CS 325 - Reading Packet: "More database fundamentals"

SOURCES:

- Kroenke, "Database Processing: Fundamentals, Design, and Implementation", 7th edition, Chapter 1, Prentice Hall, 1999.
- Connolly and Begg, "Database Systems: A Practical Approach to Design Implementation and Management", 3rd Edition, Addison-Wesley.
- Rob and Coronel, "Database Systems: Design, Implementation, and Management", 3rd Edition, Thomson, 1997.
- Wikipedia, http://en.wikipedia.org, accessed 2015-06-17.

Interlude: Definition of a Database

Definition of a Database

Earlier, we informally defined a database to be a collection of tables, holding information about different interrelated entities. Now we are going to be a bit more formal, defining a database as follows:

"A database is a self-describing collection of integrated records", (Kroenke, p. 14).

This term **self-describing** is important; it is talking about how a database is more than just the desired data. It means that a database contains **metadata** -- data **about** the data, information about the structure of the data -- in addition to the data itself. This metadata may be in the form of a data dictionary, which we'll discuss a bit in a later packet.

Why do we want a database to be self-describing? Because this description of its own structure can be used by a program (say, the DBMS) to determine what a database contains. And other programs besides the DBMS can sometimes use metadata to good effect as well; this can be another angle promoting program/data independence.

Now consider the "collection of integrated records" aspect. This means you not only have source data, but also a description of the relationships among the data records; that's what makes the records integrated. That is, in addition to metadata, a database includes indexing information that represents relationships among the data (and that can also be used to improve the performance of database applications).

Let us consider one more important concept before going on. It is useful to realize that a database is a model -- a simplified abstraction of something -- but it is not really a model of reality. It is more helpful and more accurate to think of a database as a model of a model. When modeling reality, you might need to include many more details; for example, imagine all the details required to reasonably model a real-world person! But a database doesn't have to model reality -- it does need to model the *users'* model of their "world" -- of their business or organization or inventory or data collection or study, etc. The degree of detail provided in a well-designed database should be based on the users' needs and desires, not on all of the possible levels of detail possible. Any person might have favorite colors, but a university database probably does

not need to record a student's favorite colors, although a personal shopper database might very well need to record a client's favorite colors.

History Part 2: to Relational Databases

The Organizational Context

Remember, databases were originally designed to overcome file-based processing shortcomings experienced by large companies in the 1960's as the amount of data they generated increased to the point that it could not be managed well with a file-based approach. These early databases, then, tended to be organizational databases --- encompassing the whole, or large portions of, the organization involved --- and the applications being supported by the databases were often organization-wide, transaction-processing systems.

What is a **transaction**? One definition is a representation of an event; we'll have another definition later in the semester. For many companies, for example, a sale is an example of a transaction. For a bank, a deposit and a withdrawal are two examples of different kinds of transactions.

There were a lot of growing pains in the development of database-processing systems. So, early databases (mid-1960's) served the organizational level --- they focused on transaction processing of organizational data for major corporations producing reams of data at huge rates. Eventually, these databases got to be very good at dealing with keeping track of regular transactions, or for creating regularly scheduled reports.

BUT, these early database systems were not very flexible, and application programs accessing the databases tended to need to be written in procedural languages such as COBOL and PL/I. Users might have a question the data could answer, but they might not be able or willing to wait for a programmer to get around to writing a program to answer their question.

The Relational Model

E. F. Codd developed the relational model in 1970, based on relational algebra. In a relational database, you consider data to be stored in the form of relations, a formal kind of tables; we'll be discussing this in more detail later! Relations/tables have rows and columns. In the relational model, data is stored as tables, and relationships between rows of tables are visible in the data.

It turns out that this relational model permits one to ask questions about the data in very flexible ways, as we'll see. But there are other benefits of the relational model, as well. For one thing, in a well-designed relational database, data ends up being stored in a way that minimizes duplicated data, and even eliminates certain kinds of processing errors that can arise when data are stored in other ways. For another, columns can be used to contain data that relate one row to another. In general, it is simply easier to think about data in the relational model as compared to the earlier, hierarchical and network, database models. And, because support for ad-hoc queries (those spur-of-the-moment questions about one's data) is more practical in the relational model, this could potentially encourage more creative uses of one's data.

There was initial resistance to the relational model; there is a lot of overhead involved in maintaining this powerful abstraction. Many thought this model would never be practical, that it would always be too slow. Fortunately, computer hardware, memory speed, and power increased (and price decreased) to where it could be practical. And the development of this relational model in 1970, combined with the advent of microcomputers a few years later, encouraged the

development of DBMS software usable on a personal level. (Although one needs to be careful -- some of the earliest personal "database" programs were not DBMSs at all. For example, the original Ashton-Tate dBase II was neither a true DBMS, nor a true relational database system --- (Kroenke, p. 18) "it was a programming language with generalized file-processing capabilities"! But, gradually, true relational DBMSs were moved from the mainframe to the microcomputer, and the microcomputer products were converted to true DBMSs.

And consider your typical "home" computer user: they won't put up with the ugly (clumsy, awkward) user interfaces that were common to mainframe applications. So, this encouraged a dramatic improvement in DBMS user interfaces. And this combination of widely-available microcomputers, the relational model, and improved user interfaces helped database technology to move from an organizational context to a personal-computing context, and really allowed it to simply become more accessible in general to a wider variety of settings and applications.

We'd be remiss if we didn't also mention how this setting, along with the (middle-to-late-1980s) trend for users to connect their microcomputers and workstations together using local area networks, helped to set the stage for client-server database applications. If not everyone's workstation could have its own local DBMS running, it could perhaps connect to another DBMS server running on another computer on the network; many clients can use a database running on a server elsewhere.

And it is a logical progression from there, once you have DBMS servers that can be accessed via the Internet, to have users using database applications -- supported by DBMSs -- across the Internet. And, there is increased demand for these DBMSs to support more kinds of data, including multimedia.

Of course, the field of databases is still a developing field. For example, there is a growing area of research and interest in **distributed database processing**: what would it mean for a database not to be centralized, at a single site, but instead distributed across several sites? Perhaps with the data repeated, or replicated, at more than one of those sites? How might that increase availability of data and performance, while still keeping a high degree of data integrity and security? Another area of research and interest is in **object-oriented databases** and **object-relational databases**, trying to bring in some of the advantages of objects to databases, and trying to determine what it would mean for objects to persist between executions of object-oriented programs. And for situations where large quantities of data are streaming in at a high rate, some are using deliberately non-relational databases -- oddly called **NoSQL databases**, which Wikipedia says is often interpreted to mean "Not only SQL" databases -- an example of which is MongoDB.

FOUR MAIN ELEMENTS of a DATABASE

We said, last time, that "a database is a self-describing collection of integrated records". Let's talk a little more about what that means, and talk about what, then, is in a database.

We already discussed how a database is self-describing -- in addition to the actual user data, it also includes **metadata**, a description of its own data, often in the form of a data dictionary, which we will discuss later in the semester. But a database also often includes a few more elements as well.

For example, a database also includes **indexes**: these are what make the database a collection of integrated records. Indexes are used to represent relationships among the data, and to improve the performance of database applications.

In addition, databases may also contain data about the applications that use the database --- the structure of a data entry form, or a report, etc. That's what some call **application metadata**.

So, one can generally consider most databases as including at least four kinds of "things", four categories of elements:

- user data
- metadata
- indexes
- application metadata

User data

User data is the most obvious of the elements in a database! In a relational database, user data is represented in the form of special tables called **relations**. We'll be discussing a more formal definition of relations, but for now you can consider a relation to informally be a table of data, with rows and columns. This is the proper mental model for the user data in a relational database -- user data is a collection of relations/tables, NOT a collection of files.

One can depict such relations/tables in a variety of ways, depending on your purpose. We will use several depictions of relations/tables in this course, and you are responsible for being comfortable with each of them.

Tabular Form

One of these ways is in **tabular form**, with rows and columns and column headings. For example, a relation `student` might be expressed in tabular form as:

```
student:
student_name        student_phone       adviser_name
------------        -------------       ------------
Jones, Jane         123-4567            Smith, Ann
Nguyen, Anh         234-5678            Silva, Jay
Garza, Juan         345-6789            Schmidt, John
```

Some of the advantages of tabular form include:

- it is straightforward;
- you can see the table's contents; and
- you can see the basic relation structure.

Some of the disadvantages of tabular form include:

- you cannot (usually) see what a relation's **primary key** is;
  - [**Important Aside**: what is a primary key?

    Like the precise definition of a relation, we will defer the precise definition of a relation's primary key for later. For now, we'll say that is is the column or collection of columns whose values **uniquely identify** a row -- that is, the column or collection of columns whose values **cannot be the same** for any two or more rows.

For example, the combination of `student_name` and `student_phone` would probably be a suitable primary key for the table depicted in tabular form above (although a better design for this particular table would be to change the table to include a unique student identifier -- for example, a student's ID number -- and to use that as the primary key instead).]

- you cannot see the specific **domain** for each column (although you often can imply/infer it);
  - [**Important Aside**: what is a column's **domain**?

    The domain of a column is the type/kind of value that is permitted for the values of that column in table rows. (If you prefer, it is the type/kind of value permitted in a "cell", the intersection of a row and a column.) For example, the domain of an `item_quantity` column might be integers greater than or equal to 0, while the domain of a `student_name` column might be strings representing a person's legal or preferred name, and the domain of a `course_added` column might be dates during the period of time that one can add a course to one's schedule.]
- you cannot see how relations **relate** to each other; and,
- this form can be inconvenient to write/type.

Interestingly, only occasionally will we use tabular form to represent a relation in this course. Much of the time, you want to talk about a table in general, and then you are not really interested in actual data, but in the relation's structure -- what columns does it have? What is its primary key? How is it related to other tables? For this kind of general discussion of a relation, the tabular form is clunky and inconvenient (and too big/detailed); often, a much simpler format will suffice.

Relation Structure Form

This simpler format is called **relation structure** form (and you'll be seeing it on exams, in homework problems, and many times during discussions, so you should get comfortable with this format, and know that this is the format meant when I ask for a relation "in relation structure form"!)

In relation structure form, one represents a relation's basic structure by:

- starting with the name of the relation,
- then putting an opening parenthesis,
- then giving a comma-separated list of the relation's column names (also called the relation's **attributes**), with the column or columns making up the primary key visually distinguished in some way, such as being written in all-uppercase, or in bold style, or underlined, or some eye-catching and consistent combination of these),
- then putting a closing parenthesis.

(We'll sometimes follow this with some additional information, to be discussed later in the semester.)

Consider the relation `student` given in tabular form earlier -- it could be expressed in relation structure form as:

```
student(STUDENT_NAME, STUDENT_PHONE, adviser_name)
```

Some of the advantages of relation structure form include:

- it is still straightforward, but it is also concise and easy to write or type;
- it makes the relation's basic structure apparent; and,
- you should be able to easily see what the relation's primary key is.

Some of the disadvantages of relation structure form (depending on your point of view) include:

- you cannot see the table's contents; (but, for a surprising number of purposes, you don't need to;)
- you cannot see or infer the domains for the values of the columns; and,
- you cannot see how relations relate to one another unless you add additional notation (which we sometimes will, as will be discussed later).

We will be using relation structure form frequently -- it is very convenient as a first format for a **database design** (also called a **database schema**), which will be a major topic later in this course, even though relation structure form lacks some important information you'll need before considering a database design/schema to be complete.

Create-table Form

The third format we'll sometimes use for relations is to use the actual statements that one can use to create relational tables in the language SQL (Structured Query Language, pronounced "sequel" or "ess-cue-ell"). We'll call this **create-table form**, or I might just say "write a relation in the form of a SQL `create table` statement".

The SQL `create table` statement is introduced in the first SQL Reading Packet, but here is an example of such a statement for the `student` relation used previously:

```
create table student
(student_name    varchar2(30),
 student_phone   char(8),
 adviser_name    varchar2(30),
 primary key     (student_name, student_phone));
```

Some of the advantages of create-table form include:

- you can see the relation's basic structure;
- it is easier to type than tabular form;
- you should be able to see what a relation's primary key is;
  - (It is considered good style -- and in this course, it is a class coding/style standard -- to always explicitly declare a primary key in a `create table` statement.)
- you can see the specific physical domain for each column; and,
  - (We'll discuss column domains further in a future reading packet -- but by "physical" domain, I mean at least the rough type of values allowed to be in a particular column.)
- even though you do not see it in this example, you usually should be able to see how relations relate to each other using this form, as we'll also be discussing in a future packet.

Some of the disadvantages of create-table form (depending on your point of view) include:

- it is a little less straightforward and a little less convenient to type than relation structure form; and,
- you still cannot see the table's contents (but, again, you don't always need to).

There are surely many other ways to represent relations as well -- for example, consider Microsoft Access, which provides a Table definition view, or a graphical view of a database's tables as little rectangles. But these will be our three major means for depicting relational tables this semester.

User Data-related aside: introduction to the importance of good relation/table design

We will often discuss relations in terms of their structure (without discussing their contents). One reason for being interested in a database's relations' structure is because one set of relations may often not be as good as another for a given purpose. Even when the same data is included, how you structure the relations in a database can make a significant difference in usability, long-term data integrity, and more.

That is, relations can be considered to be well- or poorly-structured -- or, well- or poorly-designed.

Consider the relation:

```
student1(STU_NUM, stu_last_name, stu_phone, adviser_id,
         adviser_last_name, adviser_phone)
```

versus the pair of relations:

```
student2(STU_NUM, stu_last_name, stu_phone, adviser_id)

adviser(ADVISER_ID, adviser_last_name, adviser_phone)
```

Do you see how the relation `student1` has information about 2 different "things", students and advisers? If an adviser's phone number changes, you have to change that for **every** student having that adviser! For example, if an adviser advises 20 students, then you would need to change all 20 phone number instances! And, notice that the adviser advising 20 students has their phone number in that database 20 times -- that's **unnecessarily duplicated** data.

In the second pair of relations, notice that each adviser's phone number only appears once, and that changing an adviser's phone number only requires a single change. You may very properly ask: how can you (easily/reasonably) find out the phone number for student Baker's adviser? The answer is, through **relational operations** (using **relational algebra**) -- which lets you manipulate relations! And we'll be discussing those operations in an upcoming reading packet.

It turns out that, for day-to-day, so-called **operational** use of data, especially data that changes frequently, it works better to store the relations separately, and virtually combine them as needed using relational operations, than it is to store them as a combined table. (This may **not** be true for all uses of data -- it often is not the case for **data warehouses**, which combine multiple databases and data stores for data analysis. But those are beyond the scope of this course, although we'll occasionally mention them.)

Also notice: **not** all data duplication has disappeared -- `adviser_id` appears in more than one relation, and a given `adviser_id` value may occur in relation `student2` more than once. But we'll be arguing that that is **necessary** data duplication, in this case, to allow for **integration**

of the data -- for example, we'll see in a later reading packet that this can allow us to reasonably determine the name and/or phone number of a particular student's adviser.

SECOND MAIN ELEMENT of a DATABASE: metadata

Remember, **metadata** is data about the data, the description of the structure of the database -- it is what makes the database **self-describing**.

Since, in a relational DBMS, the user data is stored in the form of relational tables, these often store the metadata in the form of relational tables, too -- sometimes these are called s**ystem tables**, and collectively these are sometimes called a **data dictionary**. For example, the DBMS might include a system table giving the tables in the database -- as a fictitious example,

```
sys_tables(TABLE_NAME, number_of_columns, primary_key)
```

A system table giving information about the columns (attributes) in each table might look like:

```
sys_columns(COL_NAME, TABLE_NAME, data_type, length)
```

These system tables are not just useful for the DBMS -- they are also often useful for users, too, who are often given access to query appropriate subsets of these tables. In Oracle, for example, you can access such system tables as `user_catalog`, `user_objects`, `user_tables`, `user_views`, and `user_tab_columns` for the tables in your account (in your "database").

THIRD MAIN ELEMENT of a DATABASE: indexes

Really, this element should be stated as "overhead data such as indexes". This is data that improves the performance and accessibility of the data (although some include the information that makes the data integrated as part of this data as well. Others include that data under "metadata".) In the classic sense, however, indexes are rather literally meant as *additional* data that reduces the amount of work needed to sort and search tables -- in this classical sense, indexes included just to improve performance are sometimes called **physical indexes**.

(That is, using a DBMS, you have the idea of a table of data, and this is physically stored in some data structure, and you do not have to know or care what kind of a data structure is used, thanks to the abstraction provided by the DBMS. A physical index is something added by the DBMS to whatever that data structure is -- for example, a pointer, or a new table entry -- to make certain searches faster. For example, you could define a physical index for a column -- especially a non-primary-key column -- that you think will be frequently-searched, so that the DBMS would include appropriate additional data to make searching those columns more efficient

Such physical indexes do help with sorting and searching, but, of course, at a cost -- the DBMS must update them, for example, every time a row in a table with such a physical index is updated. This is not necessarily awful, but physical indexes are not free, either, and "should be reserved for cases where they are truly needed". We will not be discussing physical indexes further in this course.

FOURTH MAIN ELEMENT of a DATABASE: application metadata

Application metadata is "used to store the structure and format of user forms, reports, queries, and other application components". Not all DBMSs support "application components", and of those that do, not all store their structure in the database. But you have probably encountered some DBMSs that do this -- Microsoft Access, for example, includes tools for making reports, and a resulting report's specifications are stored as part of an Access database, even though that

report specification is really a database application. Note that, in general, neither developers nor users access this application metadata directly -- they use tools in the DBMS to do so.