Theory for evaluating & Improving relational schema

ISYS2120 Data and Information Management

Prof Alan Fekete

University of Sydney

Acknowledge: slides from Uwe Roehm and Alan Fekete, and from the materials associated with reference books (c) McGraw-Hill, Pearson

This week's material

- Lots of content
- Focus on the crucial concepts and skills
  - Functional dependency
  - Candidate key, superkey
  - BCNF
  - Decomposition (lossless-jopin, dependency preserving)
- For each concept: why it is important, develop intuition, understand precise definition using maths/logic, able to do calculations/checks using the procedure we teach

Today's Agenda

- Motivation
- Making it precise: Functional Dependency
- Making it precise: Normal Form
- Making it precise: Schema Decomposition

Schema Design Process

- The relational schema is best obtained by starting with a conceptual design (eg. an E-R model)
  - This can be converted to relational schema
- However, the relational schema may arise in other ways
  - eg. start from data in a spreadsheet
    - Typically gives one wide table!
  - eg. choose tables by raw intuition
- However we get it, we should evaluate the schema, and improve it if necessary
  - This lecture teaches a theory-based way to do this

Motivating Example

- Example: *Assume a direct data import from an Excel worksheet*

Mining Data Collection						
mine	**state**	**commodity**	**abbrv**	**capacity**	**company**	**homepage**
Olympic Dam	SA	Uranium	U	100	BHP Billiton	www.bhpbilliton.com
Blair Athol	QLD	Coal	Cbl	920	Rio Tinto	www.riotinto.com
Hunter Valley	NSW	Coal	Cbl	1430	Rio Tinto	www.riotinto.com/index.asp
Hunter Valley	NSW	Coal	Cbl	1430	Coal & Allied	www.coalandallied.com.au
Blair Athol	QLD	Uranium	U	76	Rio Tinto	www.riotinto.com

Inconsistent Information

Redundant Information

- There are "better" and "worse" relational schemas; How can we judge the quality of relational schemas?

Evaluation of a DB Design

- The most important requirement is **adequacy**:
  that the design should allow representing all the important facts about the design
  - Make sure every important process can be done using the data in the database, by joining tables as needed
  - eg. “can we find out which driver made a particular delivery?”

- If a design is adequate, then we seek to **avoid redundancy** in the data
  - and at a side effect, being able to insert/update/delete information without the need for (extensive) use of null values, or risk to introduce inconsistency

  (Redundant data is where the same information is repeated in several places in the db)

Evils of Redundancy

- *Redundancy* is at the root of several problems associated with relational schemas:
  - wasteful storage
  - **Insertion Anomaly:**
    Adding new rows forces user to create duplicate data or to use *null* values.
  - **Deletion Anomaly:**
    Deleting rows may cause a loss of data that would be needed for other future rows!
  - **Update Anomaly:**
    Changing data in a row forces changes to other rows because of duplication (or if not done, inconsistency can be the outcome)

- Note: It is the anomalies with modifications that are the serious concern, not the extra space used in storage

Anomalies Example

Mining Data Collection						
mine	**state**	**commodity**	**abbrv**	**capacity**	**company**	**homepage**
Olympic Dam	SA	Uranium	U	100	BHP Billiton	www.bhpbilliton.com
Blair Athol	QLD	Coal	Cbl	920	Rio Tinto	www.riotinto.com
Hunter Valley	NSW	Coal	Cbl	1430	Rio Tinto	www.riotinto.com
Hunter Valley	NSW	Coal	Cbl	1430	Coal & Allied	www.coalandallied.com.au
Blair Athol	QLD	Uranium	U	76	Rio Tinto	www.riotinto.com

Start from a version with inconsistency corrected

Question – Is this a relation?

Answer – Yes: unique rows and no multivalued attributes

Question – What seems to be the primary key?

Answer – Composite: (Mine, Commodity, Company)

Question – What happens with data modifications?

Anomalies in Previous Example

- Insertion Anomaly:
  - If another company buys a stake into an existing mine, we have to re-enter the ‘mine/state’ information, also “commodity/abbrv/capacity”, causing duplication.
  - What if we want to insert a mine which has no owner so far? We either cannot do it at all (PK!) or we get many *NULL* values.
- Deletion Anomaly:
  - If we delete all Uranium mines, we loose the information that ‘U’ is the chemical identifier for the commodity ‘Uranium’!
  - Or if composite PK, we cannot delete the last company for a mine!
- Update Anomaly:
  - For changing, e.g., the *homepage* of a company, we have to update multiple tuples, ., or risk introducing inconsistency

Why do these anomalies exist here?

Because there are multiple themes (entity types) placed into one relation. This results in duplication and unnecessary dependencies

Normal Forms

- There is a theory that allows one to see whether or not a particular schema risks having redundancy anomalies
- This theory looks at the design and also at constraints on the application, expressed in a mathematical form as *functional dependencies*
- If the design and its dependencies have certain properties, we say that the design is in a particular *normal form*
  - ***There are several that are used; we focus on BCNF***
- Our goal is to use schema which are in BCNF

Decomposition

- Decomposes changes a schema by replacing one relation by other relations
  - Between them, the decomposed relations have all the attributes of the original
  - The data in each decomposed table is a projection of the data in the original table

Decomposition

mine	state
Olympic Dam	SA
Blair Athol	QLD
Hunter Valley	NSW

Mine information

mine	commodity	capacity
Olympic Dam	Uranium	100
Blair Athol	Coal	920
Hunter Valley	Coal	1430
Blair Athol	Uranium	76

Mine-commodity connection

commodity	abbrv
Uranium	U
Coal	Cbl

Commodity information

mine	company
Olympic Dam	BHP Billiton
Blair Athol	Rio Tinto
Hunter Valley	Rio Tinto
Hunter Valley	Coal & Allied

Ownership connection

company	homepage
BHP Billiton	www.bhpbilliton.com
Rio Tinto	www.riotinto.com
Coal & Allied	www.coalandallied.com.au

Company information

Evaluating a Decomposition

- A proposal to use decomposition on a schema should be evaluated
- Does it allow all the original information to be captured properly? [This is vital for the decomposition to be appropriate]
- Does it allow all the original application domain constraints to be captured properly? [This is desirable]
- These issues are formalized as properties of a decomposition
  – It needs to be *lossless-join*
  – It should be *dependency-preserving*

Overall design process

- Consider a proposed schema
- Find out application domain properties expressed as functional dependencies
- See whether the schema has a certain property (every relation is in BCNF)
- If not, pick a relation in the schema which is not in BCNF and can be decomposed in a particular way (the decomposition is lossless-join and dependency-preserving)
- Replace the original relation by its decomposed tables
- Repeat the evaluation

This lecture teaches the theory of this, and particular steps to perform

Today's Agenda

- Motivation
- Making it precise: Functional Dependency
- Making it precise: Normal Form
- Making it precise: Schema Decomposition

Functional Dependency

- A mathematical way to describe a variety of constraints on the valid data
- These can be used to identify whether a schema has problems such as redundancy, and then to suggest improvements.
- **Functional Dependency** (often abbreviated as FD):
  - Intuitively: "If two tuples of a relation R agree on values in X, then they must also agree on the Y values."
  - Note that X and Y are each a set of columns
    - Include as a special case where X or Y is a single column
- We write $X \rightarrow Y$
  - "X (functionally) determines Y"
  - "Y is functionally dependent on X"

Functional Dependency

- Recall that X and Y are each a *set of columns*
  - Include as a special case where X or Y is a single column
- In mathematics notation: Given a relation R with schema ***R***, and two sets of attributes $X = \{X_1, \ldots, X_m\} \subseteq \boldsymbol{R}$ and $Y = \{Y_1, \ldots, Y_n\} \subseteq \boldsymbol{R}$.
  A **functional dependency (FD)** $X \rightarrow Y$ holds over relation ***R*** if, for every allowable instance *R* of ***R***:

$$\forall \; r, s \in R : \quad r.X = s.X \;\Rightarrow\; r.Y = s.Y$$

FD Example

Recall: FDs like other constraints,
must be given by domain experts

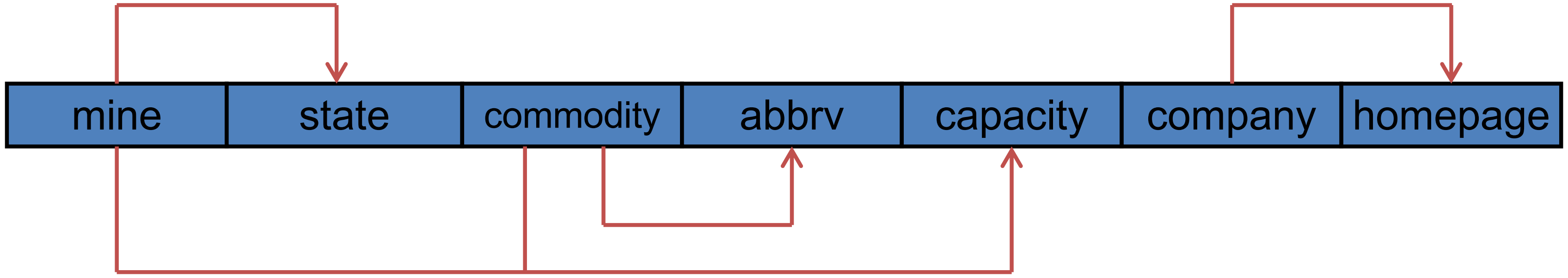
- FDs for motivating example:
  - *mine → state*
  - *commodity → abbrv*
  - *mine commodity → capacity*
  - *company → homepage*

Expresses: there is only one state which is associated to a given mine

Expresses: there is only one capacity which is associated to a given combination of mine and commodity

Note: in general, a mine can produce several commodities.
So we do not say mine → commodity

Graphical notation is sometimes used:



We can draw a line with arrowheads
going to the dependent column(s)
from the column(s) that are depended on

FD Example

- Check the FDs in motivating example:
  - *mine → state*
  - *commodity → abbrv*
  - *mine commodity → capacity*
  - *company → homepage*

Mining Data Collection

mine	state	commodity	abbrv	capacity	company	homepage
Olympic Dam	SA	Uranium	U	100	BHP Billiton	www.bhpbilliton.com
Blair Athol	QLD	Coal	Cbl	920	Rio Tinto	www.riotinto.com
Hunter Valley	NSW	Coal	Cbl	1430	Rio Tinto	www.riotinto.com
Hunter Valley	NSW	Coal	Cbl	1430	Coal & Allied	www.coalandallied.com.au
Blair Athol	QLD	Uranium	U	76	Rio Tinto	www.riotinto.com

Some Remarks

- A FD is an assertion about the schema of a relation not about a particular instance.
  - It must be fulfilled by all allowable instance relations.
- If we look at an instance, we cannot tell for sure that a FD holds
  - The most insight we can gain by looking at a “typical” instance are “hints”…
  - We can however check if the instance violates some FD
- FDs must be identified by experts, based on the semantics of an application.

Extending the terminology

- We can also speak of FDs that apply to the schema of a database containing several relations
- These are simply giving each relation with its own FDs

Candidate key

- A set of columns X makes up a *candidate key* for R provided both the following hold
  - For every attribute A of R, we have X $\rightarrow$ A
    - That is, we cannot have two rows which are the same in the set of columns, but different somewhere
  - No subset of X has this property
- So, given a set F of FDs, we can check if a set of columns makes a candidate key by seeing whether they functionally determine all the other columns
  - But, X $\rightarrow$ A might be always true because of what we are told, even though it is not be *explicitly* listed in the set of FDs we are told
  - **So, we will need a way to reason with the FDs we are told**

Related terms

- Choose one *candidate key* as the ***primary key***
  - Used as identifier to capture relationships, and stored in other tables as foreign key
- A ***superkey*** is a column or set of columns that includes a *candidate key*
  - A *candidate key*, plus perhaps extra columns
  - X is a superkey for R means: for every attribute A of R, we have $X \rightarrow A$; this is also written as $X \rightarrow R$

Keys Example

Suppose we are told that R(A,B,C,D) has FDs:

AB → D ,

AB → C

Then: AB determines all other attributes (namely, C and D).

No subset of AB determines all other attributes (eg data instance shows A does not determine B, B does not determine C, etc

So AB is candidate key for R

AB can be assigned as primary key (PK).

ABC is a superkey, as is ABD

Table			
<u>A</u>	<u>B</u>	C	D
1	3	X	V1
2	4	Y	V2
2	5	Z	V1
3	3	Y	V2
…	…	…	..

Deducing more FDs

- Given some FDs, we can usually infer additional FDs:
  - Example: *cid* $\rightarrow$ *points*, *points* $\rightarrow$ *lot* implies *cid* $\rightarrow$ *lot*
    - If two tuples agree on *cid*, then they agree on points (because *cid* $\rightarrow$ *points*), and because of that, they agree on *lot* (because of *points* $\rightarrow$ *lot*)

- A FD *f* is *implied by* a set of FDs *F* if *f* holds whenever all FDs in *F* hold.

- ***F^+ : closure of F*** is the set of all FDs that are implied by *F*

Rules for deducing more FDs

- **Armstrong's Axioms** for deducing more FDs (X, Y, Z are sets of attributes):
  1. ***Reflexivity rule***: If $X \subseteq Y$, then $Y \rightarrow X$

     These are called "trivial fds"; every trivial fd is always true in any schema
  2. ***Augmentation rule***: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
  3. ***Transitivity rule***: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

More on reasoning about FDs

- Armstrong's Axioms are
  - Sound: anything they generate, starting from a set F of fds, is a valid fd in F^+
  - Complete: repeated application of these rules will generate all FDs in the closure F^+
- A couple of additional rules (that follow from Armstrong's Axioms.):

4. ***Union rule***: If $X \longrightarrow Y$ and $X \longrightarrow Z$, then $X \longrightarrow YZ$
5. ***Decomposition rule***: If $X \longrightarrow YZ$, then $X \longrightarrow Y$ and $X \longrightarrow Z$
6. ***Pseudotransitivity rule:*** If $X \longrightarrow Y$ and $SY \longrightarrow Z$, then $XS \longrightarrow Z$

Example of inferring more FDs

- Suppose we have $R = (A, B, C, G, H, I)$ with the following set of fds
  $FD = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- some members of FD^+
  - $A \rightarrow H$
    - 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 transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - from $CG \rightarrow H$ *and* $CG \rightarrow I$ *:* this an example of the "union rule"; it follows by
      - Augmentation of $CG \rightarrow I$ to infer $CG \rightarrow CGI$, augmentation of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity

More example of reasoning about FDs

- Example: Orders (*date,cid,pid,descr,price,weight,amount*) and FDs {*date cid pid* $\rightarrow$ *amount*, *pid* $\rightarrow$ *descr price weight* }.
- It follows:

date, cid, pid $\rightarrow$ *pid* (reflexivity rule)

descr, price, weight $\rightarrow$ *descr* (reflexivity rule)

pid $\rightarrow$ *descr* (decomposition rule using *pid* $\rightarrow$ *descr price weight*)

date, cid, pid $\rightarrow$ *descr* (transitivity rule using *date, cid, pid* $\rightarrow$ *pid* and *pid* $\rightarrow$ *descr*)

FD Attribute Closure

- Computing the closure of a set of fds can be expensive (Size of closure is exponential in #attr!) so we typically don't do it

- However, we often want to check if a particular fd $X \rightarrow Y$ is logically implied by the given set of FDs F.
  - That is, check whether $X \rightarrow Y$ will hold in every valid state of the relation (which must satisfy all the fds in F)
- An efficient check is:
  - Compute the *attribute closure* of X (denoted X^+) based on F:
    - X^+ is the set of all attributes A such that $X \rightarrow A$ is in F^+
    - There is a simple algorithm to compute this (time cost is linear in #attr and #fds)
  - Check if Y is in X^+

Computing the Closure of Attributes

This calculation is often called "The Chase"
(though that term is also used for other algorithms too)

You need to be able to do this calculation

- Starting with a given set of attributes, one repeatedly expands the set by adding the right sides of FD's as soon as their left side is added:

1. Initialise result with the given set of attributes: $X=\{A_1, \ldots, A_n\}$
2. Repeatedly search for some FD $A_1 A_2 \ldots A_m \rightarrow C$
   such that all $A_1, \ldots, A_m$ are already in the set of attributes *result*, but C is not.

   Note that an fd will only be used once to add attributes to result

   When such an fd is found, add C to the set *result.*
3. Repeat step 2 until no more attributes can be added to *result*
4. The final set *result* is the correct value of X^+

From FDs to Keys

You need to be able to do these calculations

- The set of Functional Dependencies can be used to check if a set A of attributes is a superkey of R:
  - Calculate A^+
  - If A^+ contains all columns of R, then A is a superkey
- To check if A is a candidate key
  - Look at every subset B of A that is produced by leaving one attribute out of A
    - Calculate B^+
    - If $B^+ = R$, then A is not a minimal superkey, and so A is not a candidate key
  - If B^+ is not R, for every subset B we consider, then A is a candidate key for R

Example of checking candidate key

- *R = (A, B, C, G, H, I)*
- *F* = {$A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H$}
- Check if *AG* a candidate key?
- Calculate $(AG)^+$
  1. *result = AG*
  2. *result = ACG* ($A \rightarrow C$, and *A* subset of *result,* so include *C* in *result*)
  3. *result = ABCG* ($A \rightarrow B$ and *A* subset of *result,* so include *B* in *result)*
  4. *result = ABCGH* ($CG \rightarrow H$ and *CG* subset of *result,* so include *H* in *result)*
  5. *result = ABCGHI* ($CG \rightarrow I$ and *CG* subset of *result,* so include *I* in *result)*

Is AG a super key? YES! (ie Does $AG \rightarrow R$?)

Is any smaller subset of AG a superkey? NO!

1. Leave out A, so check is B a superkey? NO
2. Leave out B, so check is A a superkey? NO

Today's Agenda

- Motivation
- Making it precise: Functional Dependency
- Making it precise: Normal Form
  - Especially BCNF
- Making it precise: Schema Decomposition

Schema Normalization

- FDs can be used to identify schemas with problems and to suggest refinements.
- Main Idea we use: Aim for a schema where every FD has form of a key constraint.
  - Each non-key field is anyway functionally dependent on every candidate key
  - BCNF says there are no other FDs

- **Schema Normalization:** The process of validating and improving a logical design so that it satisfies certain constraints (*Normal Forms*) that avoid unnecessary duplication of data
  - Idea: decompose relations with anomalies to produce smaller, *well-structured* relations

- Note: Starting with ER diagram conceptual model, and using the approach of designing logical relational schema based on this, then we often get very close to a fully normalised schema.
  - But to be sure we have to check…

Normal Forms

- Before doing schema refinement, the first question to ask is whether any refinement is needed!
- If a relation is in a *normal form* (such as **BCNF**), it is known that certain kinds of problems are avoided/minimised. This can be used to help us decide whether decomposing the relation will help.
- Role of FDs in detecting redundancy:
  - Consider a relation R with 3 attributes, ABC.
    - No FDs hold:
      There is no redundancy here.
    - Given A $\longrightarrow$ B:
      Several tuples could have the same A value, and if so, they'll all have the same B value! This might indicate reduncancy and potentional anomalies

BCNF

- A relation with schema ***R*** and fds *F* is in BCNF (Boyce-Codd Normal From) exactly when, for every fd $X \rightarrow Y$ in F^+, either
  - $Y \subseteq X$ (i.e., the FD is trivial) or
  - *X* is a superkey of **R** (that is, $X \rightarrow R$)
- Definition says: Check each fd that applies to the relation (including those deduced from the ones you were given)
  - If the rhs is contained in the lhs; ok (but uninteresting)
  - If the lhs is a superkey for the whole relation; ok
- If every fd has one of these cases, then the relation is BCNF
- A schema is in BCNF if every relation in it is BCNF

Be careful with the use of mathematics and logic!
Definitions are precise

Check for BCNF

You need to be able to do this calculation

- The definition wants to check every fd that holds, including those that can be deduced from *F*
- In fact, you only need to check the ones you are given (if the given fds are all ok, so are any that are deduced from these)
- A relation with schema ***R*** and fds *F* is in BCNF (Boyce-Codd Normal From) exactly when, for all $X \rightarrow Y$ in *F*, either
  - $Y \subseteq X$ (i.e., the FD is trivial) or
  - *X* is a superkey of **R** (that is, $X \rightarrow R$)
- When every fd in *F* has one of these cases, then the relation is BCNF
- A schema is in BCNF if every relation in it is BCNF

BCNF Examples

- **Example**:
  Person1(*SSN, Name, Address*)
  - Suppose that the only FD is SSN $\rightarrow$ Name, Address
  - Chase for SSN+ terminates with SSN, Name, Adress, which has all atteributes i=of Person1
  - So SSN is a superkey, and therefore Person1 is in BCNF
- **Example:**
  Order(date,cid,pid,amount) with F={date cid pid $\rightarrow$ amount}
  - This is in BCNF as chase for (date cid pid)+, terminates after one step with all attributes of Order
- **Example:**
  - Product (pid,descr,price,weight) with F = {pid $\rightarrow$ descr price weight}
    - This is in BCNF.

(non) BCNF Examples

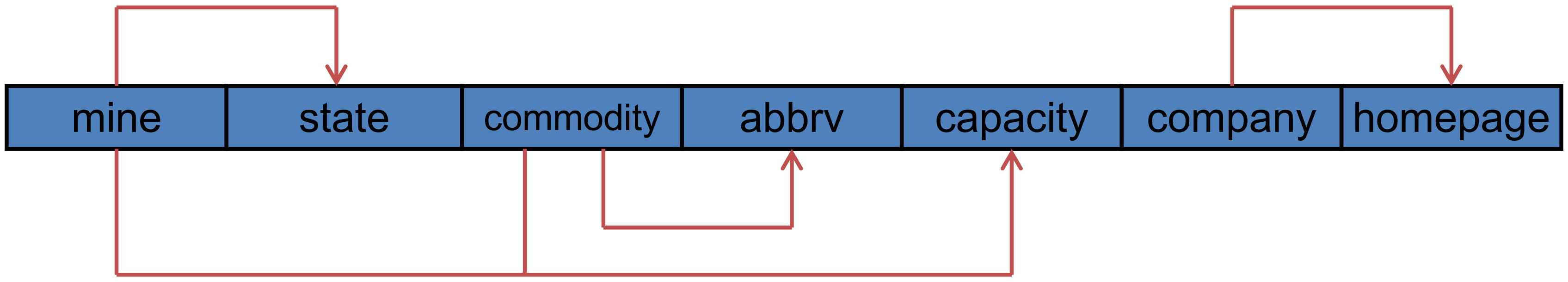
- *Person (SSN, Name, Address, Hobby)*
  - If the FD is *SSN* $\rightarrow$ *Name, Address* then the relation does not satisfy requirements of BCNF
    - *SSN* is not a superkey; it does not determine Hobby
      - The FD is neither trivial, nor is the lhs a superkey. OOPS!
    - the key is (*SSN, Hobby*)
- HasAccount (*AcctNum, ClientId, OfficeId*)
  - If the FDs are AcctNum $\rightarrow$ OfficeId and ClientId, OfficeId $\rightarrow$ Acctnum, then the relation is not in BCNF
    - (ClientId, OfficeId) is a superkey; one FD is OK!
    - AcctNum is not a superkey; it does not determine ClientId; OOPS!
    - A relation is in BCNF when every FD is trivial or has superkey on lhs
    - Candidate keys are (*ClientId, OfficeId*) and (*AcctNum, ClientId*)

Binary relation

- If a relation has TWO attributes, it is in BCNF
- R(A,B)
  - What are possible fds?
  - Suppose $A \rightarrow B$
    - Then A is superkey (actually candidate key)
  - Suppose $B \rightarrow A$
    - Then B is superkey (actually candidate key)
  - Suppose no (non-trivial) fd
    - Then ever non-trivial y fd has lhs as candidate key
    - Because there aren’t any to check!

Normal Form Exercise

Mining relation



- Is this schema in BCNF?

Other normal forms

Not examinable

- **Design theory has provided other definitions, which can be used in some situations**
  - First normal form (1NF)
  - Second normal form (2NF)
  - Third normal form (3NF)
  - Fourth normal form (4NF) etc
- But we concentrate on BCNF!

Today's Agenda

- Motivation
- Making it precise: Functional Dependency
- Making it precise: Normal Form
- Making it precise: Schema Decomposition
  - Purpose: normalization

Decomposition of Relational Schema

- Suppose that relation R contains attributes *A1 ... An.*
  A *decomposition* of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
  - Every attribute of R appears as an attribute of one or more of the new relations.
  - Note: usually, some columns will appear in more than one of the decomposed relations
    - It is a common mistake, to simply split up the attributes into groups with no overlap between them
- E.g., Can decompose
  OrderPlus(*date, cid, pid, descr, price, weight, amount)*   *with* fds
  date cid pid $\rightarrow$ *amount*, *pid* $\rightarrow$ *descr price weight*
- into Order(*date*,*cid*,*pid*,*amount*) and Product (*pid*,*descr*,*price*,*weight*).

Choice of Decomposition

- We can decompose OrderPlus(*date, cid, pid, descr, price, weight, amount)* into Order(*date*,*cid*,*pid*,*amount*) and Product (*pid*,*descr*,*price*,*weight*).
- How do we know we should decompose as above, and not as O1(*date, pid, cid, amount*) and O2(*date, desc, price, weight*)???
- There are properties which a decomposition may have (or may not have)
- The decomposition may be lossless-join, or it may not
  - We will insist that any useful decomposition have the lossless-join property
- The decomposition may be dependency-preserving, or it may not
  - We hope to find a decomposition that is dependency-preserving

Decomposition and the Data

- Suppose we replace R by a decomposition into R1 (based on columns X) and R2 (based on columns Y)
  - Each instance in the new schema has a subset of the columns in the old instance
  - Notation: R1 = $\prod_X (R)$, R2 = $\prod_Y (R)$
- The information we have about the world in the new schema is what we can deduce from the instance of R1 and the instance of R2
- These are all the facts which are produced by joining the two tables
  - That is, take a tuple from R1 and a tuple from R2, which agree in the values for the common attributes
  - Recall notation: R1 $\bowtie$ R2
- How does this compare to the information in the original relation R, before decomposition?
  - Answer: it always has the information in the original, but it may have introduced extra (spurious) information
  - Whether this has happened depends on the choice of decomposition, and the constraints on the domain

Example for Decomposition with spurious tuples in join

OrderPlus

date	cid	pid	descr	price	weight	amount
03.05	1001	1	Paper	20.0	2.0	100
03.05	1001	6	Disks	5.0	0.4	50

O1

date	cid	pid	amount
03.05	1001	1	100
03.05	1001	6	50

O2

date	descr	price	weight
03.05	Paper	20.0	2.0
03.05	Disks	5.0	0.4

O1 $\bowtie$ *O2*

date	cid	pid	descr	price	weight	amount
03.05	1001	1	Paper	20.0	2.0	100
03.05	1001	6	Disks	5.0	0.4	50
03.05	1001	1	Disks	5.0	0.4	100
03.05	1001	6	Paper	20.0	2.0	50

Tuples in join of decomposed schema, but not in original data

Definition: Lossless-Join Decomposition

- Decomposition of *R* (with fds F) into R1 based on columns *X* and R2 based on columns *Y* is *lossless-join* means *for every instance R* that satisfies *F*:
  - $\Pi_X(R) \bowtie \Pi_Y(R) = R$
- It is always true that $R \subseteq \Pi_X(R) \bowtie \Pi_Y(R)$
  - In general, the other direction does not hold! If it does for every instance, the decomposition is lossless-join.
- Definition extended to decomposition into 3 or more relations in a straightforward way.
- *It is essential that all decompositions used to deal with redundancy be lossless, so that we are not losing knowledge by having a database which implies spurious (incorrect) statements!*

What is lost?

- In the decomposed tables, the join has *extra* tuples
  - So why do we call the decomposition “not lossless”?
- Knowledge has been lost
  - We no longer know that certain combinations of facts are untrue!
  - Eg we lose knowledge that there is NOT an OrderPlus tuple with (03.05, 1001, 1, Disks, 5.0, 0.4, 100)

$O1 \bowtie O2$						
date	cid	pid	descr	price	weight	amount
03.05	1001	1	Paper	20.0	2.0	100
03.05	1001	6	Disks	5.0	0.4	50
03.05	1001	1	Disks	5.0	0.4	100
03.05	1001	6	Paper	20.0	2.0	50

Checking Lossless-Join Decomposition

You need to be able to do this calculation

- The definition is based on considering every possible instance
  - Instead, we use the following theorem
- **Theorem:** The decomposition of *R* (with fds *F*) into R1 based on columns X and R2 based on columns Y is lossless-join if and only if the fds F imply either
  - $X \cap Y \rightarrow X$, or
  - $X \cap Y \rightarrow Y$,
  - or both these
- That is, we have lossless join decomposition, when it is a decomposition (X and Y between them include all columns of R) and also the shared columns between R1 and R2 form a superkey of at least one of the new relations

- **Example:** OrderPlus(*date, cid, pid, descr, price, weight, amount*) with fds:
  - F = {*date cid pid* $\rightarrow$ *amount*, *pid* $\rightarrow$ *descr price weight* }
- If we decompose as O1(*date, pid, cid, amount*) and O2(*date, desc, price, weight*)
  - $X \cap Y$ is {date}
  - F^+ does not contain *date* $\rightarrow$ *descr price weight*
    - Check this by use chase to find *date+ ;* are *descr, price, weight* all in *date+* ?
  - F^+ does not contain *date* $\rightarrow$ *cid pid amount*
- So decomposing *OrderPlus* into *O1* and *O2,* is **not** a lossless-join decomposition

Dependency-Preserving Decomposition

- Consider CSJDPQV, C is key, JP → C and SD → P.
  - therefore JSD → C so,
  - BCNF decomposition: CJSDQV and SDP
  - Problem: Checking JP → C requires a join!

- Intuitive:
  - If *R* is decomposed into *S* and *T*, and we enforce the FDs that hold on *S* and those that hold on *T*, then all FDs that were given to hold on *R* must also hold.

- *Projection of set of FDs F*:
  If a relational schema ***R*** is decomposed into ***S***,*T* …, the **projection of *F* onto *S*** (denoted F_S) is the set of
  FDs U → V in F^+ (*closure of F*) such that *U, V* are in ***S***.

Dependency Preserving Decomposition Definition

- Given a relation *R* with schema ***R*** and set of FDs *F*.
  A decomposition of ***R*** into ***S*** and ***T*** is a dependency preserving decomposition if
  $$(F_S \cup F_T)^+ = F^+$$
  - i.e., if we consider only dependencies in the closure F^+ that can be checked in ***S*** without considering ***T***, and in ***T*** without considering ***S***, these imply all dependencies in F^+.

- It is important to consider F^+, not F, in this definition:
  - ABC, F={A $\rightarrow$ B, B $\rightarrow$ C, A $\rightarrow$ C}, decomposed into AB and BC.
  - Is this dependency preserving? Is A $\rightarrow$ C preserved?????
  - As F_{AB}={A $\rightarrow$ B}, F_{BC}={B $\rightarrow$ C}, then ($F_{AB} \cup F_{BC}$) ={A $\rightarrow$ B, B $\rightarrow$ C}
  - So F $\neq$ ($F_{AB} \cup F_{BC}$)
  - However A $\rightarrow$ C is in $(F_{AB} \cup F_{BC})^+$
  - This decomposition is dependency preserving: F= $(F_{AB} \cup F_{BC})^+$
- Dependency preserving does not imply lossless join, and vice-versa!
  - Lossless-join is more important if we can't have both

Examples for Dependency-Preserving

- The decomposition of OrderPlus(*date, cid, pid, descr, price, weight, amount*) with F = {*date cid pid* $\rightarrow$ *amount, pid* $\rightarrow$ *descr price weight* } into
  - O3(*date*,*cid*,*pid*,*amount*) with *F1* = {*date cid pid* $\rightarrow$ *amount*} and
  - O4 (*pid*,*descr*,*price*,*weight*) with *F2* = {*pid* $\rightarrow$ *descr price weight*}

  is dependency-preserving.

- The decomposition of Lectures (*course*, *time*, *room*) with *F* = {*course* $\rightarrow$ *room*, *time room* $\rightarrow$ *course*} into
  - Rooms(*course, room*) with F1={*course* $\rightarrow$ *room*} and
  - Times(*course, time*) with F2 = {}

  is not dependency-preserving.

Central Facts of Normalization

- Every relation *R* with set of FDs *F can be* decomposed into **BCNF** relations which is a *lossless-join*.
  - Unfortunately, there may be no dependency-preserving decomposition into a collection of BCNF relational schema

Algorithm for BCNF Normalization

You need to be able to do this calculation

In general, several dependencies may cause violation of BCNF…

- Consider relation *R* with FDs *F,* which is not in BCNF
  Take some fd *X* → *Y* which violates BCNF (because X is not a superkey of R),
  - clean it up, by removing from Y any columns which are already in X
  - then decompose *R* into *R1(R – Y)* and *R2(XY).*
  - Recall: X and Y are sets of columns
  - R – Y means: all columns of R except those in Y
  - XY means: the columns in X together with columns in Y
- This decomposition is lossless-join, because X is the set of common columns, and X is a superkey of R2
- Repeated application of this step, as long as some relation in the schema is not in BCNF, will eventually give us a collection of relations that are all in BCNF; each step has been a lossless join decomposition.

- The order in which we "deal with" them could lead to very different sets of relations!

Example: BCNF Decomposition

- Given the following schema:
  course-schema (*cid, title, cpoints, workload, lecturer, sid, grade)*
- *F* = { *cid* $\rightarrow$ *title cpoints lecturer,*
  cpoints $\rightarrow$ *workload,*
  cid sid $\rightarrow$ *grade* }
- *{cid}*$^+$ = *{ cid, title, cpoints, lecturer, workload}*
  {cpoints}$^+$ = *{ cpoints, workload }*
  {cid, sid}$^+$ = *{cid, sid, title, cpoints, lecturer, workload, grade}*
- *cid* is not superkey; *cpoints* is not superkey
- Candidate Key: { *cid, sid* }

Example: BCNF Decomposition (cont'd)

- *Course-schema* is not BCNF, because, e.g., first and second FD violate BCNF
- Decompose *Course-schema* using *cpoints* → *workload* first
  - *Workloads (cpoint, workload) with*
    cpoints → *workload,*
  - *Courses-schema2* (*cid, title, cpoints, lecturer, sid, grade) with*

    cid → *title cpoints lecturer,*

    cid sid → *grade*

Example: BCNF Decomposition (cont’d)

- *Course-schema2* is still not BCNF; further decompose using *cid → title cpoints lecturer*
  - *Workloads (cpoints, workload) with F2 = {cpoints → workload}*
  - *Courses* (*cid, title, cpoints, lecturer)*

    with F1 ={cid → title cpoints lecturer }
  - *Enrolled* (*cid, sid, grade) with F3 = {cid sid → grade}*
- This decomposition is lossless-join & dependency-preserving

Example: BCNF Decomposition (cont'd)

- If you decompose along *cid* → *title cpoints lecturer* **first**, there is a different schema!
  - *Courses* (*cid, title, cpoints, lecturer),*

    with F1 ={cid → title cpoints lecturer }
  - *Workload_enroll*(*cid, workload, sid, grade),*

    with F3 = {cid sid → grade}
  - *Can we decompose any of above schema further?*
  - *How about FD cpoints → workload?*

Another Example

Given: R = ($R; F$) where $R = ABCDEGHK$ and

$F = \{ABH \rightarrow C, A \rightarrow DE, BGH \rightarrow K, K \rightarrow ADH, BH \rightarrow GE\}$

step 1: Find a FD that violates BCNF

Not $ABH \rightarrow C$ since $(ABH)^+$ includes all attributes

(BH is a key)

$A \rightarrow DE$ violates BCNF since A is not a superkey ($A^+ = ADE$)

step 2: Split R into:

$R_1 = (ADE, F_1 = \{A \rightarrow DE\})$

$R_2 = (ABCGHK; F_1 = \{ABH \rightarrow C, BGH \rightarrow K, K \rightarrow AH, BH \rightarrow G\})$

Note 1: R_1 is in BCNF

Note 2: Decomposition is *lossless* since A is a key of R_1.

Note 3: FDs $K \rightarrow D$ and $BH \rightarrow E$ are not in F_1 or F_2. But both can be derived from $F_1 \cup F_2$

($E.g.$, $K \rightarrow A$ and $A \rightarrow D$ implies $K \rightarrow D$)

Hence, decomposition *is dependency preserving*.

Another Example (con't)

Given: $R_2 = (ABCGHK;\ \{ABH \rightarrow C,\ BGH \rightarrow K,\ K \rightarrow AH,\ BH \rightarrow G\})$

step 1: Find a FD that violates BCNF.

Not $ABH \rightarrow C$ or $BGH \rightarrow K$, since BH is a key of R_2

$K \rightarrow AH$ violates BCNF since K is not a superkey ($K^+ = AH$)

step 2: Split R_2 into:

$R_{21} = (KAH,\ F_{21} = \{K \rightarrow AH\})$

$R_{22} = (BCGK;\ F_{22} = \{\})$

Note 1: Both R_{21} and R_{22} are in BCNF.

Note 2: The decomposition is *lossless* (since K is a key of R_{21})

Note 3: FDs $ABH \rightarrow C$, $BGH \rightarrow K$, $BH \rightarrow G$ are not in F_{21} or F_{22}, and they can't be derived from $F_1 \cup F_{21} \cup F_{22}$.
Hence the decomposition is *not* dependency-preserving

References

- Silberschatz/Korth/Sudarshan(7ed)
  - Chapter 7.1-7.5
  - These sections also include 3NF decomoposition, which we do not require

Also

- Kifer/Bernstein/Lewis(complete version, 2ed)
  - Chapter 6.1-6.7
- Ramakrishnam/Gehrke(3ed)
  - Chapter 19.1-19.6
  - These sections also include 3NF decomoposition, which we do not require
- Garcia-Molina/Ullman/Widom(complete book, 2ed)
  - Chapter 3.1-3.4

Summary

- Motivation: avoid redundancy
- Functional Dependency
  - Capture constraints
  - Calculate closure of set of attributes
  - Work with definitions
- Normal Forms
  - BCNF
  - Check whether schema is in BCNF, based on FDs
- Decomposition
  - Lossless-join and dependency-preserving?
  - Normalize by doing lossless-join decomposition till all relations are BCNF