CIS 315 - 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.

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 also be called a data dictionary or a data directory.

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 not a model of reality. Rather, it is a **model of a model** -- it is a model of the users' model of their business, their organization, their inventory, their data collection, their study, etc. The degree of detail provided 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 is a much easier way to think about databases, and it 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, support for ad-hoc queries (those spur-of-the-moment questions about one's data) is more practical in the relational model; this encourages 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 though 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 DBMS's 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 DBMS's were moved from the mainframe to the microcomputer, and the microcomputer products were converted to true DBMS's.

And consider your typical "home" computer user: he/she 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-1980's) 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 living 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 DBMS's -- across the Internet. And, there is increased demand for these DBMS's 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.

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 called a data dictionary. 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, you can consider a database to contain four main elements:

* **user data**
* **metadata**
* **indexes**
* **application metadata**

FIRST MAIN ELEMENT of a DATABASE: 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 **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/data, 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:

```
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;
* you can see the basic relation structure;

Some of the disadvantages 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, and to use that as the primary key instead).]

* you cannot see specific **domain** information (although you often CAN **imply/infer** it);

  [IMPORTANT ASIDE: WHAT is DOMAIN INFORMATION?

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.)]

* you cannot see how relations **relate** to each other;

* it can be a pain to 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); 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, and in homeworks, and many times in discussions, so GET FAMILIAR with this format, and KNOW that this is the format meant when one asks for a relation "**in relation structure form**"!!)

In **relation structure** form, one represents a relation's structure:
* with the **name** of the relation,
* then an open parenthesis,
* then the names of the attributes (the names of the columns), with the column or columns making up the PRIMARY KEY distinguished in some way, by being written in all-uppercase, or boldface, or underlined, or some eye-catching and consistent combination of all 3...!

Consider the relation given in **tabular form** before/above --- it would 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 is also nice and concise, and easy to type;
* you can see the basic relation structure;
* you should be able to easily see what the primary key is (if written using good style, with the primary key attributes written in a distinctive, eye-catching way);

Some of the disadvantages of **relation structure form** (depending on your point of view) include:
* you cannot see the table's contents; (but often, you don't need to;)
* you cannot see specific domain information (and you cannot even infer it, except from the column's name)
* you cannot see how relations **relate** to each other (unless you add some lines or additional notation, which WE SOMETIMES WILL... more on that later.)

We will be using relation structure form frequently -- it is very convenient as a first format for a database **design** (or a database **schema**), even though it lacks some important information you'll need before you are done with the database design/schema.

Create-table Form

The THIRD format we'll use is to use the actual statements that one can use to **CREATE** 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**".

I'll defer the details of this statement until the lab to be posted later this week, but here is an example of such a statement, for the above table:

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

Some of the advantages of **create-table form** include:

* you can see the basic relation structure;
* 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 SQL CODING STANDARD -- to explicitly declare a primary key in the create-table statement for ALL relations.
* you can see some specific domain information;
* even though you do not see it in this example, you can see how relations **relate** to each other (and after we talk about how this is done, then you will also be REQUIRED to indicate such relationships when they exist).

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

* it is a little less straightforward; it is easier to type than tabular form, but harder to type than relation structure form
* you still cannot see the table's contents (but, again, often, you don't need to;)

There are surely many **OTHER** ways to represent relations as well (consider Access; it 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 tables containing user data this semester.

[USER DATA aside: introduction to the importance of good table design

We will often discuss relations in terms of their structure (without discussing their contents), because one set of relations may often NOT be as good as another for a given purpose… even with the same data, **how** you structure the relations can make a HUGE difference in usability, long-term **data integrity**, etc.

That is, relations can be well- or poorly structured, well- or poorly-designed!

Consider the relation:

Student1(**StuNum**, StuLastName, StuPhone, AdvisorID, AdviserLastName, AdviserPhone)

vs. the pair of relations:

Student2(**StuNum**, StuLastName, StuPhone, AdviserID)
Adviser(**AdvisorID**, AdvisorLastName, AdviserPhone)

Do you see how the first has information about 2 different "things", student and advisors? If an advisor's phone number changes, you have to change that for EVERY student having that advisor! (If an advisor has 20 students? You've got to change all 20 phone number copies!) And, notice that the advisor with 20 students has his/her phone number in that database 20 times… that 's **unnecessarily duplicated** data.

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

It turns out that, for **everyday, operational** use of data, especially data that changes frequently, it is **better** to store the relations separately, and combine them as needed using relational operations, than it is to store them as a combined table; (this may NOT be true for data **warehouses**. But, that is a topic for another lecture.)

Also notice: **all** data duplication has **not** disappeared --- AdviserID's appear in a number of places, and more than once. But I'll be arguing that that is **necessary** data duplication, in this case, to allow for **integration** of the data.

SECOND MAIN ELEMENT of a DATABASE: metadata

Remember, **metadata** is 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 **tables**, these often store the metadata in the form of tables, too --- sometimes called **system tables**. For example, the DBMS might include a system table giving the **tables** in the database --- for example,

SysTables (**Table-Name**, Number-of-Columns, Primary-Key)

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

SysColumns (**Column-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 data 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 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**. You would define a physical index for frequently-searched columns, for example, so that the DBMS would include additional data to make searching those columns more efficient. For example, you might define a physical index for a student's LastName column, to make searching by last name more efficient, or you might define a physical index for a student's Major column, to searching by major 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 the **Student** table is updated. (This is not necessarily **awful**, but indexes are **not free** either, and "should be reserved for cases where they are truly needed".)

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 DBMS's support "application components", and of those that do, not all store their structure in the database. But you have probably encountered some DBMS's that do this -- an Access database, for example, has tools for making reports, and the resulting report specifications are part of the 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.