CS 325 - Reading Packet: "Introduction to Normalization"

Sources:

- Kroenke, "Database Processing: Fundamentals, Design, and Implementation," 7th edition, Prentice Hall, 1999.
- Ricardo, "Databases Illuminated," Jones and Bartlett.
- Rob and Coronel, "Database Systems: Design, Implementation, and Management," 3rd Edition, International Thomson Publishing, 1997.
- Connolly and Begg, "Database Systems: A Practical Approach to Design Implementation and Management," 3rd Edition, Addison-Wesley.
- Korth and Silberschatz, "Database System Concepts"
- Sunderraman, "Oracle 9i Programming: A Primer," Addison-Wesley.
- Ullman, "Principles of Database Systems," 2nd Edition, Computer Science Press.

Introduction to Normalization

Today's topic could be thought of as "database literacy" -- it is one of those "core" topics that is considered part of every database design course and text that I have encountered. If you claim to know anything about databases, you are expected to know about normalization.

That said, here is an interesting irony: if you model and design your databases properly to begin with, starting with an ERD and then converting that ERD into a database design/schema as we will discuss, then you will quite likely not need to do much to normalize your tables! But you could still walk through the normalization process to double-check this.

Reminder: Functional Dependencies

To lead into normalization, here is a quick reminder of the concept of a functional dependency:

A functional dependency is a relationship BETWEEN or AMONG attributes. If, given the value of one attribute, A, you can look up/uniquely obtain the value of another, B, then we say that B is functionally dependent on A.

So, using the example from earlier in the semester, consider an `employees` relation:

`employees(EMPL_ID, empl_name, empl_addr, empl_salary)`

If you are given the value of a particular `empl_id`, you can look up a single `empl_salary` that corresponds to that `empl_id` value. However, multiple employees might have the same salary. You could not reasonably expect to obtain a single `empl_id` given a particular salary.

So, we say that attribute B is **functionally dependent** on attribute A if the value of A **determines** the value of B (if, knowing the value of A, we can uniquely determine the value of B).

You should also recall the notation we use related to this:

```
empl_id -> empl_salary
```

...to indicate that `empl_id` determines `empl_salary`. And, you should know that `empl_id` is the **determinant** in this particular functional dependency.

Modification Anomalies

We now turn to one of the big motivations for normalization -- the concept of **modification anomalies**.

Not all relations are equally "good" -- some are definitely "better" than others. On what kind of basis might we make such a claim? Here's one: some relations turn out to work better than others over time, as far as day-to-day use in an operational database is concerned. (In contrast to other data repositories such as **data warehouses**, sometimes a database used for day-to-day operations within a scenario is called an **operational database**. In this course, we are focusing on such operational databases; the guidelines for other kinds of data repositories might be different.)

"For some relations, changing the data can have **undesirable** consequences, called **modification anomalies**." [Kroenke] The exact terminology for categories of anomalies varies a bit from text to text, and the number of such categories varies between 2 and 3, but the basic problem is the same: for operational data that changes to reflect day-to-day activities -- selling more widgets, registering more students, recording more fish swimming by a sensor, hiring new employees, etc. -- it is a problem if reasonable changes lead to unforeseen or undesirable other losses of information. For an operational database, we would prefer to **avoid** relations subject to modification anomalies.

It is easier to discuss modification anomalies in the context of a specific example. Consider the following relation `Project_Work`, expressed in tabular form as follows:

PROJ_NUM	proj_name	EMPL_NUM	empl_name	job_class	chg_per_hr	hrs
1	hurricane	101	john news	EE	65	13
1	hurricane	102	david senior	Comm.Tech	60	16
1	hurricane	104	anne ramoras	Comm.Tech	60	19
2	coast	101	john news	EE	65	15
2	coast	103	june arbro	BioE	55	17
3	satellite	104	anne ramoras	Comm.Tech	60	18

In this particular scenario, employees can work on more than one project, and how many hours an employee works on each project is of interest. The primary key, then, of this relation is `(proj_num, empl_num)`. Note, also, that `chg_per_hr` happens to be based entirely on the employee's `job_class`, rather than on the particular employee. (That is,

```
job_class -> chg_per_hr
```

...to use functional dependency notation. That is, `job_class` is a determinant for `chg_per_hr`.)

Deletion Anomalies

A **deletion anomaly** is a modification anomaly that occurs when data is deleted. Consider our example relation `Project_Work`; what if we delete the row with `empl_num 104`, for `anne ramoras`, because she has left the company?

PROJ_NUM	proj_name	EMPL_NUM	empl_name	job_class	chg_per_hr	hrs
1	hurricane	101	john news	EE	65	13
1	hurricane	102	david senior	Comm.Tech	60	16
1	hurricane	104	anne ramoras	Comm.Tech	60	19
2	coast	101	john news	EE	65	15
2	coast	103	june arbro	BioE	55	17

Do you see that we've now lost any mention of the `proj_name satellite`, as well as what `proj_num` corresponds to project `satellite`?

What if `june arbro` were likewise to leave, and we were to delete that row?

PROJ_NUM	proj_name	EMPL_NUM	empl_name	job_class	chg_per_hr	hrs
1	hurricane	101	john news	EE	65	13
1	hurricane	102	david senior	Comm.Tech	60	16
1	hurricane	104	anne ramoras	Comm.Tech	60	19
2	coast	101	john news	EE	65	15

Do you see that we've lost the data that the charge per hour for `job_class BioE` is $55 an hour?

In a **deletion anomaly**, then, deleting a row causes the loss of additional information that it may not be reasonable to have completely disappear. Removing an employee should not cause one to lose the names of projects, nor the charges-per-hour for job classes!

Kroenke has a nice way of looking of this: a deletion anomaly can occur when facts about two or more "themes" are lost with a single deletion. With one deletion, we lose facts about two (or more) "themes"!

Insertion Anomalies

An **insertion anomaly** is a modification anomaly that occurs when data is inserted. Consider our example relation; what if we would like to add information about a new project, project `6`, whose name is `lumberjack`?

Do you see that we can't do it? The primary key here is `(emp_num, proj_num)` -- we have to have an employee assigned to the project before we can keep track of its name. That's an unreasonable restriction. Likewise, what if we'd like to add a `job_class CS`, whose `chg_per_hr` is `75`? Again, we cannot do it -- we need both a related employee and a related project before we can!

In an **insertion anomaly**, then, we cannot insert a fact about one thing until we have an additional fact about another thing.

(Some also discuss an **update anomaly**, similar to the above two, for situations where it becomes awkward to update an existing row in a table -- perhaps an update to one employee's `job_class` causes the loss of how much a rarely-used `job_class`'s charge-per-hour should be.)

What we will find is that many modification anomalies can be **eliminated** by careful **redefinition** of a relation into **two or more** related relations. In most circumstances (especially in most operational database circumstances), the redefined, or **normalized**, relations are preferred. (Note that the preferences may be different for non-operational data repositories, such as read-only **data warehouses**. Our focus in this course is operational databases, however.)

We will also find that such restructuring tends to reduce **unnecessary** data redundancy -- consider if project `1`'s name changes in the above row -- three rows need to have that name changed. And consider if we want to change the `chg_per_hr` for a `Comm.Tech` to `58` -- again, every row including a `Comm.Tech` as the `job_class` needs to have the `chg_per_hr` changed accordingly.

The process of transforming relation structures in this way to reduce modification anomalies is known as **normalization**.

Normalization

So, we would like to restructure relations so that they will not have the kinds of anomalies just described. **Normalization** is a process of "breaking" a relation up into multiple tables in such a way as to reduce modification anomalies. You could say that we split the relation into two or more separate ones, each containing a single "theme", since relations containing more than one "theme" are vulnerable to modification anomalies.

Rob and Coronel put it a little differently: "Normalization is a process for assigning attributes to [relations]. Normalization reduces data dependencies and, by extension, helps eliminate the modification anomalies that result from those redundancies." However, "normalization does **not eliminate** data redundancies; instead, it produces the **controlled** data redundancy that lets us link database tables." What is this controlled data redundancy? Foreign keys -- **referential integrity constraints** that *maintain* the relationships built into the less-normalized original relation.

We will find that, when we split relations up in this way, we nearly always need to add some referential integrity constraints (some foreign keys). So, normalization does have some costs: we need to add these foreign keys, and so we may also increase the number of **equi-join** or **natural join** operations required to produce a specified query result. But, again, for an operational setting, this is usually considered a reasonable price to pay to be able to reduce modification anomalies.

So, how does one **normalize** a relation? To explain that, we need one more concept: the concept of **normal forms**.

Normal Forms

It turns out that relations can be **classified** based on the types of modification anomalies to which they are vulnerable; these classifications are called **normal forms**.

The most important/"classic" normal forms include:

- first normal form (1NF)

- second normal form (2NF)
- third normal form (3NF)
- Boyce-Codd normal form (BCNF)
- fourth normal form (4NF)
- fifth normal form (5NF)

(There are also some additional, more theoretical normal forms that are beyond the scope of this course.)

These classifications are superset/subset -- that is, any relation in 2NF is also in 1NF, any relation in 3NF is also in 1NF and 2NF, and so on.

Rob and Coronel note that one could think of normalization as working through a series of stages involving these normal forms [Rob and Coronel, p. 282]. Being in a particular normal form assures that a certain class of modification anomalies cannot occur. Being in a higher normal form does not guarantee that modification anomalies will not occur, but it does reduce them very usefully. In fact, in practice, usually one considers relations that are in 3NF as "good enough" in terms of modification anomaly reduction. Today, then, we will only discuss 1NF, 2NF, and 3NF, and how to normalize relations into these normal forms. (If we had time, we would briefly discuss the other normal forms above, but since we do not, at least be aware that they exist...)

First Normal Form (1NF)

First Normal Form turns out to be easy to describe: *any* "true" relation is, by definition, in 1NF!

So, as you should recall from our previous discussion of relations, this implies that:

- the cells must be **single-valued**
- all entries in a column must be of the same kind, from the same domain (or null)
- each column must have a unique name within a relation, but the order of the columns is insignificant
- no two rows may be identical, but the order of the rows is insignificant
- all the primary key attributes are defined (you cannot have a null primary key)
- all attributes are dependent on the primary key ("the primary key uniquely determines a row", now restated in functional-dependency terms...!)

Any true relation, then, meets these criteria.

So, will you ever need to *do* anything to transform a relation into 1NF? The most common situation where work is needed to get into 1NF is when it turns out that there is a (previously-unnoticed) multi-valued attribute. (Perhaps after creating a relation, one belatedly realizes that, for some rows, there could reasonably be more than one value for one of the attributes.) Since cells must be single-valued, a relation that somehow permitted multi-valued cells would not be a "true" relation. In practice, then, one can make sure that a set of relations is in 1NF by making sure that all attributes are single-valued within each relation.

What do you do if a multi-valued attribute is identified in this process? There are some other "ugly"

ways to handle this, but for maximum flexibility in queries and for best results, you should "break out" each multi-valued attribute into its own relation as follows:

- A new relation is created, consisting of the primary key of the "original" relation and the multi-valued attribute. Often the new relation's name indicates the nature of the multi-valued attribute. (In terms of style, though, you should avoid having table names that are the same as column names...)
- How does this work? For each value of the multi-valued attribute, you will have a row in this new relation -- so, the primary key of this new relation is the set of BOTH attributes. (You need both to uniquely determine a row in this relation.)
- In the new relation, that primary key from the "original" relation is defined to be a foreign key referencing the "original" relation.
- And, the multi-valued attribute is removed from the "original" table.

For example, consider the following relation:

```
Student (STU_ID, stu_lname, stu_fname, stu_phone, stu_gpa)
```

...and assume it is determined that, within the scenario, it is desired for a student to be able to have multiple phone numbers. That would cause the above to NOT be a true relation, and not in 1NF.

So, you'd create a new relation -- perhaps `Student_phones` -- and put the attributes `stu_ID` and `stu_phone` in it. The pair `(stu_ID, stu_phone)` would be the primary key of the new relation, and `stu_ID` in `Student_phones` would be a foreign key referencing `Student`. And, we'd remove `stu_phone` from `Student`. Thus, the resulting, normalized relations would be:

```
Student (STU_ID, stu_lname, stu_fname, stu_gpa)

Student_phones(STU_ID, STU_PHONE)
   foreign key (stu_id) references student
```

This, then, is what should be done for *every* multi-valued attribute -- it should be "broken out" into its own relation in this way.

Notice the elegance of this solution -- if you have a student with 8 phone numbers, he/she simply has 8 rows in `student_phones`. If a student has no phones, he/she simply has no rows in `student_phones`. Want to find out all of the phone numbers, if any, for a student? Join these two relations on the common column `stu_id`, and select the rows for that particular student's name or ID: voila, you'll get them all! And for roommate situations, what if you'd like all students with a particular phone number? That's easy to find, too -- join the two relations on the common column `stu_id`, and select the rows for that particular phone number. This is very flexible, and very usable. This is preferable to the common kluge of trying to set up attributes such as `phone1`, `phone2`, etc. -- you are in trouble as soon as a student has a third phone number, you have awkward null values if a student only has one phone number, and queries based on finding students with given phone numbers are unnecessarily awkward, since you have to select rows where the phone number is in either `phone1` or `phone2`, etc.

Relations that are in 1NF avoid modification anomalies due to not being relations at all -- however, 1NF relations can still be prone to an uncomfortable number of modification anomalies. After all, the example relation we used at the beginning of the modification anomaly section is in 1NF -- it is a true

relation -- and we saw definite modification anomalies there.

Second Normal Form (2NF)

A relation is said to be in 2NF if it is in 1NF, and it includes no **partial dependencies**.

What is a partial dependency? You will recall that, in our functional dependencies discussion, we noted that a statement such as:

```
A -> B
```

...means that attribute `A` functionally-determines attribute `B`, and that `B` is functionally dependent on `A`. If we know the value of `A`, we can uniquely determine the value of `B`.

In our superkey/minimal key/candidate key/primary key sequence of definitions, we noted that a primary key functionally determines all of the other attributes within a relation -- this must be so, since a primary key uniquely determines a row.

A **partial dependency**, in particular, is when a **non-primary-ke**y attribute in a table turns out to be dependent on just **part** of the primary key. Consider our earlier table, now written in relation structure form:

```
Project_work(PROJ_NUM, proj_name, EMPL_NUM, empl_name, job_class,
             chg_per_hr, hrs)
```

Remember, the primary key here is `(proj_num, empl_num)`. But notice that `proj_name` isn't functionally dependent on `empl_num` -- that is, while it surely is true (by the definition of a primary key) that:

```
(proj_num, empl_num) -> proj_name
```

... it also turns out, in this case, to be true that:

```
proj_num -> proj_name
```

This, then (`proj_num -> proj_name`) is an example of a partial dependency. (And it is not the only partial dependency in this relation.)

So, if a relation is in 1NF and has no partial dependencies -- if all of its non-primary-key attributes are dependent on *all* of the primary key -- then it is said to be in 2NF.

How can you can you transform a 1NF relation that is not 2NF into a set of relations that are? By following this procedure:

- Write each possible combination (subset) of the primary key attributes on its own line.
- For each non-primary-key attribute, place it on the line in which it is dependent on *all* of that combination of primary key attributes.
- For each line that has any non-key attributes in addition to primary-key attributes, that line should become a new relation -- give it an appropriate name, and make the primary key attribute(s) in that line the primary key of the new relation. (Remember, in terms of style, the relation name should not be the same as any of the attribute names.)
- Do you see that the tables with fewer-attribute primary keys tend to be the "home base" for that

primary key? So, those tables with more-attribute primary keys usually need to have those fewer-attribute primary keys within the more-attribute primary keys declared as foreign keys back to their "home base" tables.

As an example, let's use this procedure on our poor table `Project_Work`:

```
Project_work(PROJ_NUM, proj_name, EMPL_NUM, empl_name, job_class,
             chg_per_hr, hrs)
```

First, write each combination of attributes within its primary key on its own line:

```
PROJ_NUM

EMPL_NUM

PROJ_NUM, EMPL_NUM
```

Now, for each non-primary key attribute, put it on the line for which it is dependent on the *entire* primary key given on that line. `proj_name` is functionally dependent on only `proj_num`, so:

```
PROJ_NUM, proj_name

EMPL_NUM

PROJ_NUM, EMPL_NUM
```

Next, `empl_name`, `job_class`, and `chg_per_hr` are only functionally dependent on `empl_num`, so:

```
PROJ_NUM, proj_name

EMPL_NUM, empl_name, job_class, chg_per_hr

PROJ_NUM, EMPL_NUM
```

And `hrs` is dependent on both `proj_num` and `empl_num` -- that is the number of hours that *that* employee has worked on *that* project:

```
PROJ_NUM, proj_name

EMPL_NUM, empl_name, job_class, chg_per_hr

PROJ_NUM, EMPL_NUM, hrs
```

Each combination of primary key attributes has some non-key attributes as well, so each of these lines should correspond to a relation -- give each of these lines appropriate names and turn them into relation structures:

```
Project(PROJ_NUM, proj_name)

Employee(EMPL_NUM, empl_name, job_class, chg_per_hr)

Project_Work(PROJ_NUM, EMPL_NUM, hrs)
```

...and since `proj_num`'s "home relation" is `Project`, and `empl_num`'s "home relation" is `Employee`, then `Project_Work` needs these to be foreign keys to those relations accordingly:

```
Project(PROJ_NUM, proj_name)

Employee(EMPL_NUM, empl_name, job_class, chg_per_hr)
```

```
Project_Work(PROJ_NUM, EMPL_NUM, hrs)
   foreign key (proj_num) references project,
   foreign key (empl_num) references employee
```

See how this reduces modification anomalies? If I want to note that a project number `6` has the name `lumberjack`, that is easy to accomplish: add that row to `Project`. And now no employee deletion can cause the loss of a `project_number`/`project_name` pair.

So, tables in 2NF do not have modification anomalies due to partial dependencies.

However, you have probably noticed that this trio of tables still has some modification anomalies -- these are not due to partial dependencies, however. And we need to convert these tables to 3NF to get rid of them.

Third Normal Form (3NF)

A relation is said to be in 3NF if it is in 1NF and 2NF, and it includes **no transitive dependencies**.

What is a transitive dependency? You might recall the **transitive property** from a math class -- when:

> if (A `op` B) and (B `op` C) implies that (A `op` C),
> then that operation `op` is said to satisfy the transitive property;

A transitive dependency is when a **non-primary-key** attribute is functionally dependent on another **non-primary-key** attribute. To put is another way, it is when a non-primary-key attribute is a **determinant**. (See how transitivity plays into this? The primary key determines all of the other attributes, and if a relation is in 2NF the other attributes are dependent on all of the primary key. But, that doesn't mean that it might not be the case that, in reality,

`PK -> a` and `a -> b`, so that `PK -> b`

You can transform a 2NF relation that is not in 3NF into a set of 3NF relations by following this procedure:

- If you have more than one transitive dependency with the same determinant, combine those (for transformation purposes) into an also-true single transitive dependency --
  - that is, if you have `a -> b` and `a -> c`, then for purposes of transformation, use the also-true `a -> b, c`
- Make each resulting transitive dependency into its own relation, making the determinant of the transitive dependency into the primary key of the new relation.
- Remove the right-hand-side attribute(s) in that transitive dependency from the original relation.
- Leave the determinant in the original relation, *BUT* now *also* make it a foreign key referencing the new relation.

Do you see how there are now no transitive dependencies? In the new table, the determinant is now a primary key, not a non-primary-key. And since you've removed the attributes dependent on that determinant into the new table, there is no longer a transitive dependency in the old table, either.

Considering our now-2NF example relations:

```
Project(PROJ_NUM, proj_name)

Employee(EMPL_NUM, empl_name, job_class, chg_per_hr)

Project_Work(PROJ_NUM, EMPL_NUM, hrs)
   foreign key (proj_num) references project,
   foreign key (empl_num) references employee
```

...you may recall that it was noted earlier that `job_class -> chg_per_hr`. Since `job_class` is thus a determinant that is not a primary key, then this is a transitive dependency, and it must be removed for this set of relations to be in 3NF. So, we create a new relation reflecting this:

```
Job_Class_Chg(JOB_CLASS, chg_per_hr)
```

...giving it the determinant of the transitive dependency as its primary key.

And we need to change `Employee` to remove the attribute `chg_per_hr`, and make `job_class` a foreign key referencing this new relation:

```
Employee(EMPL_NUM, empl_name, job_class)
   foreign key (job_class) references job_class_chg
```

The resulting collection of relations is:

```
Project(PROJ_NUM, proj_name)

Job_Class_Chg(JOB_CLASS, chg_per_hr)

Employee(EMPL_NUM, empl_name, job_class)
   foreign key (job_class) references job_class_chg

Project_Work(PROJ_NUM, EMPL_NUM, hrs)
   foreign key (proj_num) references project,
   foreign key (empl_num) references employee
```

...which is now in 3NF.

See how this reduces modification anomalies? If I want to note that a `job_class` of `CS` should have a `chg_per_hr` of `75`, I can now do so: simply add that row to `Job_class_chg`. And no employee deletion can cause the loss of a `job_class/chg_per_hr` pair.

So, tables in 3NF do not have modification anomalies due to transitive dependencies.

A Few Words on those Other Normal Forms...

Usually, this is considered sufficient -- the remaining modification anomalies are rare enough that 3NF is usually sufficient for operational databases. Most database textbooks include some discussion of the other normal forms, however, if you are interested (and one can certainly google the others for further information as well). But, here is a quick summary:

- Boyce-Codd Normal Form (BCNF):
  - a relation in which every determinant in that table is a candidate key
- Fourth Normal Form (4NF):

  - a relation in 3NF that includes no multiple sets of multivalued dependencies
  - (that is, you cannot have two different attributes that are multi-valued with respect to a third attribute)
- Fifth Normal Form (5NF):
  - this normal form has to do with relations that can be divided into sub-relations, but then cannot be reconstructed; that's as far as we'll go with that.

A Few Normalization Caveats and Final Comments

Sometimes, we make the conscious decision not to take relations all the way to 3NF. The best example I've heard of this involves zip codes: zip codes, in reality, are dependent upon street address and city. However, unless one's scenario is a post office or heavy-duty delivery company, one usually doesn't break this transitive dependency into its own relation. In many scenarios' databases and database applications, you don't really worry about whether you will lose the connection between a street address/city and the corresponding zip code due to the deletion of some item that has address attributes amongst its attributes -- likewise, you probably don't try to just store a street/city and zip by themselves. So, since these are not modification anomalies of concern in many scenarios, we often do not break out this transitive dependency. This is a useful point to keep in mind regarding normalization.

When one is designing a read-only **data warehouse** -- which is beyond the scope of this course -- the different needs of this kind of data repository actually may lead to one deciding to **denormalize** some sets of relations, to deliberately take those sets of relations back to a lower normal form! Because our interest in this course is in operational databases, we will not do such **denormalization**; however, it is good to know that this is sometimes done for other data repositories with different goals and needs.

Finally, after we have discussed converting a database model into a database design during the next two lectures, I hope you will see that following that model-then-design process should result in a set of relations that are often already in 3NF, **if** the original model is sound, (except perhaps for transitive dependencies we don't really care about, such as the `(street_addr, city) -> zip_code` dependency described above). What Kroenke calls "themes" are probably analogous to entity classes in an entity-relationship diagram -- if your ER model for a scenario really includes all of the significant entities within a scenario, then its corresponding design has a pretty good chance of being in 3NF.

You might still walk through a normalization process with your resulting design's relations to make sure that they are indeed in 3NF (or as close to that as you care to be), as a useful double-check on your model; when might you discover additional normalization as being needed? I can think of several possibilities:

- if you have one or more entities embedded within an association entity (often a significant-transaction or significant-action entity), you would likely have one or more partial dependencies in the resulting set of relations;
- if you have an entity accidentally embedded within another entity, you would likely have a transitive dependency in the resulting set of relations.

Going through the normalization process with your resulting database design thus might end up improving/clarifying your underlying ER model.