Database Management System

Lecture 4

Database Design – Normalization and View

* Some materials adapted from R. Ramakrishnan, J. Gehrke and Shawn Bowers

Today's Agenda

- Normalization
- View

Normalization

Normalization

- Process or replacing a table with two or more tables

EmpDept

EID	Name	Dept	DeptName
A01	Joshua	12	CS
A12	Bean	10	HR
A13	Bean	12	CS
A03	Kevin	12	CS

Which schema is better?
Why?

Vs.

Emp

EID	Name	Dept
A01	Joshua	12
A12	Bean	10
A13	Bean	12
A03	Kevin	12

Dept

DeptID	DeptName
10	CS
12	HR

Normalization Issues

- The EmpDept schema combines two different concepts
  - Employee information, together with
  - Department information
- To join or not to join that is the question
  - If we separate the two concepts we could save space but some queries would run slower (Joins)
  - If we combine the two ideas we have redundancy but some queries would run faster (no Joins)
- So we have a tradeoff ...
- Redundancy has a side effect: “*anomalies*”

Types of Anomalies

EmpDept

EID	Name	Dept	DeptName
A01	Joshua	12	CS
A12	Bean	10	HR
A13	Bean	12	CS
A03	Kevin	12	CS

- "*Update Anomaly*": If the CS department changes its name, we must change multiple rows in EmpDept
- "*Insertion Anomaly*": If a department has no employees, where do we store its id and name?
- "*Deletion Anomaly*": If A12 quits, the information about the HR department will be lost
- These are in addition to redundancy in general
  - For example, the department name is stored multiple times

Using NULL Values

EmpDept

EID	Name	Dept	DeptName
A01	Joshua	12	CS
NULL	NULL	10	HR
A13	Bean	12	CS
A03	Kevin	12	CS

- Using NULL values can help insertion and deletion anomalies
- But NULL values have their own issues
  - They make aggregate operators harder to use
  - Not always clear what NULL means
  - May need outer joins instead of ordinary joins
  - In this case, EID is a primary care, and so it cannot contain a NULL value!
- They don’t address update anomalies or redundancy issues

Decomposition

Emp

EID	Name	Dept
A01	Joshua	12
A12	Bean	10
A13	Bean	12
A03	Kevin	12

Dept

DeptID	DeptName
10	CS
12	HR

- Normalization involves *decomposing* (partitioning) the table into separate tables
- Check to see if redundancy still exists (... repeat)
  - The key to understanding when and how to decompose schemas is through ... *"functional dependencies"*
  - which generalizes the notion of keys

Keys

EmpDept

EID	Name	Dept	DeptName
A01	Joshua	12	CS
A12	Bean	10	HR
A13	Bean	12	CS
A03	Kevin	12	CS

- Because EID is a key:
  - If two rows have the same EID value, then they have the same value for every other attribute
  - Thus given an EID value, the other values are "determined"
- A Key is like a "function":
  - f : EID → Name × Dept × DeptName – E.g., f(A01) = <Joshua, 12, CS>
- Recall functions always return the same value for a given value

Functional dependencies

EID	Name	Dept	DeptName
A01	Joshua	12	CS
A12	Bean	10	HR
A13	Bean	12	CS
A03	Kevin	12	CS

- We say that EID "functionally determines" all other attributes
- This relationship among attributes is called a "*Functional Dependency*" (FD)
- We write FDs as:

  EID → Name, Dept, DeptName

  or

  EID → Name, EID → Dept, EID → DeptName

FDs that are not implied by keys

<u>EID</u>	Name	Dept	DeptName
A01	Joshua	12	CS
A12	Bean	10	HR
A13	Bean	12	CS
A03	Kevin	12	CS

- Is Name → Dept a functional dependency?
  - No, e.g., <Bean, 10> and <Bean, 12>
- Is Dept → DeptName a functional dependency?
  - Yes in this table it is
  - In general, it would be expected that departments only have one name

Functional Dependencies

- For *sets* A and B of attributes in a relation, we say that A (functionally) determines B ... or A → B is a Functional Dependency (FD)
  - if whenever two rows agree on A they also agree on B
- An FD defines a function in the "mathematical sense"
- There are two *special kinds* of FDs:
  - *"Key FDs"* of the form X →A where X contains a key (X is called a *superkey*)
  - *"Trivial FDs"* of the form A → B such that A ⊇ B
    - ... e.g., (Name, Dept) → Dept
    - these are boring but become important later

Functional Dependencies

- Functional dependencies, like keys, are based on the semantics of the application
- *Likely* functional dependencies:
  - ssn → name
  - account → balance
- *Unlikely* functional dependencies:
  - date → trasactionid
  - checkamt -> checknumber

Enforcing Functional Dependencies

- For the table

Emp(eid, name, dept, deptname)

- There is an FD from dept → deptname

- Although eid is the key for this table ...
  - ... is it still possible for there to be two names for the same department?
  - YES!

Every Key Implies a Set of FDs

- For the table

  Emp(eid, name, dept, deptname) `

- We have the following FDs based on ssn *being a key*:
  - eid → name
  - eid → dept
  - eid → deptname
- Each key implies a set of functional dependencies from the key to the non-key attributes

Functional Dependencies and Keys

- Given a table R with attributes *a* and *b* together forming a key, the following FDs are implied
  - Given R(*a*, *b*, *c*, *d*, *e*)

$$ab \rightarrow c$$
$$ab \rightarrow d$$
$$ab \rightarrow e$$

  - Which we can also write as $ab \rightarrow cde$

Functional Dependencies May Suggest Keys

- If we know these FDs:

ssn → name
ssn → hiredate
ssn → phone

- then **ssn is a key** for a table with these attributes:

Employee(ssn, name, hiredate, phone)

What are the key and non-trivial FDs?

- Which of these will be enforced?

Customer(<u>CustID</u>, Address, City, Zip, State)

Enrollment(<u>StdntID, ClassID</u>, Grade, InstrID, StdntName, InstrName)

Non-Trivial Functional Dependencies

- The FDs that are not enforced by the DBMS lead to both redundancy and anomalies (only keys are enforced)
- Not all redundancy is covered by FDs

  Emp(<u>ssn</u>, name, salary, birthdate)
  Employee(<u>ssn</u>, name, address)

  - name stored redundantly, and same employee can have more than one name
- Cannot be determined from the instance (instead, based on application semantics)
  - We can determine what *is not* an FD
  - DB data mining approaches infer “FDs” (i.e., association rules)

Example Decomposition based on FDs

- For this table

Emp(<u>ssn</u>, name, birthdate, address, dnum, dname, dmgr)

- We can *move* the non-trivial FDs into their own table with dnum as the key:

Dept(<u>dnum</u>, dname, dmgr)

- The Emp table becomes:

Emp(<u>ssn</u>, name, birthdate, address, dept)

- ... and Emp.dept is now a foreign key to Dept.dnum

Normalization based on FDs

- Identify all all the FDs
  - FDs implied by the keys
  - FDs not implied by the keys (the “troublesome” ones)
- Generate one or more new tables from the FDs not implied by the keys
  - Each new tables should only have FDs implied by the key
- Remove the attributes from original table that are functionally dependent on “troublesome” FDs
- Specify appropriate foreign keys to these new tables

Reasoning about Functional Dependencies

EmpDept(EID, Name, DeptID, DeptName)

- Two natural FDs are
  - EID → DeptID and DeptID → DeptName
- These two FDs imply EID → DeptName
- If two tuples agree on EID, then by EID → DeptID they agree on DeptID ...
  - ... and if they agree on DeptID, then by DeptID → DeptName they agree on DeptName
- The set of FDs implied by a given set F of FDs is called the closure of F ... which is denoted F^+

Armstrong's Axioms

- The closure F+ of F can be computed using these axioms
  - *Reflexivity(재귀)*: If X$\supseteq$Y, then X→Y
  - *Augmentation(부가)*: If X → Y, then XZ → YZ for any Z
  - *Transitivity(이행)*: If X→Y and Y→Z then X→Z
- Repeatedly applying these rules to F until we no longer produce any new FDs results in a *sound and complete* inference procedure ...
- Soundness
  - Only FDs in F^+ are generated when applied to FDs in F
- Completeness
  - Repeated application of these rules will generate all FDs in F^+

Finding Keys

- We can determine if a set of attributes X is a key for s relation R by computing X^+ as follows

Compute X^+ *from* X
let $X^+ = \{X\}$
repeat until there is no change in X^+
{
    if $Y \rightarrow Z$ is an FD and $Y \subseteq X^+$ Then
        $X^+ = X^+ \cup Z$
}
return X^+

- Let the set of attributes of R be A
- X is a key for R if and only if $X^+ = A$

Example

- Given the schema R(A, B, C, D, E) such that

  BC → A
  DE → C

- Find the keys of this schema, besides A ...
- Start with BC → A as one example
  - BC determines A is given
  - A →ABCDE because A is a key
  - BC → ABCDE by *transitivity*
  - Thus, BC is a key!
- You should understand the axioms and the algorithm ...

  they will come in handy when normalizing

Redundancy and Functional Dependencies

- Example schema

  EmpDept(<u>EID</u>, Name, Dept, DeptName)

  Assigned(<u>EmptID, JobID</u>, EmpName, Percent)

  Enrollment(<u>StdntID, ClassID</u>, Grade, InstrID, StdntName, InstrName)

- Note that every non-key FD is associated with some redundancy
- Our game plan is to use non-key and non-trivial FDs to decompose any relation into a form that has no redundancy ...
- ... resulting in a so-called *“Normal Form”*

Boyce-Codd Normal Form (BCNF)

- A relation is in "*Boyce-Codd Normal Form*" if all of its FDs are either
  - Trivial FDs (e.g., AB → A) or
  - Key FDs

- Which (if any) of these relations is in BCNF?

EmpDept(<u>EID</u>, Name, Dept, DeptName)

Assigned(<u>EmptID, JobID</u>, EmpName, Percent)

Enrollment(<u>StdntID, ClassID</u>, Grade, InstrID, StdntName, InstrName)

BCNF and Redundancy

- BCNF relations have no redundancy cause by FDs
  - A relation has redundancy if there is an FD between attributes
  - ... and there can be *repeated* entries of data for those attributes
- For example, consider

DeptID	DeptName
12	CS
10	HR
12	CS

  - if the relation is in BCNF, then the FD must be a key FD, and so DeptID must be a key
  - implying that any pair such as <12, CS> can appear only once!

Decomposition into BCNF

- An algorithm for decomposing a relation R with attributes A into a collection of BCNF relations

if R is not in BCNF and $X \rightarrow Y$ is a non-key FD then
decompose R into $A - Y$ and XY
if $A - Y$ and/or XY is not in BCNF then
recursively apply step 1 (to $A - Y$ and/or XY)

Example

Enrollment(StdntID, ClassID, Grade, InstrID, StdntName)

- First use the non-key FD StdntID → StdntName
- ... which gives the decomposition

Enrollment(StdntID, ClassID, Grade, InstrID)
Student(StdntID, StdntName)

- Now use the non-key FD ClassID → InstrID
- ... which gives the decomposition

Enrollment(StdntID, ClassID, Grade)
ClassInstructor(ClassID, InstrID)
Student(StdntID, StdntName)

- All relations are now in BCNF!

Another Example

- Given the schema

Loans(<u>BranchID, LoanID</u>, Amount, Assets, CustID, CustName)

- and assuming FDs

BranchID → Assets
CustID → CustName

- ... lets Decompose it into BCNF relations

Loans(BranchID, LoanID, Amount, CustID)
Customer(CustID, CustName)
Branch(BranchID, Assets)
– Loans.BranchID REFERENCES Branch.BranchID
– Loans.CustID REFERENCES Customer.CustID

Lossless Decomposition

- Some decompositions may lose information content
- For example, lets say we decomposed:

  Enroll(<u>StdntID, ClassID</u>, Grade)

- into

  StudentGrade(<u>StdntID</u>, Grade)
  ClassGrade(<u>ClassID</u>, Grade)

  - a row (223, A) in StudentGrade implies student 223 received an A in some course
  - and a row (421, A) in ClassGrade means that some student received an A in course 421
  - but now we have no way to recreate the original table!

- This decomposition is "Lossy"

Lossless Decomposition

- A decomposition of a schema with FDs F into attribute sets X and Y is "*lossless*" if for every instance R that satisfies F:

$$R = \pi X(R) \bowtie \pi Y(R)$$

- That is, we can recover R from the natural join of the decomposed versions of R

Example of a Lossless Decomposition

EmpDept

R

EID	Name	Dept	DeptName
A01	Joshua	12	CS
A12	Bean	10	HR
A13	Bean	12	CS
A03	Kevin	12	CS

X = EID, Name, Dept

EID	Name	Dept
A01	Joshua	12
A12	Bean	10
A13	Bean	12
A03	Kevin	12

$\pi_X(R)$

Y = Dept, DeptName

Dept	DeptName
12	CS
10	HR
12	CS
12	CS

$\pi_Y(R)$

$$\pi_X(R) \bowtie \pi_Y(R) = R$$

Example of a Lossy Decomposition

Enroll

R

SID	ClassID	Grade
123	cs223	A
456	cs421	A

X = SID, Grade

SID	Grade
123	A
456	A

$\pi_X(R)$

Y = ClassID, Grade

ClassID	Grade
cs223	A
cs421	A

$\pi_Y(R)$

SID	ClassID	Grade
123	cs223	A
456	cs223	A
123	cs421	A
456	cs421	A

$\pi_X(R) \bowtie \pi_Y(R) \neq R$

Producing Only Lossless Decompositions

- We only want to produce lossless decompositions
- This is easy to guarantee:
- The decomposition of R with respect to FDs F into attributes sets A1 and A2 is *lossless* if and only if A1 ∩ A2 contains a key for either A1 or A2
  - If they have a key in common, they can be joined back together – Note that {StdntID, Grade} ∩ {ClassID, Grade} = {Grade}
  - See page 620 in the text
- This implies that the BCNF decomposition algorithm produces only lossless decompositions
  - In this case F includes the FD X→Y and the decomposition is A1 = A – Y and A2 = X ∪ Y
  - Therefore A1 ∩ A2 = X is a key for X ∪ A

Producing Only Lossless Decompositions

- Given the schema R(S, C, G) with FD SC → G
- Is the decomposition into R1(S, G) and R2(C, G) lossless or lossy? Why?
  - Take the intersection of the two sets {S, G} ∩ {C, G} = G
  - Then determine if G is a key for either table
  - That is, does G → C?
    - NO
  - Does G → S?
    - NO
  - Therefore, this decomposition is lossy!

Dependency Preserving Decompositions

- Decompositions should also *preserve FDs*

- For example
  - Addr, City, State → Zip
  - Zip → State

Emp(EID,Addr, City, State, Zip)

- Consider this decomposition

Emp(EID,Addr, City, Zip)
ZipState(Zip, State)

- Although this is BCNF, it does not preserve the FD
  - Addr, City, State → Zip

- Here are some values

<123, 111 W 1st, Spokane, 99999> <99999,WA>
<456, 111 W 1st, Spokane, 00000> <00000,WA>

Dependency Preserving Decompositions

- Let R be a schema with FDs F and X, Y sets of attributes in R
- A dependency A→ B is in X if all attributes of A and all attributes of B are in X
- The projection FX of dependencies F on attributes X is the closure of the FDs in X
- The decomposition of R into schemas with attributes X and Y is “*dependency preserving*” if $(F_X \cup F_Y)^+ = F^+$

Example

- Consider Emp(Addr, City, State, Zip) with

$$F = \{ \text{Addr, City, State} \rightarrow \text{Zip, Zip} \rightarrow \text{State} \}$$

- If we decompose Emp so that X = {Addr, City, Zip} and Y = {Zip, State} what are the projections FX and FY?

FX = $\varnothing$ (Addr,City,State→Zip not in X, Zip→State not in X)

FY = {Zip → State} (Zip→State is in Y)

- Is X,Y a dependency preserving decomposition?
  - No ... $(\text{Zip} \rightarrow \text{State})^+$ does not contain Addr,City,State → Zip and so it can never recreate F^+

Third Normal Form (3NF)

- Some schemas do not have both a lossless and dependency preserving composition into BCNF schemas
- Every schema has has a lossless dependency preserving decomposition into 3NF ...
- A schema R with FDs F is in 3NF if for every X→Y in F either:
  - X→Y is a trivial FD (i.e., X ⊇ Y)
  - X→Y is a key FD (i.e., X is a superkey
  - Y is a part of some key for R

Definition of BCNF

Third Normal Form (3NF)

- In other words, 3NF allows FDs that only partially (i.e., do not fully) depend on the key ...
- For Emp(Addr, City, State, Zip) with

$$F = \{ \text{Addr, City, State} \rightarrow \text{Zip, Zip} \rightarrow \text{State} \}$$

  - the keys are: (Addr, City, State) and (Addr, City, Zip)
- Although there is no decomposition of this relation into BCNF ...
- This relation is in 3NF!

Wrapping up

- Almost all schemas can be decomposed into BCNF schemas that preserve all FDs
  - But every once in a while we get a schema like the previous one
- So, if we do not have an ideal decomposition (lossless, dependency preserving) into BCNF, we can decompose into 3NF and have a lossless and dependency-preserving schema
  - But with some minor redundancy

View

Views

- A "*view*" is a query that is stored in the database and that acts as a "virtual" table
- For example:

```
CREATE VIEW astudents AS
  SELECT *
  FROM Students
  WHERE gpa > 3.0;
```

- Views can be used just like base tables within another query or in another view

```
SELECT *
FROM astudents WHERE age > 20;
```

Implementing Views

- The DBMS expands (i.e., rewrites) your query to include the view definition

```
SELECT ClassID
FROM astudent S, enrollment E
WHERE S.StdntID = E.StdntID
```

- • This query is expanded to

```
SELECT ClassID
FROM (SELECT * FROM student WHERE gpa >= 4.0) AS S,
        enrollment E
WHERE S.StdntID = E.StdntID;
```

Views for Security

- For a base table:

  Student(StdntID, SSN, Name,Address,Telephone, Email, ...)

- This view gives a "secure" version of the student relation

```
CREATE VIEW sstudent AS
    SELECT StdntID, Name,Address
    FROM Student;
```

- Here, using the view we avoid exposing the SSN, Telephone,Email,etc.

Views for Integration

- Different companies might have different but similar “parts” databases

  PartsCo1(PartID, weight, ...)
  PartsCo2(PartID, weight, ...)

- We can combine these parts DBs into a single version using a view definition
- For instance, if company 1 uses pounds and company 2 uses kilograms for part weights:

```
CREATE VIEW Part AS
(SELECT PartID, 2.2066*weight, ...
FROM PartsCo1)
UNION
(SELECT PartID, weight, ...
FROM PartsCo2);
```

View Update Problem

- Views cannot always be updated unambiguously
- For example, for

  Students(stdntid,gpa,deptid,…)
  Department(deptid,dname,office,head,…)

- And views

CREATE VIEW majorgpa AS SELECT major,
AVG(gpa) FROM Students
GROUP BY major

CREATE VIEW stddept AS
SELECT stdnNd, dname
FROM Students JOIN Department
USING (depNd)

- How do we change the GPA of CS majors from 3.5 to 3.6 using majorgpa?
- How do we delete a row (e.g., <jim, cpsc>) from stddept?

View Update Problem

- A view can in general be updated if
  - It is defined over a single base table
  - It uses only selection and projection
  - It does not use aggregates, group by
  - It does not use DISTINCT
  - It does not use set operations (UNION, INTERSECT, MINUS)
- Different products provide different support for views, especially w.r.t updates
- Many more details not discussed here

Data Independence

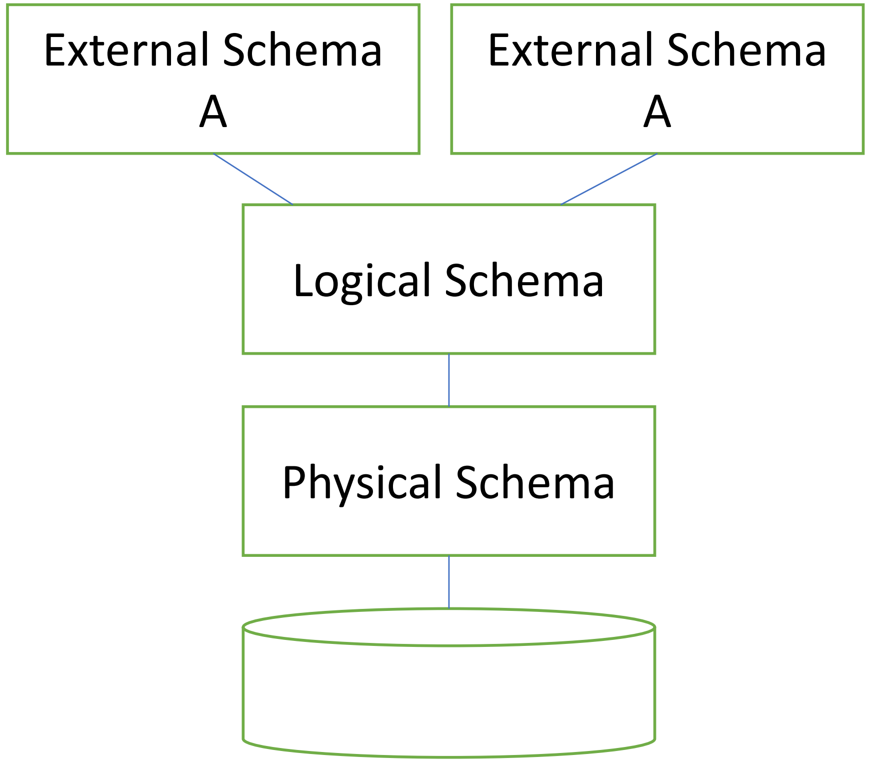
- Multiple levels of abstraction support data independence
  - Changes isolated to their "levels"
  - This is very desirable since things change often!

External View:
What application programmers see

Logical View:
The conceptual or logical relations

Phyisical View:
Optimized/normalized relations including indexes

Phyisical Storage (on disk(s) ...)



For Next Week

- Review – Quiz on the material
  - Ch. 19 to 19.6
- Reading assignments
  - Ch. 19 to 19.6
- Be sure you understand
  - Keys, Functional Dependencies, and Boyce-Codd Normal Form (FD)
  - Normalization, BCNF, 3NF