Lecture 13B (April 11)

CS 410 Spring 18

Logistics

- Saw: indexes!
- Next: full-text indexing
- Now: specialization, generalization, in EER and relational modeling

Textbook: 4, 9.2-9.3

Full-Text Indexing

MySQL docs: https://dev.mysql.com/doc/refman/5.7/en/fulltext-search.html

We saw three index types:

- B-trees, optimizing joins, lookups, </>, and LIKE ‘Foo%’
- Hashtables, optimizing lookups
- R-trees, for location-based lookup

What if we want to search text? E.g. find all articles with the word ‘trumpet’.

Many DBMSes, including MySQL, let us implement a simple version of that! In MySQL, it is done with the FULLTEXT index type.

```
CREATE FULLTEXT INDEX article_search_idx
    ON article (title, abstract);
```

We can then search our articles:

```
SELECT article_id, title
FROM article
WHERE MATCH (title, abstract) AGAINST ('trumpet');
```

If we want to search titles only, we need a second index:

```
CREATE FULLTEXT INDEX article_title_search ON article (title);
SELECT article_id, title
FROM article
WHERE MATCH (title) AGAINST ('trumpet');
```

MATCH AGAINST is just an SQL condition, we can combine it with joins and other things; let's find out where these things were published.

```
SELECT article_id, title, proc_id, proc_title
FROM article JOIN proceedings USING (proc_id)
WHERE MATCH (title, abstract) AGAINST ('trumpet');
```

Let's make this work in our Java program!

Look @ Java code ('search' method)

- The search query is a placeholder, just like in our other queries
- We bind its value with setString
- Then we run!

Run shell, do 'search trumpet'

How does this work? Take the IR class for details :). The basic idea: it indexes individual words, and makes them point back to the rows that store them.

Specialization and Subclasses

Problem from earlier: bike shop

- Sales staff, mechanics, managers
- But aren't they all employees?

Java lets us do this with inheritance or subclassing! Can we do that in the database?

Side note: attempting to encode ontology into OOP inheritance is a recipe for pain.

Yes. Enhanced Entity-Relationship Modeling (EER) lets us capture this through *subclasses*.

The process by which we derive subclasses is called *specialization* or *generalization*.

Start with specialization.

Example: bike shop employees. **Employee** has basic attributes (ID, name, address, SSN).

It has three *subclasses*: **salesperson**, **mechanic**, **manager**. Employee is called the *superclass*.

These subclasses are *overlapping*: someone can do both sales and mechanics and management.

Draw this out.

Ok, now we have:

- A superclass, Employee
- An *overlapping specialization* consisting of three subclasses: *salesperson*, *mechanic*, and *manager*.

We can also have *totality*: a superclass totally participates in a specialization if every member of the superclass must be a member of one or more subclasses. Ours is *non-total*: there might be other employees (e.g. janitorial staff, marketing people) who don't fit into one of our classes.

Specific Attributes and Relationships

So far, we've made our model more complicated. What does this gain us?

Individual subclasses can have their own attributes! These are called **specific attributes**.

- Salesperson: commission
- Mechanic: cert_date, the date of their last maintenance certification

Subclasses **do not** have primary keys **-** they share primary keys with the superclass.

Subclasses can also participate in relationships on their own; these are called **specific relationships**. To see that, let's re-introduce our invoices and work orders:

- Invoices have IDs, dates, and line items.
- Each invoice is sold by a salesperson
- Work orders are associated with invoices
- Work orders are completed by mechanics

Now, *manager* isn't doing anything. But we can add a relationship that might look a little weird: *employees* are managed by *managers* (a manager can manage multiple employees).

Why shouldn't this relationship be total? (i.e. every employee has a manager)

Multiple Participation

One big difference between EER and Java classes/objects: in Java, an object is an instance of one class (along with all its superclasses).

In EER, an entity can be of multiple types/subclasses at the same time. One employee can be a salesperson and a mechanic, and EER is fine with this since our specialization is *overlapping*. To see how this works, let's get to:

Translating to the Relational Model

The textbook outlines 4 different options for translating EER to the relational model. We'll use the Multiple Relations option.

Each entity type *and subtype* becomes a relation

- Employee, the supertype, just becomes a relation as if it does not have any subtypes.
- Each subclass becomes a relation with:
    - Primary key - superclass's primary key
    - Primary key is *also* a foreign key referencing the superclass relation
    - Specific attributes & foreign keys for specific relationships

We need to account for this in querying!

Translate our model.

Generalization in the HCIBIB Database

We just saw *specialization* - taking an entity type and making more specific versions of it.

Let's go the other way. Consider our HCIBIB database - we have papers published in proceedings.

But some papers are published in journal issues. We want to record those too!

We have two things:

- *Proceedings* in a *conference series*
- *Issue* in a *journal*

Proceedings have what we already have: date(s), series, title, key. The conference series has a key.

Issues have a volume number and issue number, a date, and are in a journal; they also have a key. The journal has a title and a key.

In specialization, we started with general and got more specific. Here, we have two specific things, and we want to *generalize* into an entity type that describes both: a **publication**.

What do they have in common?

- A publication has an ID, a date, and a key
- A proceedings has start/end dates, a title, and is in a series

- An issue has volume & number, and is in a journal

This lets us build a set of subclasses!

Two interesting differences now, though:

- The subclasses are *disjoint*: a publication is not both a proceedings and an issue. This is indicated with a ‘d’ in the specialization circle
- The specialization is *total*: every publication is a proceedings or an issue

Draw it out

Translate to relations

Where are our foreign key relationships?

We want to rename ‘proc_id’ to ‘pub_id’ now too.

Look at updated schema

Now, publication title is a little weird - for a proceedings, it is the proceedings title. For an issue, it is derived from the journal title, volume, and number. We’ll model this with a title attribute on Proceedings, and a title derived attribute on Issue; however, if we want to get titles for arbitrary publications (useful for queries!) it will be helpful to make a view:

```
CREATE VIEW pub_title
AS SELECT pub_id, COALESCE(proc_title, CONCAT_WS(', ', journal_title, iss_volume,
iss_number))
FROM publication
LEFT JOIN proceedings USING (pub_id)
LEFT JOIN issue USING (pub_id)
LEFT JOIN journal USING (journal_id);
```

Then we can query:

```
SELECT article_id, pub_id, title, pub_title
FROM article JOIN pub_title USING (pub_id)
WHERE MATCH (title, abstract) AGAINST ('trumpet');
```

Multiple Specializations

EER also supports *multiple specializations* for the same entity type! Weird, but cool!

Let’s return to the bike shop. There are two ways employees can be paid: hourly or salaried. So we can add another specialization for employee:

- Hourly employees have a pay rate
- Salaried employees have a salary

This specialization is *disjoint*, indicated with a ‘d’ instead of ‘o’ in the specialization circle.

Also, this one will be total! Every employee is either hourly or salaried.

Translate this model to relational too.

Now, employee has two specializations: one with three overlapping subclasses, and one with two disjoint subclasses.

Translation without Separate Relations

We can also do the translation by adding more attributes to the Employee relation: salary and pay_rate fields, exactly one of which is NULL.