Announcements

- HW#3 is still underway...!
  - Today we will finish the irrelevant material (☹)
  - Recall the points-forgiving deadline (if needed)
- Today’s plan:
  - Relational DB design theory (IIIrd and final part)
  - Focus will be on decomposition theory/algorithms
  - And then: RELATIONAL ALGEBRA!
**Dependency Preserving Decomposition**

- Consider CSIDPOV. C is key, JP \( \rightarrow C \) and SD \( \rightarrow P \).
  - BCNF decomposition: Two tables V and SDP.
  - Problem: Checking JP \( \rightarrow C \) now requires a join!
- Dependency preserving decomposition (intuitive):
  - If R is decomposed into X, Y and Z, and we enforce the FDs that hold on Y and Z, then all FDs that were given to hold on R must also hold. *(Avoids Problem (3)).*
- **Projection of set of FDs F**: If R is decomposed into X, ... projection of F into X (denoted \( F_X \)) is the set of FDs \( U \rightarrow V \) in \( F^+ \) \((\text{closure of } F)\) where \( U, V \) are both in X.

---

**Dependency Preserving Decomp. (Cont’d.)**

- The decomposition of R into two tables X and Y is **dependency preserving** if \( (F_X \cup F_Y)^+ = F^+ \)
  - I.e., if we consider only dependencies in the closure \( F^+ \) that can be checked in X **without** considering Y, and in Y **without** considering X, they **imply** all dependencies in \( F^+ \)!
- Important to consider \( F^+ \), not F, in this definition:
  - **Ex**: EmpDeptMix(eid, email, ename, did, dname) with \( \text{eid} \rightarrow \text{email}, \text{email} \rightarrow \text{eid}, \text{eid} \rightarrow \text{ename}, \text{email} \rightarrow \text{did}, \text{did} \rightarrow \text{dname} \)
    - Emp(eid, email, ename) - \( \text{eid} \rightarrow \text{email}, \text{email} \rightarrow \text{eid}, \text{eid} \rightarrow \text{ename} \)
    - Dept(did, dname) - \( \text{did} \rightarrow \text{dname} \)
    - Work(eid, did) - \( \text{eid} \rightarrow \text{did} \) (instead of email \( \rightarrow \text{did} \))

- Dependency preserving does **not** imply lossless join:
  - **Ex**: ABC with A \( \rightarrow B \), if decomposed into AB and BC. *(Q: Key?)

---

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke
Decomposing a Design into BCNF

- Consider relation R with FDs F. If $X \rightarrow Y$ violates BCNF, decompose R into $R-Y$ and $XY$. ($R-Y$ has X still!)
  - Repeated application of this idea will yield a collection of relations that are BCNF, a lossless join decomposition, and guaranteed to terminate. (Didn’t say dependency preserving...)

- Ex: CSJDPQV with $C \rightarrow CSJDPQV$, $JP \rightarrow C$, $SD \rightarrow P$, and $J \rightarrow S$.
  - To deal with $SD \rightarrow P$, decompose into $SDP$, CSJDQV.
  - To deal with $J \rightarrow S$, decompose CSJDQV into JS and CJDQV.
- Note that in general, several dependencies may cause violations of BCNF. (And the order in which we deal with them can lead to different sets of relations!)

---

BCNF and Dependency Preservation

- In general, there simply may not be a dependency preserving decomposition into BCNF.
  - E.g., R(CSZ) with $CS \rightarrow Z$, $Z \rightarrow C$.
  - Can’t decompose preserving the first FD; not in BCNF...
- Consider again decomposing the relation CSJDPQV into relations SDP, JS and CJDQV:
  - Not dependency preserving (w.r.t. $JP \rightarrow C$, $SD \rightarrow P$, $J \rightarrow S$).
  - However, it is indeed a lossless join decomposition.
  - In this case, adding JPC to the collection of relations would give us a dependency preserving decomposition.
    - But: JPC tuples would be used only for checking FD! (Redundancy!)
Decomposition into 3NF

- Obviously, the lossless join decomposition algorithm for BCNF can also be used to obtain a lossless join decomp into 3NF (and might stop earlier).
- One idea to ensure dependency preservation:
  - If $X \rightarrow Y$ is not preserved, add relation $XY$.
  - Problem is that $XY$ may violate 3NF! E.g., consider the addition of $CJP$ to “preserve” $JP \rightarrow C$. What if we also have $J \rightarrow C$? (Which of course implies $JP \rightarrow C$.)
- The real fix: Instead of using the given set of FDs $F$ to guide the decomposition, use a minimal cover for $F$.

Minimal Cover for a Set of FDs

- **Minimal cover** $G$ for a set of FDs $F$ such that:
  - Closure of $G = $ closure of $F$, i.e., $G^+ = F^+$.
  - Right hand side (RHS) of each FD in $G$ is a single attribute.
  - If we change $G$ by deleting any FD or deleting attributes from the LHS of any FD in $G$, the closure would change.
- Intuitively, every FD in $G$ is needed, and also $G$ is “as small as possible” to have the same closure as $F$.
- E.g., $A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow GH$, $ACDF \rightarrow EG$ has the following minimal cover:
  - $A \rightarrow B$, $ACD \rightarrow E$, $EF \rightarrow G$ and $EF \rightarrow H$
- $M.C. \rightarrow$ lossless-join, dep. pres. 3NF decomp! (See book!)
**Obtaining that 3NF Decomposition**

- Compute the minimal cover \( G \), which is sometimes denoted \( F^- \) (see book for how)
- Search for dependencies in \( F^- \) with the same attribute set on the left hand side, \( \alpha \):
  - \( \alpha \rightarrow Y_1, \alpha \rightarrow Y_2, \ldots, \alpha \rightarrow Y_k \)
  - Construct one relation as \( (\alpha, Y_1, Y_2, \ldots Y_k) \)
  - Repeat this process for all the FDs’ \( \alpha \)’s
  - Construct a relation with any leftover attributes from \( R \)
  - If none of the relations contains a candidate key for the original relation \( R \), add one *more* relation containing the attributes of a candidate key for \( R \). (Q: Why?)

---

**On Refining ER Based Designs**

- 1st diagram translated:
  - Workers(\( S, N, L, D, S \))
  - Departments(\( D, M, B \))
    - Lots associated with workers.
  - *Suppose all workers in a dept are assigned the same lot: \( D \rightarrow L \) ....
- Redundancy; fixed by:
  - Workers2(\( S, N, D, S \))
  - WorkersLots(\( D, L \))
  - Departments(\( D, M, B \))
- Can further fine-tune this:
  - Workers2(\( S, N, D, S \))
  - Departments(\( D, M, B, L \))

---

*Notice: Lot wasn’t really a “Worker attribute”!*
Relational Design Theory Summary

- If a relation is in BCNF, it is free of redundancies that can be detected using FDs. Thus, trying to ensure that all relations are in BCNF is a good goal.
- If a relation is not in BCNF, we can try to decompose it into a collection of BCNF relations.
  - Are all FDs preserved? If a lossless-join, dependency-preserving decomposition into BCNF is not possible (or unsuitable for typical queries), consider 3NF instead.
  - Decompositions should be carried out while also keeping performance requirements in mind.

Relational Query Languages

- **Query languages**: Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages ≠ programming languages!
  - QLs not expected to be “Turing complete.”
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.
Formal Relational Query Languages

- Two mathematical Query Languages form the basis for “real” languages (e.g., SQL), and for their implementation:
  - **Relational Algebra**: More operational, very useful for representing execution plans.
  - **Relational Calculus**: Lets users describe what they want, rather than how to compute it. (Non-operational, declarative.)

Preliminaries

- A query is applied to *relation instances*, and the result of a query is also a relation instance.
  - *Schemas* of input relations for a query are fixed (but query will run regardless of instance!)
  - The *schema for the result* of a given query is also fixed! Determined by definition of query language constructs.

- Positional vs. named-field notation:
  - Positional notation easier for formal definitions, named-field notation more readable.
  - Both used in SQL (but try to avoid positional stuff!)
"Sailors" and "Reserves" relations for our examples.

We’ll use positional or named field notation, and assume that names of fields in query results are "inherited" from names of fields in query input relations (when possible).

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

- Selection ($\sigma$): Selects a subset of rows from relation.
- Projection ($\pi$): Omits unwanted columns from relation.
- Cross-product ($\times$): Allows us to combine two relations.
- Set-difference ($\neg$): Tuples in reln. 1, but not in reln. 2.
- Union ($\cup$): Tuples in reln. 1 and in reln. 2.

Additional operations:
- Intersection, join, division, renaming: Not essential, but (very!) useful. (I.e., don’t add expressive power, but…)

Since each operation returns a relation, operations can be composed! (Algebra is “closed”.)
**Projection**

- Removes attributes that are not in projection list.
- Schema of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Relational projection operator has to eliminate duplicates! (Why??)
  - Note: real systems typically don’t do duplicate elimination unless the user explicitly asks for it. (Q: Why not?)

```
π snamerating (S2)
```

```
sname  rating
yuppy  9
lubber  8
guppy  5
rusty  10
```

**Selection**

- Selects rows that satisfy a selection condition.
- No duplicates in result! (Why?)
- Schema of result identical to schema of its (only) input relation.
- Result relation can be the input for another relational algebra operation! (This is operator composition.)

```
σ rating>8 (S2)
```

```
sid   sname  rating  age
28   uppy  9   35.0
58    rusty 10   35.0
```

```
π snamerating (σ rating>8 (S2))
```

```
sname  rating
yuppy  9
rusty  10
```
Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be union-compatible:
  - Same number of fields.
  - “Corresponding” fields are of the same type.
- What is the schema of result?

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
</tbody>
</table>

$S_1 \cup S_2$

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

$S_1 \cap S_2$

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
</tbody>
</table>

Q: Any issues w/duplicates?

To be continued...