CSE 344: Section 10 Notes

Overview

This guide will give an overview of topics that we did not have time to cover in lecture this quarter but we believe is important to the topic of database management and design. Topics in this document are functional dependencies (FD), the closure algorithm, and Boyce-Codd Normal Form (BCNF). Please do note that these topics are all fair game for the exam, so be sure to understand the high-level concepts and ask on Piazza if you have any questions. Also keep in mind that this guide is not meant to be comprehensive and is only meant to provide tangible understandings of the aforementioned concepts.

Functional Dependencies

What are Functional Dependencies?

A functional dependency is a relationship that exists when **a set of attributes uniquely identifies another set of attributes**. Specifically, if two tuples agree on the attributes a1,a2,…,an, then they must also agree on the attributes b1,b2,…,bm. We can write this more formally as a1,a2,…,an → b1,b2,…,bm.

So, why are functional dependencies important? We'll see that functional dependencies are crucial for relational schema design. They help in eliminating redundancy, which in turn helps in reducing **anomalies** such as **redundancy, slowed updated, and zealous deletion**. They are used to create relations in BCNF form, discussed later.

Some terminology: We can either say a functional dependency **holds** or it does not hold, depending on whether or not it is valid on a specific instance of a relation R. If we can be sure that every instance of R will be one in which the FD holds, we say that R satisfies the FD.

Example:
Say we have the following relation, with entry ID (EID) declared as a primary key:

<u>EID</u>	Artist	Album	Rating
1	Taylor Swift	Speak Now	10
2	Kanye West	Heartless	2
3	Kanye West	The Life of Pablo	2
4	Charlie Puth	Nine Track Mind	9
5	BTS	Wings	7

Based on the above relation, what valid functional dependencies can we say that we have?

There are actually many possible answers. One valid FD we have is EID → Artist. We can see from the above relation that each matching EID corresponds to the same Artist and Album.

Technically, since EID is a primary key, we already know that there will be no two matching EIDs in the relation. Similarly, we have the valid FDs EID → Album, EID → Rating, EID → Artist, Album, Rating, and so on.

A more interesting example would be Artist → Rating. We can look at the table and see that each artist always receives the same rating for their albums. For example, Taylor's albums always receive a 10, Kanye's albums always receive a 2, Charlie's albums always receive a 9, and Korean-pop sensation BTS' albums always receive a 7. Thus, this functional dependency (Artist → Rating) holds.

However, say that another row was added to the relation, consisting of 6 | Taylor Swift | Reputation | 9. If this was the case, the above FD would no longer hold, since one of Taylor's albums now has a rating of 9, and one has a rating of 10 – the ratings don't match!

What would be an example of an FD that does not hold from the get-go? Let's take a look at Artist → Album. We can see from the above relation that Kanye West made the albums Heartless and The Life of Pablo. Thus, we have Kanye West → Heartless, and Kanye West → The Life of Pablo. Remember our definition of valid functional dependencies from before? A given Artist must always correspond to the same Album. However, here we have Kanye corresponding to both Heartless and The Life of Pablo in two separate instances. Thus, we say that this functional dependency does not hold.

As shown above, as more entries are added to the table, it is possible that some functional dependencies that previously held no longer hold. Likewise, if entries are deleted from the table, some new valid FDs may pop up. For instance, if the row containing The Life of Pablo was deleted, the FD Artist → Album would now hold. That said, when we are examining what Fds we want to enforce though the lifespan of the data.

Closure Algorithm

Given certain functional dependencies, how can we find everything that a set of attributes determine/are a key for? The answer is the closure algorithm. **The closure algorithm is used for finding all the functional dependencies b1,b2,…,bm for a set of attributes a1,a2,…,an.**

Algorithm:

```
X = {A1, …, An}.
Repeat until X doesn't change do:
     if (B1, …, Bn → C is a FD) and (B1, …, Bn are all in X)
     then add C to X
```

What this is doing is having a **fixed point** and then **tracing the FDs** until there is nothing else to add. This way given a set of attributes we can find all attributes that are in its closure.

The formal notation for specifying a closure is to write $\{a1,a2,\ldots,an\}^+ = \{a1,a2,\ldots,an, b1,b2,\ldots,bm\}$

Keys

When we observe closures, we sometimes get the situation where a closure of a set of attributes is all attributes in a relation. When this happens we can say that the starting set of attributes is called a **superkey**. Of the set of superkeys that are possible for a relation, the superkeys that are the smallest in terms of cardinality are called **minimal keys** or just **keys**. Keys are particularly interesting for database schema design as they indicate what should be marked as primary keys.

Example:
We have the same data as before. Let's say that we have the FDs EID → Album and Album → Artist, Rating.

EID	Artist	Album	Rating
1	Taylor Swift	Speak Now	10
2	Kanye West	Heartless	2
3	Kanye West	The Life of Pablo	2
4	Charlie Puth	Nine Track Mind	9
5	BTS	Wings	7

From these given FDs, can we determine what the keys are in the data?

One trivial example of a superkey we can get is {EID, Artist, Album, Rating}. But of course this is a superkey! All the attributes in a tuple, will always determine all attributes in the tuple. So generally this is uninteresting.

A more interesting observation is the closure of {EID}. By tracing our FDs, we know that EID determines an Album and an album will determine an artist and rating. Thus, the closure of {EID} is {EID, Artist, Album, Rating} (all attributes in the relation!). Because, we cannot get a smaller size set that is a superkey, {EID} is a minimal key of the relation.

Boyce-Codd Normal Form (BCNF)

Anomalies

When we first gather attributes in a relation to create in a database instance it is possible to not do anything about it and put all your information into a single table. For some data, this is a bad idea due to anomalies. Anomalies are typically observed in three forms being **redundancy**, **slowed update performance**, and **zealous deletion**. The typical cause of this is due to there being FDs in a relation that has the starting attributes not as a key.

Normal Forms and BCNF

Normal forms are a concept in relational models that attempt to promote consistency and ease of use through the partitioning of data into different tables.. There are a few different partitioning methods, from the very simple 1st normal form (the only constraint that tables are flat) to the Boyce-Codd Normal Form (BCNF) which is what will be discussed here.

Definition:
A relation is in BCNF if for all X→B (that are non-trivial like A → A) X must be a superkey
Equivalently...
A relation is in BCNF if for all for all X, {X}+ = X or {X}+ = {all attributes in the relation}

One can see why BCNF would be a logical choice for how to partition our data as it guarantees that there are no anomalies as discussed before.

There are systematic algorithms to convert non-normalized data to normal forms like BCNF given any functional dependencies that you define. For BCNF in particular, lossless decomposition is the methodology for partitioning, and the chase algorithm is used for verification, however you are not expected to know how these algorithms work for the final.

Takeaway

What we hope that you take away from this small introduction to design theory for databases is that there are concrete ways to make well-behaved schema. In any future applications that you may make which use relational databases, if you are able to correctly define your functional dependencies (properties of your data), you can normalize your model to promote consistency and ease of use.

Material adapted from CSE 344 AU 2017 slides

Written by:
Allison Chou (aachou@cs)
Jonathan Leang (jleang@cs)