Recap of Feb 13: SQL, Relational Calculi, Functional Dependencies

- SQL: multiple *group bys, having*, lots of examples
- Tuple Calculus
- Domain Calculus
- Functional Dependencies
  - F+ = the closure of the set of FDs on a given relation
Relational Database Design

- A major goal in designing a database is to have a schema that
  - makes queries simpler (easy to phrase)
  - avoids redundancies and update anomalies (*about which more later*)
Schema and Query Simplicity (1)

Example Schema 1: EMP(eno, ename, sal, dno) and DEPT(dno, dname, floor, mgr)

Query 1: find all employees that make more than their manager

```sql
select e.ename from EMP e, EMP m, DEPT d
where e.dno = m.dno and d.mgr=m.eno and e.sal>m.sal
```

Query 2: for each department, find the maximum salary

```sql
select d.dname, max(e.sal) from EMP e, DEPT d
where e.dno = d.dno group by d.dno
```

Q1 requires two joins; Q2 requires a join and a group-by.
Schema and Query Simplicity (2)

Example Schema 2: (a single relation)

\[
\text{ED(eno, ename, sal, dno, dname, floor, mgr)}
\]

Query 1: find all employees that make more than their manager

\[
\text{select e.ename from ED e, ED m where e.mgr=m.eno and e.sal>m.sal}
\]

Query 2: for each department, find the maximum salary

\[
\text{select d.dname, max(sal) from ED e group by dno}
\]

Q1 requires one join; Q2 requires just a group-by.
Schema and Query Simplicity (3)

- How did we get simpler queries?
- Schema 2 was a more complicated relation with more information; in essence ED was EMP and DEPT from Schema 1 with the join pre-computed
- Should we just precompute the joins and store bigger relations?
- Taken to the extreme, we could compute the universal relation with all attributes inside it and null values for those values that make no sense
- Why wouldn’t we want to do that?

- Problems with too-complex relations: repetition of information (data redundancy) and inability to represent certain information (update anomalies)
DB Design: Redundancy and Anomalies

- Redundancy (repetition of information)
  - each department is repeated for each employee in it
  - great risk of inconsistencies -- suppose the department is moved to a new floor?
  - A simple update (change in mgr name, department floor, etc) in Schema 1 becomes multiple updates in Schema 2

- Anomalies (inability to represent some types of information)
  - departments can’t exist without employees. A department cannot exist until the first employee is inserted, and it can no longer exist when the last employee is deleted from the ED relation
DB Design: Dealing with Anomalies

- So complex relations make for simpler queries, but have the disadvantages of data redundancy and creation of anomalies. How do we balance the two objectives? We want:
  - simple queries
  - no anomalies; minimize data redundancy

- If we start with Schema 2 and discover anomalies we can decompose the relation(s) until the problems go away. This process is called *normalization*. 
Objectives of DB Design (Normalization)

• no redundancy
  – for space efficiency and to reduce the potential for inconsistencies

• update integrity
  – avoid update anomalies

• linguistic efficiency
  – simpler queries are much better for the application programmer and for the query optimizer

• good performance
  – smaller relations imply more joins (bad)
Lossy Decompositions

- Not all decompositions are reversible (*lossless*)

Example:

<table>
<thead>
<tr>
<th>Shipments (S#, P#, J#) decomposed into SP(S#, P#) and SJ(S#, J#)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S#  P#  J#</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>s1  p1  j1</td>
</tr>
<tr>
<td>s2  p2  j1</td>
</tr>
<tr>
<td>s2  p3  j2</td>
</tr>
<tr>
<td>s3  p3  j3</td>
</tr>
<tr>
<td>s4  p4  j3</td>
</tr>
</tbody>
</table>
Lossy Decompositions

Shipment(S#, P#, J#) decomposed into SP(S#, P#) and SJ(P#, J#)

<table>
<thead>
<tr>
<th>S#</th>
<th>P#</th>
<th>J#</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
<td>j1</td>
</tr>
<tr>
<td>s2</td>
<td>p2</td>
<td>j1</td>
</tr>
<tr>
<td>s2</td>
<td>p3</td>
<td>j2</td>
</tr>
<tr>
<td>s3</td>
<td>p3</td>
<td>j3</td>
</tr>
<tr>
<td>s4</td>
<td>p4</td>
<td>j3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S#</th>
<th>P#</th>
<th>J#</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>p2</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>p3</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>p3</td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>p4</td>
<td></td>
</tr>
</tbody>
</table>

If we join SP and SJ again into SP-PJ(S#, P#, P#, J#) we get:

<table>
<thead>
<tr>
<th>S#</th>
<th>P#</th>
<th>P#</th>
<th>J#</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
<td>p1</td>
<td>j1</td>
</tr>
<tr>
<td>s2</td>
<td>p2</td>
<td>p2</td>
<td>j1</td>
</tr>
<tr>
<td>s2</td>
<td>p3</td>
<td>p3</td>
<td>j2</td>
</tr>
<tr>
<td>s3</td>
<td>p3</td>
<td>p3</td>
<td>j2</td>
</tr>
<tr>
<td>s3</td>
<td>p3</td>
<td>p3</td>
<td>j3</td>
</tr>
<tr>
<td>s4</td>
<td>p4</td>
<td>p4</td>
<td>j3</td>
</tr>
</tbody>
</table>

from the joined tuples we cannot deduce the original form of the data. This is called the connection trap and the decomposition is lossy.
Example of Lossy Join Decomposition

- Lossy-join decompositions result in information loss
- Example: decomposition of $R=(A,B)$ into $R_1=(A)$ and $R_2=(B)$

\[
\begin{array}{cc}
R=(A, B) & R_1= (A) & R_2= (B) \\
\text{?} & 1 & ? \\
\text{?} & 2 & ? \\
\text{?} & 1 & \\
\end{array}
\]

$R_1 \times R_2 = (A, B)$

\[
\begin{array}{cc}
\text{?} & 1 \\
\text{?} & 2 \\
\text{?} & 1 \\
\text{?} & 2 \\
\end{array}
\]
Decomposition Continued

• Decompose the relation schema
• All attributes of an original schema (R) must appear in the decomposition (R₁, R₂)
• Lossless (reversible) join decomposition: for all possible relations r on schema R, the decomposition into (R₁, R₂) is lossless if
  \[ r = ?_{R_1}(r) \Join ?_{R_2}(r) \]
• The decomposition of R into R₁ and R₂ is lossless if and only if at least one of the following dependencies is in F⁺:
  \[ R_1 \rightarrow\leftarrow R_2 \rightarrow\leftarrow R_1 \]
  \[ R_1 \rightarrow\leftarrow R_2 \rightarrow\leftarrow R_2 \]
Lossless Join Decomposition and Functional Dependencies

• So FDs can help determine whether a decomposition is lossless

• R is a relation schema and F its FDs. Then a decomposition

$$R = R_1 \cup R_2$$

is lossless if at least one of the following dependencies holds

$$R_1 \rightarrow R_2 \rightarrow R_1$$
$$R_1 \rightarrow R_2 \rightarrow R_2$$

• either of the above FDs guarantees uniqueness in the mapping (and therefore that the decomposition is lossless)
Dependency Preservation

• Dependencies are preserved in a decomposition if we do not need to join in order to enforce FDs -- all FDs remain intra-relational and do not become inter-relational

• To check if a decomposition is dependency preserving, we need to examine all FDs in F+

• There is an algorithm for testing dependency preservation (requires the computation of F+)
Goals of Normalization

• Decide whether a particular relation R is in “good” form

• if it is not in “good” form, decompose it into a set of relations (R₁, R₂, R₃, ..., Rₙ) such that:
  
  – each relation is in “good” form

  – the decomposition is a lossless-join decomposition, based upon functional dependencies
Normalization

• Types of FDs in R(A, B, C, D) with (A, B) a candidate key:
  – trivial:  \( AB \implies A \)
  – partial:  \( A \implies C \)  
    (\( C \) depends upon a part of the key)
    
    TEACH(student, teacher, subject)
    
    student, subject \( \implies \) teacher  
    (students not allowed in the same subject
      of two different teachers)
    
    teacher \( \implies \) subject  
    (each teacher teaches only one subject)
  – transitive:  \( A \implies C \implies D \)
    
    ED\((\text{eno, ename, sal, dno, dname, floor, mgr})\)
    
    eno \( \implies \) dno \( \implies \) mgr
Normalization using FDs

- When we decompose a relation schema R with a set of functional dependencies F into R₁, R₂, R₃, ..., Rₙ we want:
  - lossless-join decomposition: otherwise the decomposition results in loss of information relative to the original schema R
  - no redundancy: the relations Rₙ should be in either BCNF (Boyce-Codd Normal Form) or 3NF (Third Normal Form) (about which more in a slide or two)
  - Dependency preservation: let Fᵢ be the set of dependencies in F+ that include only attributes in Rᵢ:
    - preferably the decomposition should be dependency preserving. That is, F₁ ⊆ F₂ ⊆ F₃ ⊆ ... ⊆ Fₙ = F+
    - Otherwise checking updates for violation of FDs may require computing joins, which is expensive
The Normal Forms

- **1NF**: every attribute has an atomic value
- **2NF**: 1NF and no partial dependencies
- **3NF**: 2NF and no transitive dependencies.

Equivalently (text definition): if for each FD \( X \rightarrow\! Y \) either
  - it is trivial, or
  - \( X \) is a superkey, or
  - \( Y - X \) is a proper subset of a candidate key (each attribute in \( Y \) that isn’t in \( X \) is contained in some candidate key)

- **BCNF**: if for each FD \( X \rightarrow\! Y \) either
  - it is trivial, or
  - \( X \) is a superkey
Distinguishing Examples

- **1NF but not 2NF**: 
  \( \text{SUPPLY}(\text{sno, pno, jno, scity, jcity, qty}) \)
  - (sno, pno, jno) is the candidate key
  - sno ==> scity, jno ==> jcity are both partial dependencies

- **2NF but not 3NF**: 
  \( \text{ED}(\text{eno, ename, sal, dno, dname, floor, mgr}) \)
  - transitive FD: eno ==> dno ==> dname

- **3NF but not BCNF**: 
  \( \text{TEACH}(\text{student, teacher, subject}) \)
  - student, subject ==> teacher
  - teacher ==> subject
Boyce-Codd Normal Form

BCNF is perhaps the most useful Normal Form for database design

A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F+ of the form $X \Rightarrow Y$ where $X \subseteq R$, $Y \subseteq R$; at least one of the following holds:

- $X \Rightarrow Y$ is trivial (that is, $Y \subseteq X$)
- $X$ is a superkey for R
BCNF Example

• \( R = (A, B, C) \)
• \( F = (A \rightarrow \rightarrow B, \)
  \( B \rightarrow \rightarrow C) \)
• \( R \) is not in BCNF
• Decomposition \( R_1 = (A, B), R_2 = (B, C) \)
  – \( R_1 \) and \( R_2 \) are in BCNF
  – Lossless-join decomposition
  – Dependency preserving
Third Normal Form: Motivation

• There are some situations where
  – BCNF is not dependency preserving, and
  – efficient checking for FD violation on updates is important
  – In these cases BCNF is too severe and a looser Normal Form
    would be useful

• Solution: define a weaker Normal Form, called Third
  Normal Form, where
  – FDs can be checked on individual relations without performing a
    join (no inter-relational FDs)
  – There is always a lossless-join, dependency-preserving
    decomposition
Third Normal Form

- A relation schema \( R \) is in 3NF with respect to a set \( F \) of functional dependancies if for all functional dependancies in \( F^+ \) of the form \( X \rightarrow Y \) where \( X \subseteq R \), \( Y \subseteq R \); at least one of the following holds:
  - \( X \rightarrow Y \) is trivial (that is, \( Y \subseteq X \))
  - \( X \) is a superkey for \( R \)
  - Each attribute \( A \) in \( X \rightarrow Y \) is contained in a candidate key for \( R \)
    (note: possibly in different candidate keys)

- A relation in BCNF is also in 3NF

- 3NF is a minimal relaxation of BCNF to ensure dependency preservation
3NF Example

- \( R = (J, K, L) \)
- \( F = (JK\Rightarrow L, \ L\Rightarrow K) \)
- Two candidate keys: JK and JL
- R is in 3NF
  - JK\Rightarrow L \quad JK \text{ is a superkey}
  - L\Rightarrow K \quad K \text{ is contained in a candidate key}
- BCNF decomposition has \( R_1 = (J, L), R_2 = (J, K) \)
  - testing for JK\Rightarrow L requires a join
- There is some redundancy in this schema
Testing for 3NF

- Optimization: need to check only FDs in F, need not check all FDs in F+
- Use attribute closure to check, for each dependency $X \Rightarrow Y$, if $X$ is a superkey
- If $X$ is not a superkey, we have to verify if each attribute in $Y$ is contained in a candidate key of $R$
  - This test is rather more expensive, since it involves finding candidate keys
  - Testing for 3NF has been shown to be NP-hard
  - Interestingly, decomposition into 3NF can be done in polynomial time (testing for 3NF is harder than decomposing into 3NF!)
Comparison of BCNF and 3NF

- It is always possible to decompose a relation into relations in 3NF such that:
  - the decomposition is lossless
  - the dependencies are preserved

- It is always possible to decompose a relation into relations in BCNF such that:
  - the decomposition is lossless
  - but it may not be possible to preserve dependencies
BCNF and 3NF Comparison (cont.)

Example of problems due to redundancy in 3NF

- \( R = (J, K, L) \)

- \( F = (JK \rightarrow L, L \rightarrow K) \)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>j₁</td>
<td>l₁</td>
<td>k₁</td>
</tr>
<tr>
<td>j₂</td>
<td>l₁</td>
<td>k₁</td>
</tr>
<tr>
<td>j₃</td>
<td>l₁</td>
<td>k₁</td>
</tr>
<tr>
<td>null</td>
<td>l₂</td>
<td>k₂</td>
</tr>
</tbody>
</table>

A schema that is in 3NF but not BCNF has the problems of:

- repetition of information (e.g., the relationship between \( l₁ \) and \( k₁ \))
- need to use null values (e.g., to represent the relationship between \( l₂ \) and \( k₂ \) when there is no corresponding value for attribute \( J \))
Design Goals

• Goal for a relational database design is:
  – BCNF
  – Lossless Join
  – Dependency Preservation

• If we cannot achieve this, we accept one of
  – lack of dependency preservation (or use of more expensive inter-relational methods to preserve dependencies)
  – data redundancy due to use of 3NF

• Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys
  – can specify FDs using assertions, but they are expensive to test
  – Even if we have a dependency preserving decomposition, using SQL we cannot efficiently test an FD whose left hand side is not a key
BCNF and Over-normalization

- Goal is to obtain schemas that are:
  - BCNF
  - Lossless Join
  - Dependency Preserving
- but sometimes we have to look at the meaning, too

Example: TEACH(student, teacher, subject)

student, subject ==> teacher  (students not allowed in the
                                 same subject of two teachers)

teacher ==> subject           (each teacher teaches one subject)

- This 3NF has anomalies:
  - Insertion: cannot insert a teacher until we have a student taking his subject
  - Deletion: if we delete the last student of a teacher, we lose the subject he teaches
BCNF and Over-normalization (2)

- What is the problem? Schema overload. We are trying to capture two meanings:
  - 1) subject X can be taught by teacher Y
  - 2) student Z takes subject W from teacher V

It makes no sense to say we lose the subject he teaches when he does not have a student. Who is he teaching the subject to?

- Normalizing this schema to BCNF cannot preserve dependencies, so we better stay with the 3NF TEACH and another (BCNF) relation SUBJECT-Taught (teacher, subject) to capture the meaning of the real-world environment more effectively.