
007	Marge	3000	Lisa

Universal Relation: Faculty

<i>StudId</i>	<i>StName</i>	<i>NoPts</i>	<i>CourId</i>	<i>CoName</i>	<i>Grd</i>	<i>LecId</i>	<i>LeName</i>
007	James	80	M214	Math	A+	333	Peter
131	Susan	18	C102	Java	B-	101	Ewan
007	James	80	C102	Java	A	101	Ewan
555	Susan	18	M114	Math	B+	999	Vladimir
007	James	80	C103	Algorithm	A+	99	Peter
131	Susan	18	M214	Math	ω	333	Peter
555	Susan	18	C201	C++	ω	222	Robert
007	James	80	C201	C++	A+	222	Robert
010	John	0	C101	INET	ω	820	Ray

Update Anomalies

- The URS Faculty satisfies (and suppose has) two keys:
 - *StudId* + *CourId*, and
 - *StudId* + *LecId*
- Recall **entity integrity constraint**: a constraint requiring that no component of any relation schema key may have a null value
- Update **anomalies** are:
 - Insertion anomaly,
 - Deletion anomaly, and
 - Modification anomaly

Insertion Anomaly

- A **new student** cannot be inserted before he/she enrolls a course that is already lectured by someone
- A **new course** cannot be introduced before it is associated with a lecturer and enrolled by some students
- A **new lecturer** cannot be hired before he / she is assigned a course and at least one student enrolled the course taught by the new lecturer

Deletion Anomaly

- If there is a **student** that is the **only one** associated either with a **course**, or a **lecturer** (or both), and this student withdraws, **deleting** his / her tuple will cause the loss of course, or lecturer information
- e.g.
 - $((StudId, 10), (StName, John))$
- Similarly, if there is a **lecturer** that..., and similarly if there is a **course**...

Modification Anomaly

- Modification anomaly is a direct consequence of data **redundancy** in the universal relation, and refers to the fact that modification of an attribute value have to be performed on **many** tuples, instead of on just one
- e.g.
 - Suppose James passes another exam, then besides a new tuple, the values of the *NoPts* attribute of all tuples belonging to James have to be modified, as well

A Question

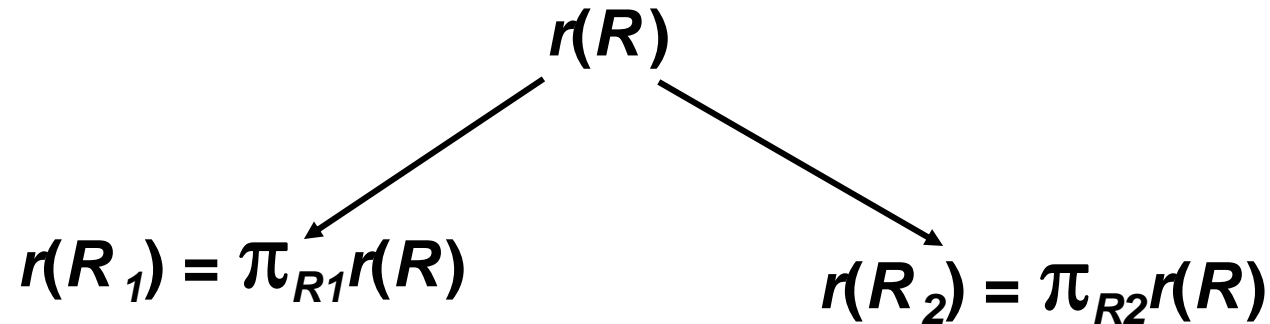
- How to prevent update anomalies?
 - a) To place a ban on database updates
 - b) To leave the database in an inconsistent state
 - c) To brake *URS* into smaller pieces that will exhibit less redundancy-causing dependencies

How to Avoid Update Anomalies

- To avoid update anomalies we need to avoid data redundancies (redundancy-causing dependencies)
- We need **split** the universal relation onto a number of smaller ones which do not contain data redundancies
- Splitting a relation into a set of relations is called **decomposition**
- A further natural expectation would be that we should be able to **recover** the universal relation (or its arbitrary part) by using this decomposition without any loss of information
- We give a formal definition of lossless join decomposition later

Decomposing a Universal Relation

- **Projection** is the only relational algebra operation that can be used to decompose a **universal relation**



- Requirement $R_1 \cup R_2 = R$ (**attribute conservation**)
- e.g.

Department (*LecId*, *LeName*, *CourId*, *CoName*, *DptId*,
DptName)

We split onto:

Lecturer (*LecId*, *LeName*, *CourId*)

Course (*CourId*, *CoName*, *DptId*, *DptName*)

Reconstructing the Universal Relation

- **Natural join** is the only relational algebra operation that can be used to recover **universal relation** or one of its parts from projections

$$\begin{array}{ccc} r(R_1) = \pi_{R_1} r(R) & & r(R_2) = \pi_{R_2} r(R) \\ & \searrow \quad \swarrow & \\ & \pi_{R_1} r(R) * \pi_{R_2} r(R) & \end{array}$$

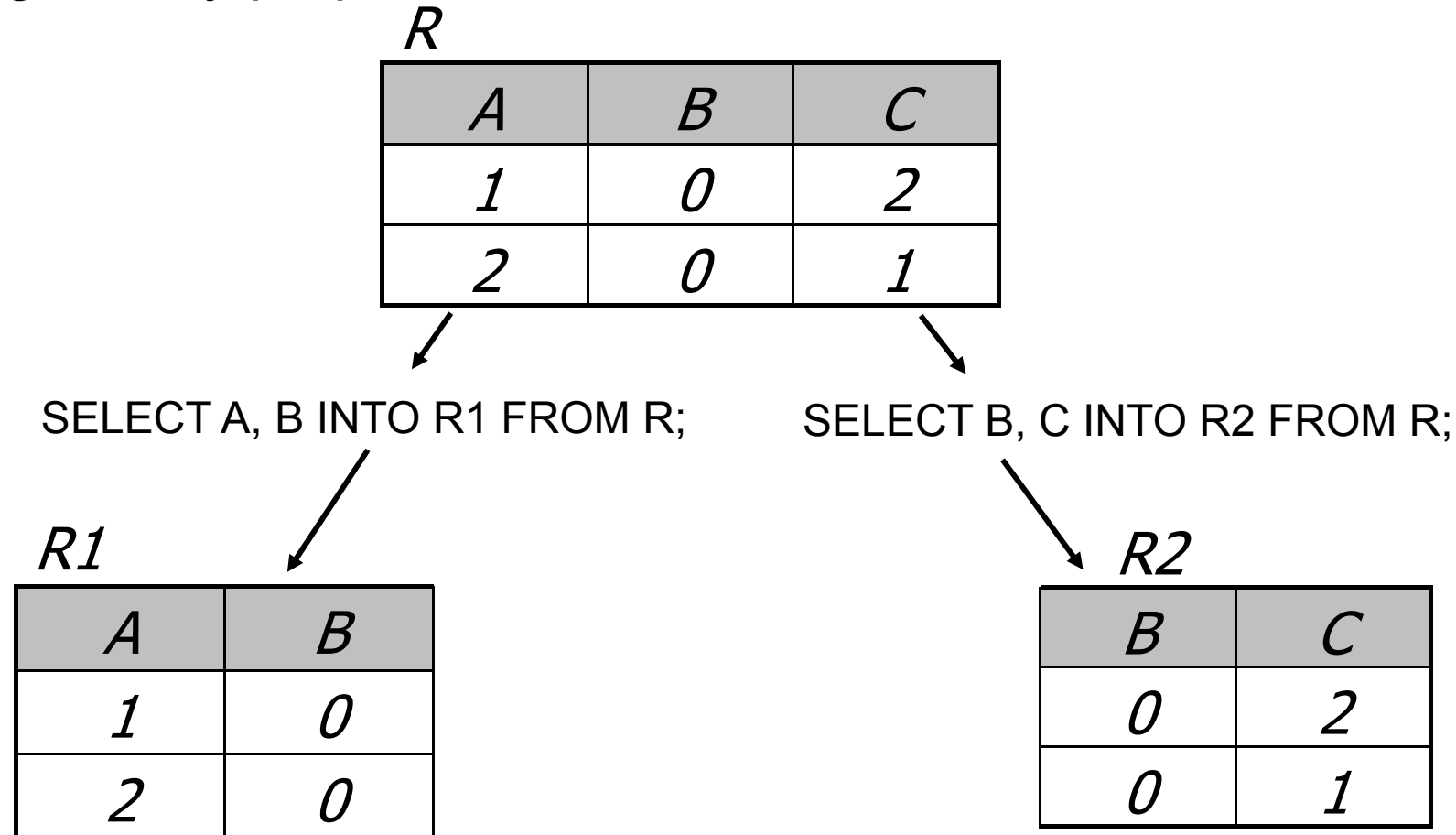
- This reconstruction places an **additional** requirement towards decomposition

$$R_1 \cap R_2 \neq \emptyset,$$

otherwise the join would turn into cross product

Additive (lossy) Join

The condition $R_1 \cap R_2 \neq \emptyset$ is even **not** sufficient to guaranty proper reconstruction



A Question

A Query: SELECT * FROM R1 NATURAL JOIN R2;

R1

<i>A</i>	<i>B</i>
<i>1</i>	<i>0</i>
<i>2</i>	<i>0</i>

R2

<i>B</i>	<i>C</i>
<i>0</i>	<i>2</i>
<i>0</i>	<i>1</i>

- Which answer will be produced?

a)

R

<i>A</i>	<i>B</i>	<i>C</i>
<i>1</i>	<i>0</i>	<i>2</i>
<i>2</i>	<i>0</i>	<i>1</i>

b)

R'

<i>A</i>	<i>B</i>	<i>C</i>
<i>1</i>	<i>0</i>	<i>2</i>
<i>1</i>	<i>0</i>	<i>1</i>
<i>2</i>	<i>0</i>	<i>2</i>
<i>2</i>	<i>0</i>	<i>1</i>

- How to explain b) being produced by SQL?
 - Lossy (or additive) join?

Lossless Join Decomposition

- A decomposition $D = \{R_1, R_2, \dots, R_m\}$ of a relation R has the **lossless (nonadditive) join** property wrt. the set of dependencies F on R if, for every relation $r(R)$ that satisfies F ,

$$\pi_{R_1}r(R) * \dots * \pi_{R_m}r(R) = r(R)$$

where $*$ is the natural join of all the relations in D

- It is proved in the theory of the relational data model that the decomposition of a relation schema R onto R_1 and R_2 is *lossless (nonadditive)* if the intersection $R_1 \cap R_2$ contains a **key** of R_1 or a key of R_2

Lossless Join Decomposition Examples

- Decomposition of

Department (*LecId*, *LeName*, *CourId*, *CoName*, *DptId*, *DptName*)

- onto

Lecturer (*LecId*, *LeName*, *CourId*),

Course (*CourId*, *CoName*, *DptId*, *DptName*)

- is a lossless join decomposition, because

$\{LecId, LeName, CourId\} \cap \{CourId, CoName, DptId, DptName\}$
 $= CourId$

Summary

- Update anomalies emerge when one relation contains data redundancies
- A solution of the problem is sought through decomposition
- Lossless (nonadditive) join ensures that the original relation can be recovered from its projections, and is guaranteed by the presence of a relation schema key in the intersection of the decomposition