“Good” Relations

We’ve been talking about what makes a good database table. + Not a lot of NULLs + Minimal keys + Little, if any, data redundancy + No anomalies + No lost information (we haven’t talked about this yet, but we will)

But when database designs get larger (and more realistic), or when someone else (like the customer) gave you a design “just to get you started”, identifying good vs. poor tables becomes less intuitive.

Today we begin talking about a generalized process called “normalization” which is a process for evaluating and correcting table structures to minimize data redundancies, thereby reducing the chance of data anomalies.

This normalization process identifies which attributes (columns) should be in which relations (tables) based on the their functional dependencies.

The normalization process involves transforming the set of relations into a series of Normal Forms (thus “normalization”). + First normal form (1NF), 2NF, and 3NF are typically enough for real use. + 2NF is preferable to 1NF, and 3NF is preferable to 2NF and 1NF… + Academica studies higher forms, but its unlikely you will ever encounter them elsewhere. + There is another normal form between 3NF and 4NF called Boyce Codd NF, or BCNF which adds some desirable characteristics.

As is often the case in computer science, Normalization is not necessarily optimal for all situations. + In general, the higher the NF the more joins you will need for typical queries * This can use more run-time resources. + If performance is a key attribute of the final system, then careful denormalization may be necessary. * The only real downside to denormalization is higher data redundancy

As soon as you have ER models, you can begin.

- Goal is for all Relations to be well-formed
  - Each Relation represents a distinct thing
  - No Attribute will appear in more than one Relation
  - All attributes which are not part of the Primary Key are dependent on the Primary key
  - Every Relation is free of insertion, update, and deletion anomalies.
- Working one relation at a time,
Determining the dependencies of a relation
possibly breaking the relation up into multiple relations

Before we begin learning about the details of the normalization process, we need to learn how to reason about FD’s.

Functional Dependencies

note: This section derived from a superb lecture by Jennifer Widom of Stanford and the DBCB text.

The Normalization process depends upon the ideas of functional dependence. Let’s review:

Functional dependence: The value of one or more attributes determines the value of one or more other attributes.

- Determinant: Attribute whose value determines another
- Dependent: Attribute whose value is determined by the other attribute

FD’s (as they are often abbreviated) are a generalized notion of keys, but they have other uses in databases beyond design: + they enable more efficient data storage + compression schemes can be used as a result + they also enable some forms of query optimization

FD’s are based on knowledge of the real world of the entities you’re modeling.

All instances of a Relation must maintain the FD’s of that Relation … otherwise something is awry.

Formally: In lecture 03 we discussed this concept.

Write on board

Let $R$ be a relation schema, and $\alpha$ and $\beta$ are sets of its attributes.

The functional dependency

$\alpha \rightarrow \beta$

holds on $R$ if and only if for any legal relations $r(R)$, whenever any two tuples $t_1$ and $t_2$ of $r$ agree on the attributes $\alpha$, they also agree on the attributes $\beta$. That is:

Write on board

$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$

An attribute $\beta$ is fully functionally dependent on the attribute $\alpha$ if each value of $\alpha$ determines exactly one value of $\beta$

An attribute $\beta$ is fully functionally dependent on the composite attribute $\alpha$ if each value of $\alpha$ determines exactly one value of $\beta$

AND $\beta$ is not functionally dependent on any subset of the attributes in the composite key $\alpha$.

DRAW ON BOARD Let’s use our soccer tryouts example.

Player $(pID, pName, pAddr, pHSID, pHSName, pHSCity, pRank, pFri)$
Tryout$(pID, cName, pos, decision)$
Where \( p\text{Rank} \) in HS is a percentage, as in “top 90%”.

Suppose tryout priority is determined by \( p\text{Rank} \):

<table>
<thead>
<tr>
<th>( p\text{Rank} )</th>
<th>priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>90–100</td>
<td>1</td>
</tr>
<tr>
<td>70–89</td>
<td>2</td>
</tr>
<tr>
<td>60–69</td>
<td>3</td>
</tr>
</tbody>
</table>

Here we can see that any two tuples (rows) in \( \text{Player} \) that have the same \( p\text{Rank} \) will have the same priority.

Formally: if \( s \) and \( t \) are tuples of \( R \), then if

\[
s.p\text{Rank} = t.p\text{Rank} \Rightarrow s.p\text{Pri} = t.p\text{Pri}
\]

This is a functional dependency for \( R : p\text{Rank} \rightarrow p\text{Pri} \).

In fact, we can generalize this to multiple attributes on either side:

\( s_1, s_2, ..., s_n \rightarrow t_1, t_2, ..., t_n \)

Note that a functional dependency need not include all of the attributes in the Relation.

What other FD’s can we see in the Applications example?

\( p\text{ID} \rightarrow p\text{Name} \) which makes sense in the real world. Of course, as we said last time, using \( p\text{ID} \) as a primary key is a bad idea from a privacy perspective.

\( p\text{ID} \rightarrow p\text{Address} \) also makes sense, but only if you assume the student never moves!

Perhaps the same assumption for \( p\text{ID} \rightarrow p\text{HSID} \) if we assume only one HS

\( p\text{HSID} \rightarrow p\text{Name}, p\text{HSCity} \) is another FD that seems intuitive.

Two attributes on the left works too \( p\text{Name}, p\text{HSCity} \rightarrow p\text{HSID} \) since no two HS’s in the same city have the same name.
As we said before $pID \rightarrow pRank$ and $pRank \rightarrow pPri$, so can we also say that $pID \rightarrow pPri$?

The Apply relation is tougher to find FD's for.

If students can only apply for one major at any given college, then we have $pID, CName \rightarrow Major$.

SO, if we have a set of attributes $A$ in Relation $R$ that always determines all the other attributes $B$, then we can say that $A$ is a key for that Relation.

**Types of FD's**

- **Trivial**
  - $A \rightarrow B$ if $B \subseteq A$
  - Add $B$ as subset of $A$ columns

- **Nontrivial**
  - $A \rightarrow B$ if $B \not\subseteq A$
  - Part of $B$ is in $A$ and part is not
  - But note that if two tuples agree in their $A$ values then they will also agree in their $B$ values, both inside $A$ and out.

- **Completely nontrivial**
  - $A \rightarrow B$ where $A \cap B = \emptyset$
  - None of $B$ is on the $A$ side
  - But note again that if two tuples agree in their $A$ values then they will also agree in their $B$ values, even if distinct from $A$.
  - These are the most useful FD's

**Splitting Rule**

- right side only:

  $$\bar{A} \Rightarrow B$$

  which means

  $$\bar{A} \Rightarrow B_1, B_2, \ldots, B_n$$

  thus

  $$\bar{A} \Rightarrow B_1, \bar{A} \Rightarrow B_2, \ldots, \bar{A} \Rightarrow B_n$$

- However, left side splitting doesn't work: if we have
we can’t split it to imply

\[ pHSName, pHScity \Rightarrow pHSID \]

since there might be HS’s with the same name in different cities.

**Combining Rule**

\[ A \rightarrow B_1, A \rightarrow B_2, \ldots, A \rightarrow B_n \quad \text{then} \quad A \rightarrow B_{1, \ldots, n} \]

**Transitive Rule**

\[ \text{if} \quad A \rightarrow B \quad \text{and} \quad B \rightarrow C \quad \text{then} \quad A \rightarrow C \]

\[
\begin{array}{ccc}
A & B & C \\
\bar{a} & \bar{b} & \bar{c} \\
\bar{a} & \bar{b} & \bar{c} \\
\ldots & \ldots & \ldots \\
\end{array}
\]

\[ A^+ \text{: Closure of attributes} \]

Find the complete set of attributes \( B \) such that \( \bar{A} \rightarrow B \):
1. Start with
   \[ B = A_1, A_2, \ldots, A_n \]

2. IF
   \[ \bar{A} \rightarrow \bar{B} \]
   and all of
   \[ A \]
   is in the set

3. THEN
   add
   \[ \bar{B} \]
   to the set.

4. REPEAT until no further changes

So if we started with \( B \) above, we might find that \( A_5 \rightarrow C \) and then \( C \rightarrow DE \) etc., so the final answer might be

\[ \bar{A}^+ = B = A_1, A_2, \ldots, A_n, C, D, E \]

So let's try it with our soccer relations **ON THE BOARD WITH STUDENTS**

| Player (pID, pName, pAddr, pHSID, pHSName, pHSCity, pRank, pPri) |
| Tryout(pID, cName, pos, decision) |

We have

- \( pID \rightarrow pName, pAddr, pRank \)
- \( pRank \rightarrow pPri \)
- \( pHSID \rightarrow pHSName, pHSCity \)

We want to find the closure of the attribute set \( \{pID, pHSID\}^+ \). That is, we want to find all of the attributes that are functionally determined by those in this initial set.

- \( \{pID, pHSID\} \)
- \( \{pID, pHSID, pName, pAddr, pRank\} \)
- \( \{pID, pHSID, pName, pAddr, pRank, pPri\} \)
- \( \{pID, pHSID, pName, pAddr, pRank, pPri, pHSname, pHScity\} \)

and we're done now because this is all of the attributes in the relation!

Thus, if all the attributes in a relation can be functionally determined by the attributes \( \{pID, pHSID\} \), that is the definition of a key for that relation!
So we can determine whether some attribute set $\bar{A}$ is a key by computing $\bar{A}^+$ and if that produces all attributes in the relation then the original set is a key.

We can also find all the possible keys if we’re given the relation and a set of FD’s by considering every subset of the attributes and then running the algorithm to see if that subset functionally determines all the attributes!

**Normalization**

An process for minimizing data redundancies and data anomalies

Involves redistributing attributes between tables, creation of new tables and relations.

Lowest is 1st NF, and we will look at 2, 3, and one more called BCNF that’s just a little bit better than 3NF.

Higher is better, in general, but denormalization may be appropriate for performance / efficiency reasons.

Do Normalization as soon as you have an accepted ERD. *Changes resulting from Normalization need to be reflected back in your ERD for documentation!*

Restating our goals: + All Relations will be well-formed + Each Relation represents a distinct thing + No Attribute will appear in more than one Relation + All attributes which are not part of the Primary Key are dependent on the Primary key + Every Relation is free of insertion, update, and deletion anomalies.

**NF Summary**

Functional Dependency review **SLIDE 2**

**Partial Functional Dependencies**

Suppose you have $\alpha \rightarrow \beta$ and $\alpha$ is actually a compound of two (or more) attributes, $\alpha_1$ and $\alpha_2$.

If $\beta$ is actually functionally determined by only one of them, say $\alpha_1$, then this is a *partial dependency*.

**Transitive dependencies**

If neither $\alpha$ nor $\beta$ are part of the primary key, then $\alpha \rightarrow \beta$ is a *Transitive dependency*.

**1NF**

Steps: 1. Eliminate the repeating groups 2. Identify the PK 3. Identify all FD's 4. verify all attributes depend on PK

In other words (Jukic):

A table is in 1NF if each row is unique and no column in any row contains multiple values from the column’s domain.

Generally all the tables we’ve used are in 1NF … but customer’s designs may not be. For example here’s our relation.

**Show the RPT_FORMAT table in MYSQLWorkbench** Explain Repeating groups
Dependency diagrams

Show all the dependencies within a Relation * easier to see how they all fit together, helops keep track of them

** Draw ours on the board w/o arrows **

Conversion to 2NF

2NF means: * 1NF and no partial dependencies

Steps 1. Make new tables to eliminate partial dependencies 2. Reassign corresponding dependent attributes

Use our diagram to identify the PD’s and then make new relations by removing the dependent attributes into new relations with
their determinant as PK.

** Update ours on the board w/o arrows **
Conversion to 3NF

3NF means: * 2NF and no transitive dependencies

Steps: 1. Make new relations to eliminate transitive dependencies 2. Reassign corresponding dependent attributes

Use our diagram to identify the transitive dependency and make new relations by removing the dependent attributes into new relations with their determinant as PK.

** Update ours on the board w/o arrows **
3NF Dependency Diagram

And that’s 3rd NF!

Show completed database slides 3, 4, 5, & 6

Other improvements

- Review PK definitions
- Check naming conventions
- Review Attributes for atomicity (cannot/shouldn’t be further subdivided)
- Missing attributes and relations?
- Evaluate using derived attributes or storing them?
- Ensure proper granularity of PK’s
  - Can you access the data at the desired level of detail?
  - Address as a whole, or do you need access to zip?
- Be careful about history, backwards compatibility with old tables and other applications
  - may need “bridge relations”.

STOP HERE FOR QUIZ Q&A REVIEW

BCNF

Boyce-Codd Normal Form is a special case of 3NF

It goes a little beyond 3NF because it requires that all redundancy based on FD’s is removed.

- A Relation is in Boyce-Codd Normal Form (BCNF) when it is in 3NF and the left side of every non-trivial FD (determinant) of the Relation is a Superkey.
  - Recall that a Superkey is an attribute or combination of attributes that uniquely identifies each row in a table.
For example, if the table is in 3NF and it contains a nonprime attribute that determines a prime attribute, the BCNF requirements are not met.

- **Formal definition:**
  - We say a relation $R$ is in BCNF if whenever $X \rightarrow Y$ is a nontrivial FD that holds in $R$, then $X$ is a superkey.
  - Remember: nontrivial means $Y$ is not contained in $X$.

- **Same as 3NF when there is only one candidate key**

- **BCNF can only be violated if there is more than one candidate key**

**BCNF and non-BCNF Examples**

- **Example from Garcia-Molina:**
  - FD's: $name \rightarrow addr, favBeer$, $beersLiked \rightarrow brewer$.
  - In each FD, the left side is not a superkey.
  - This, either FD shows the Relation is not in BCNF.

- **Another example from G-M:**
  - FD's: $name \rightarrow manf, manf \rightarrow manfAddr$.
  - $name \rightarrow manf$ does not violate BCNF, but $manf \rightarrow manfAddr$ does.

- **Converting to BCNF**
  - Start with the offending FD: $X \rightarrow Y$ for the relation $R$.
  - Compute $X^+$ (which isn’t all attributes, since that would make $X$ a superkey).
- Replace \( R \) with new relations with these schemas:
  1. \( R_1 = X^+ \) and
  2. \( R_2 = R - (X^+ - X) \)

- Continuing the G-M example:
  - \( \text{Drinkers}(\text{name}, \text{addr}, \text{beersLiked} \text{brewer}, \text{favBeer}) \)
  - Choose BCNF violation FD: \( \text{name} \rightarrow \text{addr} \) (splitting rule)
  - Closure of \( \{\text{name}\}^+ = \{\text{name}, \text{addr}, \text{favBeer}\} \)
  - Decomposed relations:
    1. \( \text{Drinkers1}(\text{name}, \text{addr}, \text{favBeer}) \)
    2. \( \text{Drinkers2}(\text{name}, \text{beersLiked}, \text{manf}) \)
  - Are we done? No, we do it again until no BCNF violations
    1. For \( \text{Drinkers1} \) the relevant FDs are \( \text{name} \rightarrow \text{addr} \) and \( \text{name} \rightarrow \text{favBeer} \)
       Thus, \( \{\text{name}\} \) is the only key and \( \text{Drinkers1} \) is in BCNF.
    2. For \( \text{Drinkers2} \) the only FD is \( \text{beersLiked} \rightarrow \text{manf} \)
       and the only key is \( \{\text{name}, \text{beersLiked}\} \)
       which is a violation of BCNF, so we do it again
  3. \( \{\text{beersLiked}\}^+ = \{\text{beersLiked}, \text{manf}\} \) so we decompose \( \text{Drinkers2} \) into:
    - \( \text{Drinkers3}(\text{beersLiked}, \text{manf}) \)
    - \( \text{Drinkers4}(\text{name}, \text{beersLiked}) \)
  - The final result of the decomposition of \( \text{Drinkers} \) is:
    1. \( \text{Drinkers1}(\text{name}, \text{addr}, \text{favBeer}) \)
    2. \( \text{Drinkers3}(\text{beersLiked}, \text{manf}) \)
    3. \( \text{Drinkers4}(\text{name}, \text{beersLiked}) \)
  - Where
    1. \( \text{Drinkers1} \) tells us about drinkers,
    2. \( \text{Drinkers3} \) tells us about beers, and
    3. \( \text{Drinkers4} \) tells us the relationship between drinkers and the beers they like.

- Another example (from Silberschatz)
  - \( \text{instr_dept}(\text{ID}, \text{name}, \text{salary}, \text{dept_name}, \text{building}, \text{budget}) \)
  - with FD: \( \text{dept_name} \rightarrow \text{building}, \text{budget} \) but \( \text{dept_name} \) is not a superkey of \( \text{instr_dept} \)
  - \( \{\text{dept\_name}\}^+ = \{\text{dept\_name}, \text{building}, \text{budget}\} \)
    so we decompose into
    - \( \text{instr\_dept1} = (\text{dept_name}, \text{building}, \text{budget}) \)
    - \( \text{instr\_dept2} = (\text{ID}, \text{name}, \text{salary}, \text{dept_name}) \)

**So what does this 3NF and BCNF buy us?**

There are two important properties of any decomposition: 1. Lossless Join: it should be possible to project the original relations onto the decomposed schema, and then reconstruct the original. 2. Dependency Preservation: it should be possible to check in the projected relations whether all the given FD’s are satisfied.

Not all decompositions are good. Here’s an example from Silberschatz:

* Decompose \( \text{Employee}(\text{ID}, \text{name}, \text{street}, \text{city}, \text{salary}) \) into
  \( \text{employee1}(\text{ID}, \text{name}) \) and
  \( \text{employee2}(\text{name}, \text{street}, \text{city}, \text{zip}) \)
* Can we reconstruct the original relation?
A decomposition that is not lossless is called *lossy*.

The 3NF conversion process ensures no losses.

**Denormalization**

Our design goals have been

* Creation of normalized relations
* Processing requirements and speed

Motivations were due to defects in unnormalized tables

* Data updates are less efficient because tables are larger
* Indexing is more cumbersome
* No simple strategies for creating virtual tables known as views (we'll see this later)
However, the number of database tables expands when tables are decomposed to conform to increasing normalization requirements.

This results in having to join a larger number of tables to use the DB
* Takes additional input/output (I/O) operations and processing logic
* Reduces system speed

Common examples of denormalization:
- derivable data: storing both $MILES\_FLOWN$ and $AWARD\_LEVEL$ in a frequent fyer DB since $AWARD\_LEVEL$ is derivable from $MILES\_FLOWN$. + avoids extra joins, enables auditing since $AWARD\_LEVEL$ can be verified by looking at $MILES\_FLOWN$.
- Preaggregated data: Storing GPA when it could be calculated from the $ENROLLED$ and $COURSE$ tables.
+ also avoids extra joins and the GPA is recomputed everytime a grade is entered or updated.

**Review of Data Modeling**

*Use Coronel's CH06 slides 41–44*

1. Lecture notes based on texts by Coronel, Widom, Ullman, Jukic, and Silberschatz. 