MOUNT ROYAL
UNIVERSITY
1910

MOUNT ROYAL UNIVERSITY

Department of
MATHEMATICS AND COMPUTING

COMP 4522

Database II: Advanced Databases

Contents

EXERCISE

FIVE

NORMALIZATION

Why do we normalize?
How do we normalize?
When we would not want to normalize?
When can we denormalize?

THE RELATIONAL MODEL defines properties of valid relations, see 2 but even if you follow the basic rules for forming relations, integrity constraints and the like, it is still possible to create bad relations. We need more than just valid relations, we need well formed relations!

Look at the following table which is a valid relation by the criteria in 2.

No.	Name	Elevation	Country
1.	Mt. Robson	3958	Canada
2.	Mt. Washington (Agiocochook)	1916	USA
3.	Chomolungma	8848	Tibet
4.	Mt. Victoria	3564	Canada
5.	Chief Mountain	2769	USA
6.	Bugaboo Spire	3204	Canada

This is a nice table that lists some mountains and their elevations. Say we also wanted to know who was the first person to climb each mountain (first ascent is a big deal in mountain climbing). We could add columns.

No.	Name	Elevation	Country	First
1.	Mt. Robson	3958	Canada	Conrad Cain
2.	Mt. Washington (Agiocochook)	1916	USA	Unknown 1642
3.	Chomolungma	8848	Tibet	Tenzing Norgay
4.	Mt. Victoria	3564	Canada	J. Norman Collie, Arthu
5.	Chief Mountain	2769	USA	
6.	Bugaboo Spire	3204	Canada	Conrad Cain
7.			Alan Fedoruk	Canada

And this is still a valid relation by the criteria in 2 but we see some issues if we need to update the table.

- Mt. Victoria was first climbed by multiple people and so we have three names in one space!

- We have missing data for Chief Mountain. Does that mean never climbed? No, just unknown.

- We realize that Tenzing Norgay had a guy with him on the first ascent, Edmund Hillary, so how do we add him?

- We realize that Conrad Kain's name is spelled wrong, it should have a K rather than a C. We need to make the change in multiple places. If we don't then we have two people, Conrad Cain and Conrad Kain.

- If we decided that this database is only for North American mountains we could delete the Chomolungma row, but we would also delete the information about Tenzing Norgay, that we may want to keep.

- We notice that Chomolungma has more than one name (Everest, Sagarmatha), how do we record that? Mt. Washington has more than one name. It is quite common for geographical places to have many names.

- What if we want to include climbers who never had a first ascent? We would have to add a row like the last

These are all examples of *anomalies* that can occur when updating a relation, even one that has the properties we want.

- **Insertion Anomaly**: adding new rows forces user to create duplicate data

- **Deletion Anomaly**: deleting rows may cause a loss of needed data.

- **Modification Anomaly**: changing data in a row forces changes to other rows due to duplication of data.

We need a way to create relations that are not just valid, but are well formed to avoid these problems.

First we need some definitions.

Functional Dependency Within a relation, there are relationships between the attributes. We refer to these relationships as *functional dependencies*, and there are three types:

1. **Primary Key**: All of the non-key attributes depend on the key.
2. **Partial**: Some non-key attributes depend on part of the key.
3. **Transitive**: A non-key attribute depends on another non-key attribute.

Functional dependencies are expressed as $A \to B$, which means that A determines B and A is the *determinant*.

Candidate Key A *candidate key* is a determinant that determines all the other columns in the relation. A relation can have more than one candidate key.

If a relation has any partial or transitive dependencies, it is possible that we may introduce anomalies when inserting, updating and deleting rows. The anomalies can be losing data, or missing an update.

Now we can look at the process of *normalization*.

We essentially break up our relations up into smaller ones until only primary key dependencies remain. First, we remove partial key dependencies, then we remove transitive dependencies. At each stage, our relations are in different normal forms.

- **1st Normal Form (1NF)** = Meets the conditions for a relation as described in the relational model and has a primary key defined.
- **2nd Normal Form (2NF)** = 1NF + All non-keys are identified by the whole key
- **3rd Normal Form (3NF)** = 2NF + All transitive dependencies are removed
- **Boyce-Codd Normal Form (BCNF)** = 3NF + Every determinant is also a candidate key

In the end, we want the attributes to depend on *the key, the whole key and nothing but the key. So help you, Codd*.

Normalization Process

Process for Putting a Relation into BCNF

1. Identify every functional dependency.
2. Identify every candidate key.
3. If there is a functional dependency that has determinant that is not a candidate key:
   (a) Move the columns of that functional dependency into a new relation.
   (b) Make the determinant of that functional dependency the primary key of the new relation.
   (c) Leave a copy of the determinant as a foreign key in the original relation.

   (d) Create a referential integrity constraint between the original relation and the new relation.

4. Repeat step 3 until every determinant of every relation is a candidate key;

Another way to think about normalization is themes. When you are writing an essay each paragraph should have a single theme or idea. If not, you break it up into two or more paragraphs. Look at table. If it has more than one theme, it needs to broken up.

The mountain table has two themes, mountains and climbers and hence the issues!

Commentary

In practise if you are developing a database from scratch, you rarely need to do much normalization. The process of creating an ERD and transforming the entities and relationships into relations, if done well, will result in normalized relations. Often though, you are not building from scratch but your inherit some tables. Sometimes there will be anomalies that users have noticed and you will need to normalize. A common occurrence is users realize that an attribute needs to be multi-valued rather than single valued. Another is when you get a “database” prepared by someone who did not take COMP 2521 and you need to fix it. These are often large Excel spreadsheets that have been saved as a *.csv* and therefore become a “database”.

Normalizing is complicated and results in more tables and so when using the database you need to create DML with lots of joins. For example, your PERSON record in the student system banner has many tables: An ID table, a name table, an address table, an email table, a phone number table, a biographical information table and an emergency contact table. That is seven tables just to capture basic information about a person. When retrieving the data all those tables need to be joined. Queries and updates in a fully normalized database have more overhead in both space and time. Separate tables take more space, and each table needs an index or two which take space and processing to maintain.

Sometimes application designers fall into the trap of designing the interface around the normalized tables, rather than from a user perspective. That topic is for another class, but it is yet another drawback of fully normalized databases.

So why do we do it? For Flexibility and Correctness. If a person can have more than one name, they could have 1, 2, or 20. Our database needs to handle that. A student needs to be able to apply for multiple programs. And most importantly, for most databases we need to be able to rely on the data being correct! By reducing the possibility of an anomaly, we can be more confident that our data is correct. You want your bank account balance to be accurate. You want your transcript to have your true grades.

There is a thing that you may have noticed in all this. All of the trouble with anomalies starts when we start to update the database. If we are not updating things we cannot have those anomalies. That leads us to a discussion around de-normalization. Just when you get good at normalizing, we pull the rug out and say that we can also denormalize.

De-normalization

Normalization optimizes a database for consistent updates, but it may be that we may want to optimize something else. We could be trying to optimize to reduce computing resources (CPU and storage), or we may have a system that rarely is updated and so we can optimize for retrieval. The latter is the case with a data warehouse, which we will get into next. Or we may be more concerned with data privacy or our database may be widely distributed and locating data close to where it is being accessed may be important.

While processing and network speed continues to increase and the price and efficiency of storage improves, so does the size of our databases so there may always be times to de-normalize.

There are two ways to denormalize and whatever we do, we do it cautiously.

Recombining Normalized Tables

Some examples include,

- with 1:1 relationships. You can eliminate a table and a join.
- with many to many relationships you have three tables. You can make this two and avoid a join.
- Static reference data. This data may be mostly static and while it will be repeated, it may be OK.

but there are other possibilities. Doing this reintroduces transitive and partial key dependencies so we will need to ensure that our processes and procedures are done in a way that will not introduce anomalies.

Partitioning

The previous cases all involve joining tables but it is also possible to split normalied tables up into smaller tables.

Consider this relation.

ID	Name	SIN	Street	City	Prov
1	Red	123456789	1st	Calgary	AB
2	Green	123456789	2nd	Calgary	AB
3	Blue	123456789	4rd	Saskatoon	SK
4	Yellow	123456789	Main	Saskatoon	SK
5	Pink	123456789	Broadway	New York	NY
6	White	123456789	Broadway	New York	NY
7	Mauve	123456789	McLeod	Calgary	AB
8	Magenta	123456789	Robson	Vancouver	BC
9	Cyan	123456789	St. Catherines	Montreal	PQ
10	Purple	123456789	Bloor	Toronto	ON

It is perfectly — yes perfectly — normalized. There is only one dependency, the ID determines everything else. So we could just leave it as is.

We could *horizontally* partition the table by province. You may do this to make the size of the table more manageable, to locate the partition closer to where the data is being used, and to provide extra security (people only see and work with the data that they are supposed to).

A good example is student records. As chair of Math and Computing I only need to see student information for students in the programs and courses our department runs. That would be a partition by program. (They don't do this, I can see everyone's data, which is a FOIP violation, strictly speaking.)

Advantages:

- Efficiency;
- Local optimization;
- Security;
- Maintenance

- Load Balancing

Disadvantages:

- Inconsistent access speeds;
- Complexity;
- Extra space and update time;
- Maintenance
- Load Balancing

Formally this is:

$F = \{f | f = \sigma_{\phi}(R)\}$

where F is a set of fragments, R is some relation, ϕ is a condition, range, list, etc. to assign rows to a partition.

We must ensure that all rows in R are present in the partitions, that rows are not duplicated in partitions, and that the original R can be reconstructed (with a Union).

It is also possible to *vertically* partition a relation R

$F = \{f | f = \pi_{pk,a_1,\ldots a_n}(R)\}$

Where pk is the primary key of R and appears in all partitions. Again we must ensure all attributes are accounted for and that the original can be reconstructed.

There is typically done to ensure privacy of data at the column level. For example we could partition the table above into two. This would allow us to provide address data to those users that need it but protect the more sensitive SIN data.

ID	Name	Street	City Prov	
1	Red	1st	Calgary	AB
2	Green	2nd	Calgary	AB
3	Blue	4rd	Saskatoon	SK
4	Yellow	Main	Saskatoon	SK
5	Pink	Broadway	New York	NY
6	White	Broadway	New York	NY
7	Mauve	McLeod	Calgary	AB
8	Magenta	Robson	Vancouver	BC
9	Cyan	St. Catherines	Montreal	PQ
10	Purple	Bloor	Toronto	ON

ID	SIN
1	123456789
2	123456792
3	123456795
4	123456798
5	123456801
6	123456804
7	123456807
8	123456810
9	123456813
10	123456816

Exercise

For each example, find the candidate keys, the functional dependencies, then create new relations in BCNF.

1. Sporting Goods

SKU	SKUDescription	Department	Buyer
100100	Std Scuba Tank, Yellow	Water Sports	Joe Simpson
100200	Std Scuba Tank, Blue	Water Sports	Joe Simpson
100300	Std Scuba Tank, Green	Water Sports	Joe Simpson
100400	Std Scuba Tank, Red	Water Sports	Joe Simpson
100500	Std Scuba Tank, Orange	Water Sports	Joe Simpson
100600	Std Scuba Tank, Purple	Water Sports	Joe Simpson
101100	Dive Mask, Small Clear	Water Sports	Fred Beckey
101200	Dive Mask, Med Clear	Water Sports	Fred Beckey
201000	Half Dome Tent	Camping	Edmund Hillary
202000	Half Dome Tent, Vestible	Camping	Edmund Hillary
203000	Half Dome Tent, Vestible, Wide	Camping	Edmund Hillary
301000	Light Fly Climbing Harness	Climbing	Tenzing Norgay
302000	Locking Carabiner, Oval	Climbing	Tenzing Norgay

2. Equipment Repair

ItemNum	EquipmentType	AcquisitionCost	RepairNum	RepairDate
100	Drill Press	3500.00	2000	2018-05-05
200	Lathe	4750.00	2100	2018-05-07
100	Drill Press	3500.00	2200	2018-06-19
300	Lathe	27300.00	2300	2018-06-19
100	Drill Press	3500.00	2400	2018-07-05
100	Drill Press	3500.00	2500	2018-08-17

3. Sporting Goods

SKU	SKUDescription	Department	BudgetCode	Buyer
100100	Std Scuba Tank, Yellow	Water Sports	BC-100	Joe Simpson
100200	Std Scuba Tank, Blue	Water Sports	BC-100	Joe Simpson
100300	Std Scuba Tank, Green	Water Sports	BC-100	Joe Simpson
100400	Std Scuba Tank, Red	Water Sports	BC-100	Joe Simpson
100500	Std Scuba Tank, Orange	Water Sports	BC-100	Joe Simpson
100600	Std Scuba Tank, Purple	Water Sports	BC-100	Joe Simpson
101100	Dive Mask, Small Clear	Water Sports	BC-100	Fred Beckey
101200	Dive Mask, Med Clear	Water Sports	BC-100	Fred Beckey
201000	Half Dome Tent	Camping	BC-200	Edmund Hilla
202000	Half Dome Tent, Vestible	Camping	BC-200	Edmund Hilla
203000	Half Dome Tent, Vestible, Wide	Camping	BC-200	Edmund Hilla
301000	Light Fly Climbing Harness	Climbing	BC-300	Tenzing Norga
302000	Locking Carabiner, Oval	Climbing	BC-300	Tenzing Norga