The Relational Data Model: Formalisms on Design

1 Overview

- We've talked about database design from entity-relationship (E-R) and UML perspectives, and we've also talked about how to map designs in these paradigms into the relational data model.
- While mapping E-R or UML to the relational data model has a mechanical, recipe element to it, you may have noticed that:
  1. There is still more than one way to translate E-R and UML into possibly equivalent relations, and
  2. A lot of database design decisions are still dependent on prior experience and, in some cases, good instinct or aptitude for accurately translating the real world into the database world.
- Fortunately, at the level of the relational model, there are other, more formal ways to assess the quality of a database's design. That's what this handout is about.
- And yes, we can't say this enough: we owe this power and certainty to the mathematical foundation behind the relational data model. In this, as in many other areas, it's the mathematics that truly powers the technology.

2 What We Want From a Relational Design

Good relational database designs share very similar properties to good software designs:

- *Minimal to zero redundancy.* Information is stored in one place, without autonomous copies. Thus, one change affects one clear section of the database, and no changes need to be applied more than once.

- *Let the system work for you.* Relational databases implement clear, unambiguous rules in areas such as the behavior and interaction of keys (primary, foreign), the acceptability of null values, and the exact behavior of relational operations. A good design takes full advantage of these features, allowing the computer to do what it does best, with minimal additional maintenance.

- *Accurate reflection of the domain.* In the end, a database's usefulness is determined by how closely it matches the needs and rules of its application domain. Relationship cardinalities must reflect their real world counterparts; impossible situations in the real world must be similarly impossible (or at least sufficiently improbable) in the database incarnation; the most frequently-performed activities in the real world should be the most natural and efficient ones in the database.

The relational model allows us to express these criteria in a very formal, precise manner — with it, we can actually *prove* certain properties of our design, and also algorithmically discover any flaws in it. So, without further ado...

3 Atomic Domains and First Normal Form

- Quick rehash: relations are sets of attributes, and each attribute is supposed to belong to a *domain*. We said this before but didn't pay attention to it much then: domains are supposed to be *atomic* — that is, their members may not be broken down into subcomponents.

- This is the first distinction between E-R and UML designs and a relational design — *no non-atomic attributes*. While E-R and UML both allow *multivalued* attributes (e.g., list of phone numbers) and *composite* attributes (e.g., address consisting of street, city, state, and zip), the relational model does not.

- So, the first step in relational design: remove all multivalued and composite attributes. Or, *make all attributes atomic*.

- A relational schema R is in *first normal form* (1NF) if the domains of all attributes in R are atomic.

- In other words, a relational schema R is in first normal form if it conforms to the formal definition of a relational schema. Seems like an obvious no-brainer, but it is an essential first building block.

4 Functional Dependencies

We have formally defined *keys* in a relational database; keys determine the uniqueness of certain attribute values, whether or not they may appear multiple times in a relation (super-, candidate, and primary keys), and whether or not they must correspond to tuples in other relations (foreign keys). We now introduce and define a new but similar concept — that of a *functional dependency*.

4.1 Definitions

- A relation is *legal* if its content satisfies the constraints imposed on the relational schema.
- Given a relational schema R, with attribute sets $\alpha \subseteq R$ and $\beta \subseteq R$. The *functional dependency* $\alpha \rightarrow \beta$ holds on R if, for any legal relation $r(R)$, $\forall\ t_1, t_2 \in r,\ t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$.
- With functional dependencies, we can rephrase our definition of a superkey by saying that a set of attributes $K \subseteq R$ is a superkey of R if $K \rightarrow R$.
  - Replacing K for α and R for β in the functional dependency definition above, we see that $t_1[K] = t_2[K] \Rightarrow t_1[R] = t_2[R]$.
  - Since $t_1[R] = t_2[R]$ is the same as saying that $t_1 = t_2$, then we can see how the functional dependency-based definition of keys matches our original tuple equality-based definition.
- Given a set of functional dependencies F, a relation $r(R)$ *satisfies* F if it is legal under F.
- Conversely, we say that F *holds* on R if we are constraining relations on schema R only to those relations that satisfy F.
- Functional dependencies do not necessarily commute — in other words, $\alpha \rightarrow \beta$ does *not* imply that $\beta \rightarrow \alpha$.

4.2 Trivial Functional Dependencies

A *trivial* functional dependency is one that is satisfied by *all* relations. Some examples:

- $A \rightarrow A$ is trivial (and obvious, we hope!).
- A tad less obvious, but still straightforward, is $AB \rightarrow A$. Easy to prove (right?), and rewritable as $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$.

4.3 Inferring Functional Dependencies

- We can infer new functional dependencies from a basic, initial set of functional dependencies F. For example:
  - For a relational schema whose attributes can be partitioned into subsets A, B, and C: if $A \rightarrow B$ and $B \rightarrow C$ hold on R, then $A \rightarrow C$ holds on R as well. Again, take a minute or so to prove this in your head.
- The *closure* of F, written as F^+, is the set of all functional dependencies that can be inferred from the original set of functional dependencies F. Needless to say, $F^+ \supseteq F$.
- The "sledgehammer way" for determining F^+ is by exhaustively reasoning about all of the functional dependencies in F and seeing what other functional dependencies may be inferred from them purely based on the definition of a functional dependency.
- But, it should be quite clear right away that we don't want to do that.
- Fortunately, we have *Armstrong's axioms* — higher-level rules that allow us to build F^+ more quickly:

  Reflexivity rule. If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ — this is the same "trivial" rule as above.

  Augmentation rule. If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.

  Transitivity rule. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds. Again, same as above.
- Armstrong's axioms are *sound* because they don't generate incorrect functional dependencies.
- They are also *complete*, because following them exhaustively is guaranteed to generate F^+.
- But wait! There's more — we can now build further upon Armstrong's axioms to produce additional rules. Again, these rules all ultimately derive from the definition of a functional dependency, but they integrate the most common inference types into proven higher-level rules:

  Union rule. If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.

  Decomposition rule. If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.

  Pseudotransitivity rule. If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.
- With these higher-order rules, we can define an algorithm that will build F^+ for us:

```
Fplus = F
repeat
    for each functional dependency f in Fplus {
        apply reflexivity and augmentation rules on f
        add the resulting functional dependencies to Fplus
    }

    for each pair of functional dependencies f1 and f2 in Fplus {
        if f1 and f2 can be combined using transitivity {
            add the transitive-derived functional dependency to Fplus
        }
    }
until Fplus does not change any further
```

- This algorithm is guaranteed to terminate because the set of all possible functional dependencies in a relation is finite: $2 \times 2^n = 2^{n+1}$ possible functional dependencies.

4.4 Functional Dependencies and the Real World

- It is important to note that, while we can define functional dependencies quite arbitrarily or by inspection of specific instances of a relation, their real usefulness comes from whether they *accurately reflect the characteristics of the real world*.

- Thus, "incidental" functional dependencies — functional dependencies that only *happen* to hold for a relation r purely by virtue of the tuples that are in r at a given moment — are not important to us.

- What we care about are the functional dependencies that reflect how the real-world equivalents of our data behave. For example, given any nation, you immediately know its capital city. However, while there aren't currently any nations whose capital cities have the same name, if you think about it there isn't really any hard and fast rule that says that two nations can't have capital cities of the same name. Thus, we can't really say that capital cities functionally determine the nation, even if a database of nations seems to bear that out for now. But there is the operative phrase — "for now."

- Functional dependencies that hold "for now" aren't of use to us — we care about the functional dependencies must reflect the rules, logic, and common sense of the real world.

5 Boyce-Codd Normal Form (BCNF)

Back to normal forms — after all, 1NF wouldn't be called "first" if there weren't other normal forms lurking around.

- For a relation schema R and a set of functional dependencies F, R is in *Boyce-Codd normal form* (BCNF) with respect to F if, $\forall\ \alpha \rightarrow \beta \in F^+,\ \alpha \subseteq R\ \wedge\ \beta \subseteq R$, at least one of the following is true:
  1. $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  2. α is a superkey for R
- A relational database schema is in BCNF if every relation schema in that relational database is in BCNF.
- BCNF allows us to discover and eliminate data redundancy based on functional dependencies. Here's how:
  1. Take any relational schema R that is not in BCNF with respect to some set of functional dependencies F.
  2. Since R is not in BCNF, then there is at least one nontrivial functional dependency $\alpha \rightarrow \beta$ for which α is not a superkey of R.
  3. Given these attribute subsets *alpha* and β, we replace R with two new relations:
     (a) $R' = \alpha \cup \beta$
     (b) $R'' = R - (\beta - \alpha)$
  4. Finally, check if R' and R'' are in BCNF, if not, repeat the algorithm for the non-BCNF relations.

Note that, yet again, the crucial element here is that we have a clear, complete understanding of our application domain — that is what determines the functional dependencies F that must hold for the relations in the database.

6 Third Normal Form (3NF)

One issue with BCNF is that it can be thought of as "too strong" — there are some real-world constraints that, in combination, cannot be expressed in BCNF relations (see the textbook for an example). The technical term for this is *dependency preservation*. We note that some BCNF relational database schemas are not dependency preserving, and so define a "weaker" normal form that preserves dependencies: *third normal form* (3NF).

- A relational schema R is in *third normal form* (3NF) with respect to a set of functional dependencies F if, $\forall\ \alpha \rightarrow \beta \in F^+,\ \alpha \subseteq R\ \wedge\ \beta \subseteq R$, at least one of the following is true:

  1. $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  2. α is a superkey for R
  3. Each attribute $A \in \beta - \alpha$ is contained in some candidate key for R

- Note how the term "weaker" applies to 3NF in comparison to BCNF because it allows one additional condition that allows a relation to satisfy 3NF.

- Again due to the purely additional condition, it should be clear that any schema that satisfies BCNF also satisfies 3NF.

7 Putting Functional Dependencies in Perspective

- It should be noted that the three primary concepts we've discussed so far — functional dependencies, BCNF, and 3NF — are all means to an end. We don't set out to create a BCNF schema or a 3NF schema; we set out to create a *good* schema, with minimal redundancy but which accurately reflects the real-world rules that govern the data.

- What functional dependencies, BCNF, and 3NF do is provide is with reusable, repeatable tools that allow us to test our design *systematically*. Generally speaking, you are more likely to have a good relational database design if this design conforms to either BCNF or 3NF, and that is why they're around.