Database design and quality

Lecture topics:

- measuring the quality of a schema
- schema design with normalization and normal forms

References:

- text 3rd edition, Chapter 14: sections 1, 2.1, 4.2, 4.3, 5, 6; supplementary 2.2-2.4
- text 4th edition, Chapter 10: sections 1, 2.1, 4.2, 5, 6; supplementary 2.2-2.4

Normal forms

- What is a good relational database schema? How can we measure or evaluate a relational schema?

- Goals:
  - intuitive and straightforward retrieval and changes
  - nonredundant storage of data

- Normal forms:
  - Boyce-Codd Normal Form (BCNF)
  - Third Normal Form (3NF)

Design anomalies

- Consider:

Supplied_Items

Sno	Sname	City	Phone	Ino	Iname	Price
S1	Magna	Ajax	416 555 1111	I1	Bolt	0.50
S1	Magna	Ajax	416 555 1111	I2	Nut	0.25
S1	Magna	Ajax	416 555 1111	I3	Screw	0.30
S2	Delco	Hull	613 555 2222	I3	Screw	0.40

Typical operations:

- change vendor's phone number
- add a new supplier (no items yet)
- cease getting "I3" from "S2"
- add a new part (no supplier yet)

...continued

Supplied_Items

Sno	Sname	City	Phone	Ino	Iname	Price
S1	Magna	Ajax	416 555 1111	I1	Bolt	0.50
S1	Magna	Ajax	416 555 1111	I2	Nut	0.25
S1	Magna	Ajax	416 555 1111	I3	Screw	0.30
S2	Delco	Hull	613 555 2222	I3	Screw	0.40

Discussion:

- redundancy: duplicated data, wasted space and time
- update anomaly: update all copies of data, corrupt otherwise
- insert anomaly: cannot insert without complete information
- delete anomaly: deletion may unintentionally remove useful data
- functional dependencies: attributes that “go together”

...continued

- compare the preceding with:

Supplier

Sno	Sname	City	Phone
S1	Magna	Ajax	416 555 1111
S2	Delco	Hull	613 555 2222

Supplies

Sno	Ino	Iname	Price
S1	I1	Bolt	0.50
S1	I2	Nut	0.25
S1	I3	Screw	0.30
S2	I3	Screw	0.40

- universal table has been *decomposed*
- Supplier table has only supplier data
- some anomalies gone, some remain

...continued

- finally, compare with:

Supplier

Sno	Sname	City	Phone
S1	Magna	Ajax	416 555 1111
S2	Delco	Hull	613 555 2222

Item

Ino	Iname
I1	Bolt
I2	Nut
I3	Screw

Supplies

Sno	Ino	Price
S1	I1	0.50
S1	I2	0.25
S1	I3	0.30
S2	I3	0.40

- Supplies table decomposed further
- all anomalies gone
- intuitive arrangement (!?)
- functional dependencies like primary keys

...continued

- extreme decomposition is undesirable (information about relationships is lost)

Snos

Sno
S1
S2

Snames

Sname
Magna
Delco

Cities

City
Ajax
Hull

Phones

Phone
416 555 1111
613 555 2222

Inums

Inum
I1
I2
I3

Inames

Iname
Bolt
Nut
Screw

Prices

Price
0.50
0.25
0.30
0.40

- this is a “lossy” decomposition – what we want is “lossless” (more later)

Good database design

- What is a “good” relational database schema?
- Rule of thumb: Independent facts in separate tables
- or: Each relation schema should consist of a primary key and a set of mutually independent attributes

Functional dependencies

- Generalizes notion of superkey, used to characterize BCNF and 3NF

- Notation for tuple projection: reference the tuples as *t*, *u* etc.

Supplier

Sno	Sname	City	Phone
S1	Magna	Ajax	416 555 1111
S2	Delco	Hull	613 555 2222

If the first tuple in Supplier is labelled *t*, then:

t [Sno] = (S1)

t [Sname, City] = (Magna, Ajax)

...continued

- Consider another example schema:

EmpProj

SIN	PNum	Hours	EName	PName	PLoc	Allowance

- Primary key constraint applies to entire rows; forbids two different rows *t* and *u* in EmpProj with *t* [SIN, PNum] = *u* [SIN, PNum]
  - SIN, PNum $\rightarrow$ Hours, Ename, Pname, *etc*
- But also want to disallow within row:
  - two employees with one SIN
  - one project number with two project names or two locations
  - different allowances for the same number of hours at the same location
- Use functional dependencies to describe:

  SIN $\rightarrow$ EName

  PNum $\rightarrow$ PName, PLoc

  PLoc, Hours $\rightarrow$ Allowance

...continued

- FDs can predict anomalies. Consider:

Supplied_Items

Sno	Sname	City	Phone	Ino	Iname	Price

Sno $\rightarrow$ Sname, City, Phone
Ino $\rightarrow$ Iname
Sno, Ino $\rightarrow$ Price

- Some indications:
  - Sno is part of (not the entire) relation superkey, and is also the entire left side of an FD
    - deleting Sno information by itself would be impossible
  - Sno appears on the left of more than one FD
    - adding just Sno information means some other information is missing

Formal definitions

- Let R be a relation schema, and X, $Y \subseteq R$
- The **functional dependency** (FD)

  $$X \rightarrow Y$$

  holds on R if no legal instance of R contains two tuples t and u with $t[X] = u[X]$ and $t[Y] \neq u[Y]$
- *X functionally determines Y,*
  Y is functionally dependent on X
- $K \subseteq R$ is a superkey for relation schema R if dependency $K \rightarrow R$ holds on R

Boyce-Codd Normal Form (BCNF)

- Formalization of the goal that independent relationships are stored in separate tables

- Let R be a relation schema and F a set of functional dependencies. A functional dependency $X \rightarrow Y$ is **trivial** if $Y \subseteq X$.

- Schema R is in **BCNF** if and only if whenever $(X \rightarrow Y) \in F^+$ and $XY \subseteq R$, then either
  - $(X \rightarrow Y)$ is trivial, or
  - X is a superkey of R

- A database schema $\{R_1, ..., R_n\}$ is in BCNF if each relation schema R_i is in BCNF

...continued

- How does BCNF avoid redundancy?
- For schema Supplied_Items we had

  FD: Sno $\rightarrow$ Sname, City, Phone
- Implies: “Magna”, “Ajax”, “416 555 1111” must be repeated for each item supplied by supplier S1.
- Assume FD holds over a schema *R* that is in BCNF. This implies
  - Sno is a superkey for *R*
  - each Sno value appears on one row only

A design method

- To create a “good” database schema, define all tables in BCNF
- Done!

- Problems:
  1. what to do with existing schemas that are not BCNF?
  2. is BCNF always possible?

- Answers:
  - decompose existing schemas so that they are BCNF
  - theoretically yes, but practically no ☹

Decomposing a schema

- Let R be a relation schema (set of attributes). Collection $\{R_1, ..., R_n\}$ of relation schemas is a decomposition of R if

$$R = R_1 \cup R_2 \cup ... \cup R_n$$

- A good decomposition:
  - eliminates redundancy (BCNF)
  - minimal number of relations
  - is lossless
  - dependency-preserving

Lossless decompositions

- Also called “lossless-join decompositions”
- Earlier: a decomposition is lossless if a join of all the tables results in the original table
- Formally: A decomposition $\{R_1, R_2\}$ of R is lossless if and only if the common attributes of R_1 and R_2 form a superkey for either schema:

  $R_1 \cap R_2 \rightarrow R_1$ *or*

  $R_1 \cap R_2 \rightarrow R_2$

continued...

- For example, recall Supplier–Item table:

  R = [Sno,Sname,City,Phone,Ino, Iname, Price]

- Consider the first decomposition:

  R_1=[Sno,Sname,City,Phone];

  R_2=[Sno,Ino,Iname,Price]

- $R_1 \cap R_2$ is (Sno) and Sno $\rightarrow R_1$, so this is a lossless decomposition
  - similarly for decomposing R_2 into R_3=[Ino,Iname] and R_4=[Sno,Ino,Price]

continued...

- However, for R_1=(Sno); R_2=(Sname); R_3=(City), etc.:
  - since $R_i \cap R_j = \emptyset \quad \forall$ i,j (i≠j), this is lossy ($\emptyset$ cannot $\rightarrow$ anything)
- Re-joining parts of a lossy decomposition creates *spurious tuples*
  - e.g., consider R_1 join R_2:

R_1

Sno
S1
S2

R_2

Sname
Magna
Delco

  - this will create tuples with values (S1,Magna), (S1,Delco), (S2,Magna), (S2,Delco)
  - (S1,Delco) and (S2,Magna) do not exist anywhere in the original relation and are spurious

Dependency preservation

- Informally: ensuring that constraints (i.e. FDs) are represented efficiently in a decomposition
- Practical goal: testing FDs is efficient if the are in a single table, expensive if they require joining tables

- **E.g.**
  Relation R = [A, B, C];
  FD set $F = \{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$
- Consider two decompositions:
  - $D_1 = \{ R_1[A, B], R_2[B, C] \}$
  - $D_2 = \{ R_1[A, B], R_3[A, C] \}$
  - F still applies to both D_1 and D_2

...continued

- In D_1:
  $A \rightarrow B$ can be tested easily in R_1
  $B \rightarrow C$ can be tested easily in R_2
  $A \rightarrow C$ is automatic (FD implication)
- In D_2:
  $A \rightarrow B$ can be tested easily in R_1
  $A \rightarrow C$ can be tested easily in R_3
  $B \rightarrow C$ **cannot** be tested easily
  - $B \rightarrow C$ is an **interrelational constraint**: to test, must join tables R_1 and R_3

- Let R be a relation schema and F a set of functional dependencies on R. A decomposition $D = \{R_1, ..., R_n\}$ of R is **dependency preserving** if F (or an equivalent to F) contains no interrelational constraints.

The plot so far...

- BCNF is good
- Non-BCNF schemas can be decomposed
- Decompositions should be lossless and dependency-preserving
- Lossless BCNF decompositions always exist
- Dependency-preserving BCNF decompositions might not!

- Third normal form (3NF) is almost as good as BCNF, and has the advantage that a lossless, dependency-preserving decomposition always exists

Third Normal Form (3NF)

- Let R be a relation schema and F a set of functional dependencies
- Schema R is in **3NF** if and only if whenever $(X \rightarrow Y) \in F^+$ and $XY \subseteq R$, then either
  - $(X \rightarrow Y)$ is trivial, or
  - X is a superkey of R, or
  - each attribute of Y is contained in a candidate key of R
- A database schema $\{R_1, ..., R_n\}$ is in 3NF if each relation schema R_i is in 3NF
- Any schema that is BCNF is already 3NF
- Because 3NF is less restrictive than BCNF, it allows more redundancy

...continued

- **E.g.**
  Relation *Delivery* =[time, supplier, carrier]
  FD set *F*:
  - supplier $\rightarrow$ carrier
  - time, carrier $\rightarrow$ supplier
- Example instance:

time	carrier	supplier
overnight	fedx	s1
2-day	fedx	s1
overnight	ups	s2
bulk	ups	s2

- *Delivery* is not BCNF, but is 3NF
- Is a BCNF decomposition possible?

Summary

- Formal algorithm exists to compute a BCNF decomposition
  - guarantees losslessness
  - does not guarantee dependency preservation
  - computationally expensive
- Formal algorithm exists to compute 3NF
  - guarantees losslessness
  - guarantees dependency preservation
  - computationally efficient
- If a “good” BCNF decomposition does not exist, use 3NF
- In practice, dependency preservation is important for efficiency